An Investigation Into The Structure Of Chinese Remainder Representation

Thesis submitted by
David Anthony LAING
in September 2009

for the degree of Doctor of Philosophy
in the School of Business and Information Technology
James Cook University

Statement of access

I, the undersigned, author of this work, understand that James Cook University will make this thesis available for use within the University Library and, via the Australian Digital Theses network, for use elsewhere.

I understand that, as an unpublished work, a thesis has significant protection under the Copyright Act and;

I do not wish to place any further restriction on access to this work.

____ ______________

Signature Date

Statement of sources

DECLARATION

I declare that this thesis is my own work and has not been submitted in any form for another degree or diploma at any university or other institution of tertiary education. Information derived from the published or unpublished work of others has been acknowledged in the text and a list of references is given.

______________________________ ____________

Signature Date

Statement of contribution of others

This works was supported financially by a part-time stipend provided by the James Cook University School of Maths, Physics and Information Technology and by a write-up grant provided by the James Cook University School of Business and Information Technology.

My supervisor, Bruce Litow, was co-author for both of the accompanying papers.

Acknowledgements

This work would not have been possible without the guidance and mentoring of my supervisor, Bruce Litow.

I cannot imagine how I would have completed the thesis without the motivation and support provided by my partner, Laisa Teleni.

I am also very grateful to Ian Atkinson and my colleauges in the APAC and ARCHER projects, both for the experience and for the associated good times.

Abstract

This dissertation began as an investigation into the pseudorank function for Chinese remainder representation (CRR) integers and its relation to the rank function. Through an innovative reformulation of the problem we discovered an alternative pseudorank function which made the study of pseudorank errors significantly easier. Prior to this work almost nothing was known about these errors. The alternative pseudorank lead us to the discovery of a set of integers which are related to many of the interesting CRR properties.

One of the drivers of CRR research is the fact that it can be used as the basis for highly performant arithmetic in hardware implementations. For a long time a number of CRR related problems have been recognized as being hard to implement efficiently. The lack of an efficient implementation for some of these problems has meant that CRR has only been able to be used to implement hardware solutions for very specific problems.

The second part of the thesis defines a model of computation that can be used to clearly divide the difficult CRR problems from the easy CRR problems. This work resulted in the establishment of a link between difficult CRR problems and NP-complete problems.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Overview

The research reported in this dissertation concerns basic structural properties of Chinese remainder representation systems (CRRS) and a connection between these properties and computational complexity theory. The research represents a notable departure both in outlook and method from much of the CRRS literature. Even previous complexity theory oriented work is unlike the research reported on here, which combines empirical and analytical approaches to the study of CRRS. This combination has resulted in a new way to think about the problem of computing integer comparison when restricted to CRRS data. Although CRRS research dates back to the 1960's, there appears to be no serious attempt prior to this work to develop the necessary terminology and concepts which make possible a detailed analysis what it really means to say that certain operations are "easy" to compute in a CRRS-intrinsic manner and others are "hard".

Our introduction of finite automata (FA) as a CRRS computation model makes possible a uniform treatment of all of the discussions of "easy" and "difficult" problems in the CRRS literature. The FA approach led us to a connection with NP-completeness.

1.2 Structure of the thesis

The thesis begins with Chapter 2, which defines and explores the basic properties of numbers in Chinese remainder representation systems (CRRS). Amongst these properties are the rank and the pseudorank, which is an approximation to the rank. This chapter includes the details of several experiments which were performed to further our understanding of when the rank and pseudorank differ. The results of these experiments provide the more detailed information about the differences between the rank and the pseudorank than any other effort to date.

A subtle change to the pseudorank led to the discovery of a new and well behaved set of CRRS integers and the chapter concludes with the description and exploration of this new set and how it relates to the other CRRS properties.

Chapter 3 discusses finite automata and their application to the study of CRRS. Finite automata allow us to easily ascertain whether certain types of problems are “difficult” or not.

Chapter 4 defines the notion of “difficult” problems in CRRS and then demonstrates a link between the complexity of these difficult problems and NP-completeness. The chapter includes a review of the CRRS literature that dealt with these “difficult” problems. A number of these difficult problems are described and they are shown to all require a similar amount of computational power to compute.

Finally, the contents of Chapter 5 describes an investigation into the distribution of the different values of the rank. The distribution was less amenable to complete analysis than the other problems that we covered in the thesis, and so the focus of the chapter is on the empirical methods and approximations that were applied.

Appendix A discusses the theoretical and practical aspects of an arbitrary precision polynomial multiplication algorithm, including benchmarks for the implementation. The algorithm is due to Nussbaumer and has similarities to the Schönhage-Strassen algorithm. The selection of the Nussbaumer algorithm came from work done prior to this thesis, and so the focus of the appendix is narrowed to the Nussbaumer algorithm and comparison with standard polynomial multiplication algorithms. The benchmarks show that it begins to outperform Karatsuba multiplication for even reasonably small polynomials.

Appendix B consists of two published papers on our CRRS work, which report census results for the older version of pseudorank.

A library was developed for experimenting with CRRS, FA and polynomials. Both the library and the code for the experiments that we performed make heavy use of the GMP library for multiple precision arithmetic. The source for our code library and various experiments is available from `http://dlaing.org/projects/phd.html`.

1.3 Notation

Define $x = |b|_c$ to be the least non-negative integer x such that $x \equiv b \bmod c$. We usually employ the $|b|_c$ notation, but occasionally $x \equiv b \bmod c$ will also be used.

The floor and ceiling functions follow Iverson’s convention, such that

$$\lfloor x \rfloor$$

is the greatest integer less than or equal to x and

$$\lceil x \rceil$$

is the least integer greater than or equal to x.

Let

$$a = \|x\|$$

denote that a is the absolute value of x.

Let

$$A = \{a, b, c\}$$

defines a set containing the elements a, b and c, and let

$$s = |A|$$

be the number of elements in the set A whenever A is finite.

To filter the members based on some condition we have

$$A = \left\{ set \;\middle|\; condition \right\}$$

For example, if

$$D = \{1, 2, 3, 4, 5\}$$

and

$$E = \left\{ x \in D \;\middle|\; |x|_2 = 1 \right\}$$

then

$$E = \{1, 3, 5\}$$

The Iverson bracket

$$p = [condition]$$

is used to convert conditions into integers using the convention that

$$[a = a] \Leftrightarrow 1$$

and

$$[a \neq a] \Leftrightarrow 0$$

Note that

$$\left|\left\{ x \in A \;\middle|\; p(x) \right\}\right| = \sum_{x \in A} [p(x)]$$

Let

$$B(z)$$

be a polynomial B in the indefinite variable z, let

$$B(z)[i]$$

be the ith coefficient of B and let

$$|B(z)|$$

be the degree of B.

We then have

$$B(z) = \sum_{i=0}^{|B(z)|} B(z)[i] \cdot z^i$$

Finally, let

$$\log_k x$$

be the logarithm of x with base k and let

$$\log x$$

be the logarithm of x with base 2.

Chapter 2

Chinese Remainder Representation

2.1 An introduction to CRR

2.1.1 Definitions and uses

A Chinese remainder representation system (CRRS) is principally defined by a set of r moduli. We use p_i for $i = 1, \ldots, r$ to denote these moduli, which are mutually co-prime. In this work and in most cases the moduli will be chosen from the primes.

Each CRRS has a size, $P = \prod_{i=1}^{r} p_i$ and can encode the integers in the range $0 \leq x < P$ into the r residues $\langle x \rangle_i$ for $i = 1, \ldots, r$. The weak form of Chinese remainder representation (CRR) uses $\langle x \rangle_i = |x|_{p_i}$ and converts the residues back to an integer via the extended GCD algorithm.

The strong form of CRR uses

$$\langle x \rangle_i = \left| x \cdot \left(\frac{P}{p_i} \right)^{p_i - 2} \right|_{p_i}$$

which leads to a more efficient way to convert the residues to an integer. Unless otherwise stated we use the strong form of CRR throughout this work.

Euler's theorem states that $\left| a^{\phi(b)} \right|_b = 1$ for co-prime a and b. When all p_i are prime $\frac{P}{p_i}$ and p_i will be co-prime and we will have $\phi(p_i) = p_i - 1$, so $\left| \left(\frac{P}{p_i} \right)^{p_i - 1} \right|_{p_i} = 1$.

This means that $\left| \frac{P}{p_i} \cdot \left(\frac{P}{p_i} \right)^{p_i - 2} \right|_{p_i} = 1$, so $\left(\frac{P}{p_i} \right)^{p_i - 2}$ is the modular inverse of $\frac{P}{p_i}$ modulo p_i.

The integer x is recovered from the residues using the strong form of CRR with

$$x = \left| \sum_{j=1}^{r} \frac{P}{p_j} \cdot \langle x \rangle_j \right|_P \tag{2.1}$$

For each modulus p_i we have

$$\left|\left|\sum_{j=1}^{r} \frac{P}{p_j} \cdot \langle x \rangle_j\right|_P\right|_{p_i} = \left|\frac{P}{p_i} \cdot \langle x \rangle_i\right|_{p_i} = \left|\frac{P}{p_i} \cdot x \cdot \left(\frac{P}{p_i}\right)^{p_i-2}\right|_{p_i} = |x|_{p_i}$$

and the r values $|x|_{p_i}$ uniquely specify x in the range $0 \leq x < P$.

The parameters of a CRRS can be chosen to guarantee that the CRRS will be able to support n-bit numbers. This is a common enough requirement that we use n as the primary measure of CRRS problem size when discussing time and memory bounds.

Hardy and Wright [30] give the bound $\prod_{2<p<2n} p = 2^{\Theta(n)}$ where p runs over the primes. From this we use the primes from 3 to $2n-1$ as the moduli. The prime number theorem, also discussed by Hardy and Wright, means we can expect to find $r = \Theta(n/\log n)$ primes in this range.

One of the main advantages of working in a CRRS is that most arithmetic is easy to parallelize since addition, subtraction and multiplication can be carried out on each modulus independently. This works for addition and subtraction because $|\langle x \rangle_i + \langle y \rangle_i|_{p_i} = \langle x + y \rangle_i$ and works for multiplication because $\left|\frac{P}{p_i} \cdot \langle x \rangle_i \cdot \langle y \rangle_i\right|_{p_i} = \langle x \cdot y \rangle_i$ and $\left|\frac{P}{p_i}\right|_{p_i}$ is small and pre-computable.

This is especially useful in the design of hardware for arithmetic, as the moduli are at most $O(\log n)$-bits in size and with r processors each modulus can be updated in parallel. Groβchadl [27] and Yen et al [72], for example, have implemented the RSA encryption algorithm in hardware using CRR arithmetic.

The drawback is that all of the arithmetic operates in the ring of integers modulo P. If this is undesirable then either the CRRS must be large enough that no result will exceed P or some method for detecting overflow will be required.

It is not immediately clear how to detect when CRRS arithmetic overflows in an efficient manner. There are several problems that are equivalent or very similar to overflow detection.

The standard method for dealing with signed integers in CRRS is to designate the positive integers as those in the range $0 \leq x < P/2$ and the negative integers as those in the range $P/2 < x < P$, where the negative value is $x - P$. This reduces sign detection to comparison with the fixed integer $\lfloor P/2 \rfloor$, which can also be used to detect overflow in the output of CRR arithmetic operations.

Chapter 4 has the details on the relationship between these and other computationally equivalent problems.

If we rearrange Equation 2.1 we get

$$x + P \cdot q\left(x\right) = \sum_{i=1}^{r} \frac{\langle x \rangle_i}{p_i} \tag{2.2}$$

We refer to $q\left(x\right)$ as the rank, which we define as

$$q\left(x\right) = \sum_{i=1}^{r} \frac{\langle x\rangle_i}{p_i} - \frac{x}{P} \tag{2.3}$$

and it is clear that the rank is the number of times the CRRS overflows when converting the residues to an integer in fixed-radix form. We can see that $0 \leq q\left(x\right) < r$.

The computation of the rank is as difficult as detecting overflow or sign of CRRS numbers. This is because the rank is computed - either directly or indirectly - when numbers in CRRS are converted back to integers in fixed-radix form, and overflow and sign detection in that form is trivial.

One of the earliest references to the rank is found in a paper by Akushskii, Burcev and Pak [3], who refer the rank as the “core” function. Whether it is called the “core” or “rank” function is largely a moot point, as the bulk of the CRR literature define their own independent notation. The only thing that appears to approach a convention is that the quantity which we refer to as P is normally represented be a capitalized letter and the quantities p_i are normally referred to by lower case letters.

Using CRR allows us to carry out arithmetic on $O(\log n)$-bit numbers where we would normally be dealing with n-bit numbers. Computing the rank is problematic since it involves arithmetic operations with n-bit operands, so if the rank or an equivalent computation needs to be carried out regularly than the benefits of using CRR can be diminished or even negated.

Szabó was one of the first to examine the intrinsic difficulty of these problems [65]. He showed that for sign detection all of the information in the ith residue is required for each $p_i < \sqrt{P}$. The conclusion was that no simple or fast method for sign detection could exist for non-redundant CRRS.

Several different approaches to computing the rank and equivalent problems are discussed in Chapter 4, and almost all of them make use of redundant CRRS. Redundant CRRS are created by adding a redundant set of moduli to the CRRS or by defining a range of numbers to be a “dead zone”, in which no valid integers can occur.

The alternative to redundant CRRS are to work with a lossy but well behaved approximation to one of these equivalent problems and attempt to work around the deficiencies of the approximation. We discuss one of the more common approximations in the following section.

Further details on the theory and use of CRR can be found in the standard text on the subject by Tanaka and Szabó [66].

2.1.2 The pseudorank, the good and the bad

We use $\tilde{q}\left(x\right)$ to denote the pseudorank of x, an approximation of the rank using $O(\log n)$-bit integers.

For some $S = O(n)$ we use

$$\mu_i(x) = \left\lfloor S \frac{\langle x \rangle_i}{p_i} \right\rfloor \tag{2.4}$$

as an approximation to $\langle x \rangle_i / p_i$.

We then set

$$\alpha(x) = \sum_{i=1}^{r} \mu_i(x) \tag{2.5}$$

from which we get the pseudorank

$$\tilde{q}\,(x) = \lfloor \alpha(x)/S \rfloor \tag{2.6}$$

As the problem size is n, it is clear that the pseudorank can be computed in polynomial time.

Van Vu discusses the use of pseudorank to implement efficient sign detection in hardware [69] and as a result the methods he describes are designed to get maximum utility from binary adders and lookup tables while trying to avoid modular operations with large and arbitrary moduli, as occur in Equation 2.3. While describing these techniques, Van Vu shows that the pseudorank is accurate for all x when S is at least $\lceil r \cdot n \rceil$ bits in length.

This is simply part of the error analysis for Van Vu, but as we will discuss in Chapter 4 we are concerned with what we can achieve using $O(\log n)$-bit integers.

We can use the accuracy of the pseudorank to partition the integers into the set of "good" integers, $\mathcal{G}$, and the set of "bad" integers, $\mathcal{B}$, where

$$\left\{ x \in \mathcal{G} \;\middle|\; q\,(x) = \tilde{q}\,(x) \right\} \qquad\qquad \left\{ x \in \mathcal{B} \;\middle|\; q\,(x) \neq \tilde{q}\,(x) \right\}$$

We use $S = 2^g$ where g is the least integer such that $2^g > 4 \cdot r$, which gives a pseudorank that guarantees $\left\{ x \in \mathcal{G} \;\middle|\; P/4 < x < P \right\}$. We show why this is so by examining the error introduced by the pseudorank.

Let $\epsilon_i(x) = 2^g \langle x \rangle_i / p_i - \mu_i(x)$ be the error introduced by the rounding that occurs in $\mu_i(x)$, with $0 \leq \epsilon_i(x) < 1$. We rearrange this to get $\langle x \rangle_i / p_i = \mu_i(x)/2^g + \epsilon_i(x)/2^g$.

We then note that

$$\sum_{i=1}^{r} \frac{\langle x \rangle_i}{p_i} = q(x) + \frac{x}{P} = \sum_{i=1}^{r} \frac{\mu_i(x)}{2^g} + \sum_{i=1}^{r} \frac{\epsilon_i(x)}{2^g}$$

Let $\sum_{i=1}^{r} \epsilon_i(x)/2^g = \epsilon(x)$ and note that

$$0 \leq \epsilon(x) \leq \frac{r}{2^g} < \frac{1}{4} \tag{2.7}$$

Since

$$\sum_{i=1}^{r} \frac{\mu_i(x)}{2^g} = \tilde{q}\,(x) + \frac{a}{2^g} \text{ where } 0 \leq a < 2^g$$

we have

$$q(x) - \tilde{q}\,(x) + \frac{x}{P} - \frac{a}{2^g} = \epsilon(x) \tag{2.8}$$

We know that $q(x) \geq \tilde{q}\,(x)$ since $\tilde{q}\,(x)$ approximates $q(x)$ and the only errors will be the result of using floor operations in the place of regular division.

We can see what happens when $q(x) - \tilde{q}\,(x) \geq 2$ by calculating the minimum possible value of $\epsilon(x)$ from Equation 2.8. This value by occurs when $q(x) - \tilde{q}\,(x) = 2$, $x/P = 0$ and $a/2^g = 1$, and we have $\epsilon(x) > 1$.

Since this contradicts the bounds on $\epsilon(x)$ in Equation 2.7 we have shown that

$$0 \leq q(x) - \tilde{q}\,(x) \leq 1 \tag{2.9}$$

A similar approach will demonstrate that $q(x) = \tilde{q}\,(x)$ for $P/4 < x < P$. If $q(x) - \tilde{q}\,(x) = 1$, $x/P > 1/4$ and $a/2^g = 1$ we have $\epsilon(x) > 1/4$, which contradicts the bounds in Equation 2.7.

The pseudorank can be used in a practical setting by treating integers in the range $0 \leq x < P/4$ as invalid and adjusting the arithmetic used in the CRRS accordingly. This results in the CRRS having redundant information about the integers that are in use. We will discuss redundant information in CRRS at a later point in this work.

2.1.3 An empirical look at the bad integers

Very little was known about the integers $x \in \mathcal{B}$ except that they occur in the "critical region", where $0 \leq x \leq \lfloor P/4 \rfloor$.

A program was written to count the bad x for all $0 \leq x < P/4$ by calculating the rank and the pseudorank, the results of which are in Table 2.1

r	P	bad	density
3	105	4	0.038095238095238099
4	1155	78	0.067532467532467527
5	15015	743	0.049483849483849485
6	255255	16336	0.063998746351687522
7	4849845	382249	0.078816745689810702
8	111546435	10459313	0.093766448026779164
9	3234846615	176060621	0.054426265586629677

Table 2.1: Bad counts and densities for several small CRRS.

The critical region is exponential in the CRRS problem size $n = \log P$, so we cannot count or otherwise examine the bad for even modest sized CRRS in a reasonable amount of time.

Figure 2.1 is a sketch of how the bad integers are distributed for a small CRRS, created by calculating the density of the bad integers for a fixed window around each value of x.

The graph might give some intuition about how the bad x are distributed we cannot produce

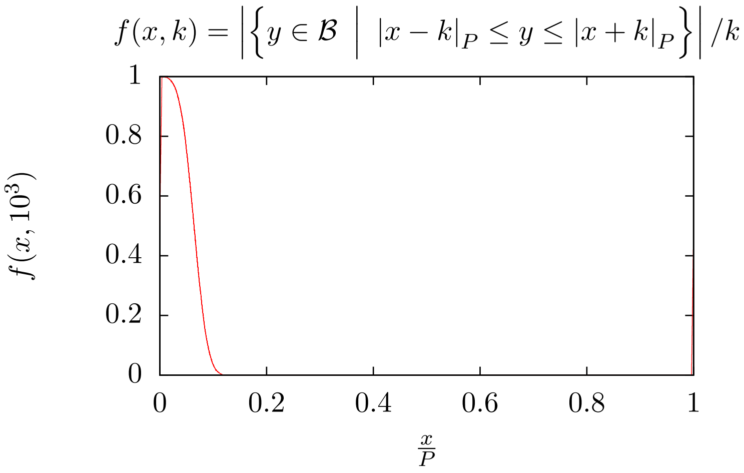


Figure 2.1: Sampled density of $\mathcal{B}$ with $\lceil \log P \rceil = 18$.

graphs for large enough CRRS to shape this intuition to the point where it might be useful.

It was clear that looking at every value in range of the CRRS was not going to scale to larger CRRS and so other methods were investigated.

2.2 The first census experiment

The direct census of the bad integers was unable to extend beyond meagre CRRS. A census algorithm was devised to give an approximation to $|\mathcal{B}| = \sum_{x=0}^{P-1} [x \in \mathcal{B}]$ with a running time that is polynomial in n. This census algorithm appears to be the first of its kind in analyzing CRRS.

2.2.1 Theory

In the description of the algorithm we will use $e(x)$ to denote the complex exponential function $e^{i\pi x}$.

The census algorithm computes

$$|\hat{\mathcal{B}}| = Re(S), \qquad S = \sum_{x=0}^{\lfloor P/4 \rfloor} [x \in \mathcal{B}] \cdot e\left(x/P\right)$$

We have $Re(e(x/P)) = \cos(\pi x/P)$ and since $1/\sqrt{2} \leq \cos(\pi x/P) \leq 1$ for $0 \leq x \leq \lfloor P/4 \rfloor$ we get

$$\frac{1}{\sqrt{2}} |\mathcal{B}| \leq |\hat{\mathcal{B}}| \leq |\mathcal{B}| \tag{2.10}$$

as an initial error bound for the census.

Note that for $0 \leq \alpha, \beta \leq 1$ the following sums can be precomputed:

$$\sum_{x=0}^{\lfloor \beta P \rfloor} e\left(x/P\right) = \frac{e\left(\frac{\lfloor \beta P \rfloor + 1}{P}\right) - 1}{e\left(\frac{1}{P}\right) - 1}$$
$$\sum_{x=\lfloor \alpha P \rfloor}^{\lfloor \beta P \rfloor} e\left(x/P\right) = \sum_{x=0}^{\lfloor \beta P \rfloor} e\left(x/P\right) - \sum_{x=0}^{\lfloor \alpha P \rfloor} e\left(X/P\right)$$

This gives us the means to convert the census of the bad integers in the critical region to a census of the good integers in the critical region, since any x has to be either good or bad and

$$\sum_{x=0}^{\lfloor P/4 \rfloor} [x \in \mathcal{G}] \cdot e\left(x/P\right) = \sum_{x=0}^{\lfloor P/4 \rfloor} e\left(x/P\right) - \sum_{x=0}^{\lfloor P/4 \rfloor} [x \in \mathcal{B}] \cdot e\left(x/P\right)$$

The algorithm begins with the calculation of the polynomials

$$A_i(z) = \sum_{j=0}^{p_i - 1} e\left(\frac{\langle j \rangle_i}{p_i}\right) z^{\mu_i(j)}$$

for $1 \leq i \leq r$ and from them the polynomial

$$A(z) = \prod_{i=1}^{r} A_i(z)$$

We have $0 \leq \mu_i(x) < S$ for all values of x and i. This means that the degree of each $A_i(z)$ is bounded by $|A_i(z)| < 2^g$, and as a result $|A(z)| < 2^g r$.

The polynomial $A_i(z)$ is related to the modulus p_i, containing a term for each of the values that a residue of that modulus can attain. The result of a polynomial multiplication contains every combination of the terms in the operand polynomials, although grouped into significantly fewer terms, and so we know that each value $0 \leq x < P$ contributes to the polynomial $A(z)$.

We have

$$\prod_{i=1}^{r} e\left(\langle x \rangle_i / p_i\right) = e\left(x/P + q\left(x\right)\right) \text{ and } \sum_{i=1}^{r} \mu_i(x) = \alpha(x)$$

Letting $A(z)[i]$ refer to the coefficient of the ith term of $A(z)$ as we get

$$A(z)[i] = \sum_{x=0}^{P-1} [i = \alpha(x)]\, e\left(x/P + q\left(x\right)\right)$$

Recalling that $\tilde{q}\,(x) = \lfloor \alpha(x)/2^g \rfloor$ and noting that $e(-\tilde{q}\,(x)) = -1^{\tilde{q}(x)}$ we compute

$$\begin{aligned} T &= \sum_{i=0}^{2^g r} -1^{\lfloor i/2^g \rfloor} \cdot A(z)[i] \\ &= \sum_{x=0}^{P-1} e\,(x/P + q\,(x) - \tilde{q}\,(x)) \\ &= \sum_{x=0}^{P-1} [x \in \mathcal{G}]\, e\,(x/P) - \sum_{x=0}^{P-1} [x \in \mathcal{B}]\, e\,(x/P) \end{aligned}$$

We are interested in the $x \in \mathcal{B}$, so

$$T - \sum_{x=0}^{P-1} e\,(x/P) = -2 \cdot \sum_{x=0}^{P-1} [x \in \mathcal{B}]\, e\,(x/P)$$

is of interest to us.

This yields

$$S = \left(\sum_{x=0}^{P-1} e\,(x/P) - T \right) /2$$

the real value of which computes our census of bad integers for a CRRS.

2.2.2 Practice

It is clear that implementing this algorithm will require a non-trivial amount of work. This work is made substantially easier through the use of the open source GMP library [1] which we use for all arbitrary precision calculations.

We use complex arbitrary precision rational arithmetic throughout the computation of the census. These complex exponential function is approximated with a Taylor series truncated to h terms and a rational approximation to π. These values are our primary means of controlling the amount of error, and we will shortly show how these values should be set to achieve a particular error bound.

We define the replacement for $e\,(a/b)$ as

$$\hat{e}_h\,(a/b) = \sum_{k=0}^{h} \frac{\left(i \dfrac{\hat{\pi}_n}{\hat{\pi}_d} \dfrac{a}{b} \right)^k}{k!} = \sum_{k=0}^{h} \frac{i^k \cdot \hat{\pi}_n^k \cdot a^k}{\hat{\pi}_d^k \cdot b^k \cdot k!}$$

where $\hat{\pi} = \hat{\pi}_n / \hat{\pi}_d$ is our rational approximation to π.

We use the notation $\hat{k}$ to refer to an approximation to k, and so we will use $\hat{A}_i(z)$, $\hat{A}(z)$, $\hat{T}$, $\hat{U}$ and $\hat{S}$ when describing various other approximations.

The multiplication of the polynomials $\hat{A}_i(z)$ is simplified by scaling the coefficients of the polynomials terms so that the multiplication is carried out with integers instead of rationals. The scaling is performed by multiplying each of terms in $\hat{A}_i(z)$ by $\hat{\pi}_d^h \cdot {p_i}^h \cdot h!$ and the reverse scaling is carried out by dividing $\hat{A}(z)$ or $\hat{T}$ by $\hat{\pi}_d^{h \cdot r} \cdot P^h \cdot h!^r$.

This reverse scaling could be applied immediately after the polynomial multiplication but we instead apply it after $\hat{T}$ is computed. This saves $2^g r - 1$ divisions.

The deferred scaling also means that the additions and subtractions performed in the computation of $\hat{T}$ are performed on integers. This is particularly useful when working with the GMP rational primitives, which attempt to canonicalize the results of every operation performed on rationals by dividing the numerator and denominator by their GCD.

The speed of the polynomial multiplication is important as it dominates the running time of the algorithm. The standard FFT-based approaches were unacceptable due to a combination of issues involving running time, memory usage and accuracy. A search through the relevant literature found the Nussbaumer polynomial multiplication algorithm. It is similar to the Schönhage-Strassen algorithm in that it uses integer arithmetic and is fast and accurate, but has the benefit that it out performs competing algorithms for even moderately sized inputs.

The algorithm is detailed in Appendix A, which describes the notation we use when dealing with polynomials and provides an overview of the theory behind the Nussbaumer polynomial multiplication as well as an analysis of the time and memory requirements of the algorithm.

In short, multiplying two polynomials with degree 2^t and m-bit coefficients takes time $O(2^t \cdot t \cdot \log t) \cdot O(m)$, and requires between $\left(4 \cdot 2^t + 8 \cdot 2^{\lceil t/2 \rceil}\right) O(m)$ and $\left(4 \cdot 2^t + 16 \cdot 2^{\lceil t/2 \rceil}\right) O(m)$ bits of memory.

Appendix A also includes a benchmarks for several of the implementation options.

2.2.3 Error bounds

Approximating $e^{i\pi x}$ with a truncated Taylor series and a rational approximation to π introduces errors. We examine these errors by looking at the absolute error introduced when the polynomials $A_i(z)$ are approximated with $\hat{A}_i(z)$.

For some polynomial K, let $L(K) = \sum_{j=0}^{|K|} \|K[j]\|$. We set the polynomial σ such that $L(A_i - \hat{A}_i) < L(\sigma)$ for $1 \leq i \leq r$.

Since $-1 < \hat{e}_h(j/p_i) \leq 1$ for $0 \leq j < p_i$ we know that $L(A_i) < p_i$, which means that $L(\hat{A}_i) < p_i + L(\sigma)$. Noting that $L(J \cdot K) \leq L(J) \cdot L(K)$, we have

$$L(A) \leq \prod_{i=1}^{r} A_i \leq \prod_{i=1}^{r} p_i = P$$

All we need to do is find a bound on $L(\hat{A})$. It is clear that

$$L(\hat{A}) \leq \prod_{i=1}^{r} L(\hat{A}_i) \leq \prod_{i=1}^{r} (p_i + L(\sigma))$$

but that is not useful to us in this form.

A partial binomial expansion hints at the path we will take:

$$\prod_{i=1}^{r}(p_i + L(\sigma)) = P + \sum_{i=1}^{r}\frac{P}{p_i}\cdot L(\sigma) + \sum_{i=1}^{r-1}\sum_{j=i+1}^{r}\frac{P}{p_i\cdot p_j}\cdot L(\sigma)^2 + \cdots + \sum_{i=1}^{r} p_i \cdot L(\sigma)^{r-1} + L(\sigma)^r$$

and we develop this into

$$L(\hat{A}) \leq \sum_{k=0}^{r}\binom{r}{k}\cdot L(\sigma)^k \cdot \frac{P}{3^k}$$

and so

$$L(A-\hat{A}) \leq \sum_{k=1}^{r}\binom{r}{k}\cdot L(\sigma)^k \cdot \frac{P}{3^k}$$

If $L(\sigma) < 1/n^2$ then we note that $\binom{r}{k} < n^k$ and see that

$$L(A-\hat{A}) \leq P \cdot \sum_{k=1}^{r}\frac{1}{(3n)^k} < \frac{P}{n}$$

which is a tight enough bound for our purposes.

We now show that we can achieve the bound $L(\sigma) < 1/n^2$ using $h = O(\log n)$ and an $O(\log n)$-bit value for $\hat{p_i}$.

We want

$$\sum_{j=0}^{p_i-1}\|e(j/p_i) - \hat{e_h}(j/p_i)\| < L(\sigma) < 1/n^2 \text{ for } 1 \leq i < r$$

Since $p_r < n$, this means that we need

$$\max\|e(a) - \hat{e_h}(a)\| < 1/n^3 \text{ where } 0 \leq a < 1$$

We start by looking at the error introduced by the truncation of the Taylor series.

$$E_{TRUNC}(h) = \sum_{k=h+1}^{\infty}\left\|\frac{(i\cdot\hat{\pi}\cdot a)^k}{k!}\right\| \leq \sum_{k=h+1}^{\infty}\left\|\frac{(\hat{\pi}\cdot a)^k}{k!}\right\|$$

Assume that our approximation to π is good enough that $\hat{\pi} < 4$. We know that the error introduced by each truncated term will be

$$T_{TRUNC}(k) = \frac{\hat{\pi}^k}{k!} < \frac{4^k}{k!}$$

Now consider what happens when

$$k > 32 \cdot e$$

From Stirling's approximation

$$\left(\frac{k}{e}\right)^k < k!$$

we have

$$T_{TRUNC}(k) < \frac{4^k}{k!} < 4^k \cdot \left(\frac{e}{32 \cdot e}\right)^k$$

and so

$$T_{TRUNC}(k) < \frac{1}{2^{3k}}$$

Since

$$E_{TRUNC}(h) = \sum_{k=h+1}^{\infty} T_{TRUNC}(k) < \sum_{k=h+1}^{\infty} \frac{1}{2^{3k}}$$

and because

$$\sum_{k=1}^{\infty} \frac{1}{2^k} = 1$$

we have

$$E_{TRUNC}(h) < \frac{1}{2^{3h-1}}$$

If we then set

$$h > 32 \cdot e \cdot \log n + 1/3$$

we get

$$E_{TRUNC}(h) < \frac{1}{n^3}$$

This supports our claim that $h = O(\log n)$ is compatible with the bound $L(\sigma) < 1/n^2$.

Next we will look at the error introduced by our approximation to π.

$$E_\pi = \max(\left\| e^{i \cdot \hat{\pi} \cdot a} - e^{i \cdot \pi \cdot a} \right\|) \text{ for } 0 \leq a < 1$$

Since

$$\left\| e^{i \cdot \theta} - 1 \right\| \leq \|\theta\| \text{ for real } \theta$$

we have

$$E_\pi \leq \|\hat{\pi} - \pi\|$$

This means that a $O(\log n)$-bit approximation to π is all that is required to set

$$E_\pi < \frac{1}{n^3}$$

and enforce the error bound $L(\sigma) < 1/n^2$.

The error introduced by the pre-calculated complex exponential function is hidden by the asymptotic notation as it is negligible when compared to the errors produced by the approximation to $A(z)$.

2.2.4 Time and memory bounds

The polynomial multiplication uses integer coefficients with a maximum size of $h! \cdot \hat{\pi}_d^{h \cdot r} \cdot P^h$. We have

$$r = \Theta(n/\log n), \quad h = O(\log n) \text{ and } \hat{\pi}_d = O(\log n)$$

so

$$\log h! = O(\log n \log\log n), \quad \log \hat{\pi}_d^{h \cdot r} = O(n \log n) \text{ and } \log P^h = O(n \log n)$$

which means we need at most $O(n \log n)$ bits for each coefficient.

The polynomial multiplication algorithm we are using is more efficient when the polynomials being multiplied have similar sizes.

We accommodate this in our analysis of the running time of the census by treating the r-fold multiplication as being performed over $\lceil \log r \rceil$ rounds, where the kth round sees $r/2^k$ pairwise multiplications of polynomials with degree at most $2^{g+k-1} < 2^k \cdot 4r$.

Since it takes $O(d \cdot \log d \cdot \log\log d)$ time to multiply polynomials of degree d we have

$$\sum_{k=1}^{\lceil \log r \rceil} r/2^k \cdot 2^k \cdot 4r \cdot \log(2^k \cdot 4r) \cdot \log\log(2^k \cdot 4r) = O(r^2 \cdot (\log r)^2 \cdot \log\log r)$$

which gives the upper bound on the running time of the census as

$$O(r^2 \cdot (\log r)^2 \cdot \log\log r) \cdot A(n \log n)$$

where $A(b)$ is the time required to multiply 2 b-bit integers.

The polynomial multiplications are performed sequentially, and so to derive a bound on the amount of memory required we only need to consider the multiplication with the largest operands.

The last pair of polynomials to be multiplied will have degree at most $r \cdot 2^{g-1} < 4r^2$, as $|A| < r \cdot 2^g$. Therefore the census requires enough working memory array to store M_r integers of size $O(n \cdot \log n)$, where M_r is bounded by

$$M_r \leq 16 \cdot r^2 + 64 \cdot r$$

and so the census will use $O(r^2 \cdot n \cdot \log n)$ bits of memory as working space.

2.2.5 Results

Figure 2.2 shows the approximate bad densities in the region from 0 to $\lfloor P/4 \rfloor$ for $3 \leq r \leq 63$. The sawtooth pattern is a result of setting $S = 2^g > 4 \cdot r$ and our approximation of the pseudorank error $\epsilon < 1/4$.

To see why this is so consider the relative sizes of r and 2^g as r changes value from 2^t to $2^{t+1} - 1$. When $r = 2^{t+1} - 1$ using g-bit integers for the pseudorank is just enough to guarantee that all

$x \in \mathcal{G}$ for $\lfloor P/4 \rfloor < x < P$. So when $r = 2^t$ we have nearly an extra bit of precision, and the pseudorank is accurate for nearly twice as many integers.

This can be seen in Figure 2.2, as the lower edge of the sawtooth pattern is defined by the $r = 2^t$, and the upper edge is defined by the $r = 2^{t+1} - 1$.

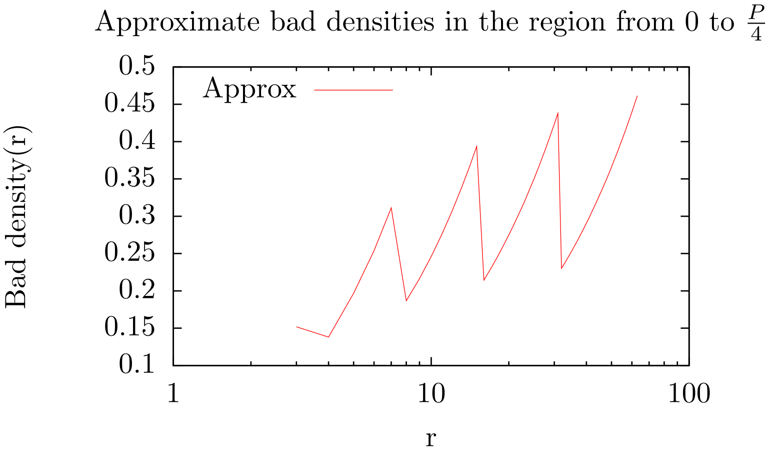


Figure 2.2: Approximate bad densities in the region from 0 to $\frac{P}{4}$.

We used the bounds $\epsilon < r/2^g < 1/4$ to define the critical region earlier. In this case we should narrow the critical region to $0 \leq x < r \cdot P/2^g$ in order to attain a clearer view of the bad density as r varies.

Figure 2.3 shows the approximate bad densities in the region from 0 to $\lfloor rP/2^g \rfloor$ for $3 \leq r \leq 63$. This gives a more accurate view of the relative numbers of good and bad in the critical region as r increases.

The empirical data this provides points toward a bad density in the region $0 \leq x \leq \lfloor rP/2^g \rfloor$ that asymptotically approaches $1/2$ as r becomes large, which seems compatible with the data we gathered for Figure 2.1.

The census algorithm was run with different values of h and $\hat{\pi}$. We use $h = 20, 40$ and $\hat{\pi} = \hat{\pi}_{small}, \hat{\pi}_{large}$. The rational number $\hat{\pi}_{small}$ has a 24-bit denominator and approximates π to within 1.61×10^{-14}, and $\hat{\pi}_{large}$ has a 52-bit denominator and approximates π to within 1.22×10^{-16}.

The effect on the output for these different configurations is too small to display in Figure 2.3. If the irregularities in the curve are not the result of the choices of h and $\hat{\pi}$ there are several possible explanations. It is possible that this is an intrinsic part of the relationship between r and the bad density, although it is more likely that the irregularities are errors introduced when we used $Re(e(x))$ as an approximation to x for $0 \leq x < 1/4$.

We assume that the census that used $h = 40$ and $\hat{\pi}_{large}$ would be the most accurate. Figure 2.4 shows the absolute error between the census run with that configuration and the census run with each of the other combinations of values for h and $\hat{\pi}$. The differences between the results for the census run with $h = 20, \hat{\pi}_{small}$ and the census run with $h = 20, \hat{\pi}_{large}$ were not large

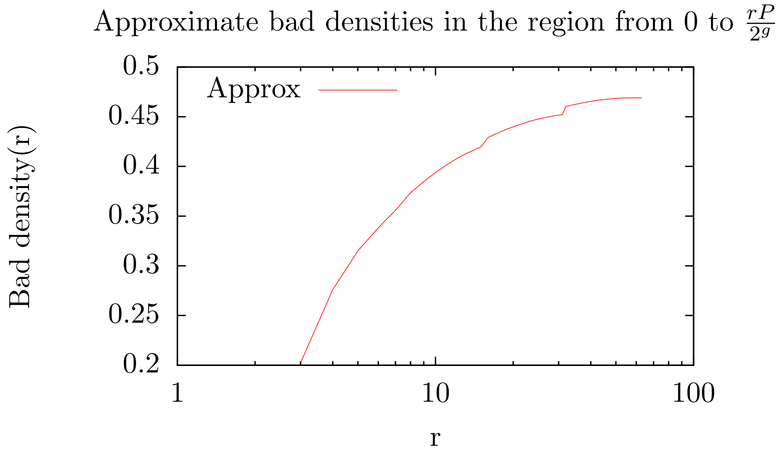


Figure 2.3: Approximate bad densities in the region from 0 to $\frac{rP}{2^g}$.

enough to be displayed.

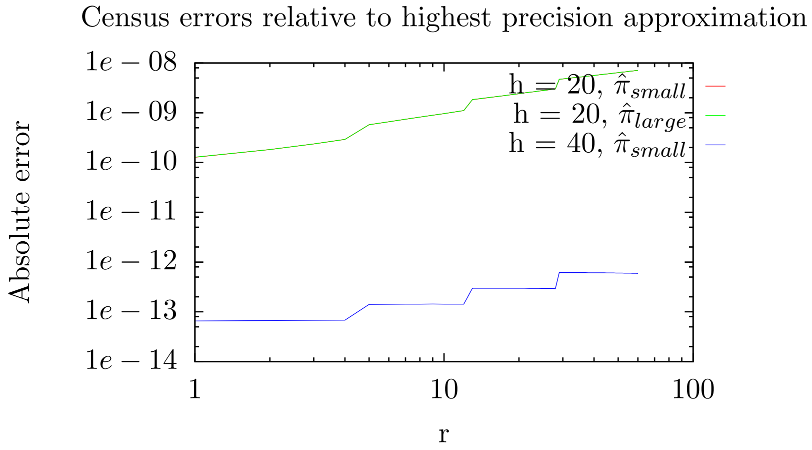


Figure 2.4: Census errors relative to h = 40, $\hat{\pi}_{large}$.

The census algorithm was significantly faster than the simple count of bad integers, although the processing time for larger values of r was still significant. It was memory rather than time that was the limiting factor in this case.

One of the previous implementations of this algorithm included the ability to store to parts of the data to disk when not in use, which was an inefficient way to trade time for memory but allowed us to gather more data than we otherwise would have been able to. A better implementation would have used the target systems memory mapping facilities to manage this, however progress from a theoretical perspective made this algorithm obsolete before we had the opportunity to make such a change.

2.3 Properties and predicates

2.3.1 Bins

At this point we took a closer look at the conditions which determine if a given x is good or bad.

We start by expanding the pseudorank equation

$$\tilde{q}\left(x\right) = \left\lfloor \left(\sum_{i=1}^{r} \left\lfloor 2^g \frac{\left\langle x \right\rangle_i}{p_i} \right\rfloor \right) /2^g \right\rfloor$$

We rearrange the equation, making use of the fact that $\left\lfloor \frac{a}{b} \right\rfloor = \frac{a - |a|_b}{b}$

$$\begin{aligned}
\tilde{q}\left(x\right) &= \left\lfloor \left(\sum_{i=1}^{r} \frac{2^g \left\langle x \right\rangle_i}{p_i} - \frac{\left|2^g \left\langle x \right\rangle_i\right|_{p_i}}{p_i} \right) /2^g \right\rfloor \\
\tilde{q}\left(x\right) &= \sum_{i=1}^{r} \frac{\left\langle x \right\rangle_i}{p_i} - \frac{\left|2^g \left\langle x \right\rangle_i\right|_{p_i}}{2^g p_i} - \left| 2^g \sum_{i=1}^{r} \frac{\left\langle x \right\rangle_i}{p_i} - \sum_{i=1}^{r} \frac{\left|2^g \left\langle x \right\rangle_i\right|_{p_i}}{p_i} \right|_{2^g} /2^g \\
2^g \tilde{q}\left(x\right) &= 2^g \sum_{i=1}^{r} \frac{\left\langle x \right\rangle_i}{p_i} - \sum_{i=1}^{r} \frac{\left|2^g \left\langle x \right\rangle_i\right|_{p_i}}{p_i} - \left| 2^g \sum_{i=1}^{r} \frac{\left\langle x \right\rangle_i}{p_i} - \sum_{i=1}^{r} \frac{\left|2^g \left\langle x \right\rangle_i\right|_{p_i}}{p_i} \right|_{2^g}
\end{aligned}$$

Since $\sum_{i=1}^{r} \left\langle x \right\rangle_i / p_i = q(x) + x/P$ we have

$$2^g \tilde{q}\left(x\right) = 2^g q\left(x\right) + \frac{2^g x}{P} - q(|2^g x|_P) - \frac{|2^g x|_P}{P} - \left| 2^g \left(q(x) + \frac{x}{P} \right) - q(|2^g x|_P) - \frac{|2^g x|_P}{P} \right|_{2^g}$$

and we use $\left\lfloor \frac{a}{b} \right\rfloor = \frac{a - |a|_b}{b}$ in the other direction to get

$$2^g \tilde{q}\left(x\right) = 2^g q\left(x\right) + \left\lfloor \frac{2^g x}{P} \right\rfloor - q(|2^g x|_P) - \left| \left\lfloor \frac{2^g x}{P} \right\rfloor - q(|2^g x|_P) \right|_{2^g}$$

We know that $\left\lfloor \frac{2^g x}{P} \right\rfloor - q(|2^g x|_P)$ can never exceed 2^g because $0 \leq \left\lfloor \frac{2^g x}{P} \right\rfloor < 2^g$.

Using this we can see that if $\left\lfloor \frac{2^g x}{P} \right\rfloor \geq q(|2^g x|_P)$ we have

$$\left| \left\lfloor \frac{2^g x}{P} \right\rfloor - q(|2^g x|_P) \right|_{2^g} = \left\lfloor \frac{2^g x}{P} \right\rfloor - q(|2^g x|_P)$$

and so $\tilde{q}\left(x\right) = q\left(x\right)$.

Conversely, if $\left\lfloor \frac{2^g x}{P} \right\rfloor < q(|2^g x|_P)$ we have

$$\left| \left\lfloor \frac{2^g x}{P} \right\rfloor - q(|2^g x|_P) \right|_{2^g} = 2^g + \left\lfloor \frac{2^g x}{P} \right\rfloor - q(|2^g x|_P)$$

and so $\tilde{q}\,(x) = q\,(x) - 1$.

This is another demonstration of the fact that either $q\,(x) = \tilde{q}\,(x)$ or $q\,(x) = \tilde{q}\,(x) + 1$.

The bad are precisely the x for which $\left\lfloor \frac{2^g x}{P} \right\rfloor < q(|2^g x|_P)$, and since $0 \leq q(|2^g x|_P) < r$ we have also found an alternative way of showing that $x \in \mathcal{B}$ only in the range $0 \leq x < \left\lfloor \frac{rP}{2^g} \right\rfloor$. In fact, we have tightened the bound and have $x \in \mathcal{B}$ only in the range $0 \leq x < \left\lfloor \frac{(r-1)P}{2^g} \right\rfloor$.

It can help to consider the $\left\lfloor \frac{2^g x}{P} \right\rfloor$ part of the condition as a separation of the integers into a series of sub-ranges, which we refer to as bins.

We say that x is in the kth bin if

$$k \leq \left\lfloor \frac{2^g x}{P} \right\rfloor < k + 1$$

or

$$\left\lfloor \frac{kP}{2^g} \right\rfloor \leq x < \left\lfloor \frac{(k+1)P}{2^g} \right\rfloor$$

The quantity $\frac{P}{2^g}$ is used frequently enough that it is convenient to define $R = \frac{P}{S}$, which gives bin widths of $\lfloor R \rfloor$.

The bad integers occur in the bins $k = 0, \cdots, r-2$ when $q(|2^g x|_P) > k$, corresponding to darker regions in Figure 2.5. We use $L = \left\lfloor \frac{(r-1)P}{2^g} \right\rfloor = \lfloor rR \rfloor$ for the width of the critical region.

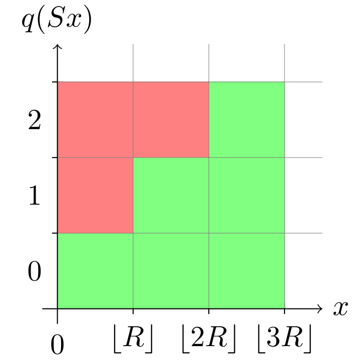


Figure 2.5: The relationship between $\mathcal{G}$, $\mathcal{B}$ and the bins when $r = 3$.

2.3.2 Pseudoparity

We introduce a new property of an integer x in a CRRS. Looking at the parity of x we see that

$$\begin{aligned} |x|_2 &= \left| P \cdot q(x) + \sum_{i=1}^{r} \frac{P}{p_i} \langle x \rangle_i \right|_2 \\ &= \left| q(x) + \sum_{i=1}^{r} \langle x \rangle_i \right|_2 \end{aligned}$$

We refer to

$$|x|_{\tilde{2}} = \left| \tilde{q}\left(x\right) + \sum_{i=1}^{r} \left\langle x \right\rangle_i \right|_2 \tag{2.11}$$

as the pseudoparity of x.

Note that

$$|x|_{\tilde{2}} = |x|_2 \text{ when } x \in \mathcal{G}$$

and

$$|x|_{\tilde{2}} \neq |x|_2 \text{ when } x \in \mathcal{B}$$

Recall that the problem size is n and that the pseudorank is defined such that it is computable with time polynomial in n. Since we already have the values of $\left\langle x \right\rangle_i$ when working with integers in CRR we see that the pseudoparity is also computable in polynomial time.

2.3.3 Notation for constructed sets

We introduce a notation to describe the set of CRRS integers $\{s\}$ in terms of the n adjacent subsets $s_1, s_2, \ldots, s_{n-1}, s_n$.

We use w_i to denote the width of the range of s that s_i describes. We have $\sum_{i=1}^{n} w_i = P$ and s_i describes s in the range $t_i \leq x < |t_i + w_i|_P$ where $t_i = \sum_{j=0}^{i-1} w_j$.

For each s_i in the description of s

- when $s_i = G$, $\left\{ x \in s \;\middle|\; |x - t_i|_P \in \mathcal{G} \,\cap\, t_i \leq x < |t_i + w_i|_P \right\}$ with $w_i = L = \lfloor rR \rfloor$
- when $s_i = B$, $\left\{ x \in s \;\middle|\; |x - t_i|_P \in \mathcal{B} \,\cap\, t_i \leq x < |t_i + w_i|_P \right\}$ with $w_i = L = \lfloor rR \rfloor$
- when $s_i = 0^k$, $\left\{ x \notin s \;\middle|\; t_i \leq x < |t_i + w_i|_P \right\}$ with $w_i = k$
- when $s_i = 1^k$, $\left\{ x \in s \;\middle|\; t_i \leq x < |t_i + w_i|_P \right\}$ with $w_i = k$
- when $s_i = 0^*$, $\left\{ x \notin s \;\middle|\; t_i \leq x < |t_i + w_i|_P \right\}$ with $w_i = P - \sum_{\substack{0 \leq j < n \\ j \neq i}} w_j$
- when $s_i = 1^*$, $\left\{ x \in s \;\middle|\; t_i \leq x < |t_i + w_i|_P \right\}$ with $w_i = P - \sum_{\substack{0 \leq j < n \\ j \neq i}} w_j$

There can be at most 1 of 0^* or 1^* in any set description. If a particular set description has $\sum_{i=1}^{n} w_i < P$, then $\{s\} = s_1 s_2 \ldots s_{n-1} s_n$ is taken to be an abbreviation of $s_1 s_2 \ldots s_{n-1} s_n 0^*$

2.3.4 The set GB

Recall that $|x|_{\tilde{2}} = |x|_2$ when $q(x) = \tilde{q}\left(x\right)$ and $|x|_{\tilde{2}} = 1 - |x|_2$ when $q(x) \neq \tilde{q}\left(x\right)$, and that $L = \left\lfloor \frac{(r-1)P}{2^g} \right\rfloor$ is the width of the region in which bad integers can occur.

We can construct the set GB with

$$\left\{x \in GB \;\middle|\; |x|_{\tilde{2}} \neq \left||x - L|_{\tilde{2}} + L\right|_2\right\} \tag{2.12}$$

As a consequence of defining the set in terms of pseudoparity operations we see that testing for membership in GB can be carried out efficiently.

For the following discussion we assume that $|L|_2 = 0$ and so

$$\left\{x \in GB \;\middle|\; |x|_{\tilde{2}} \neq |x - L|_{\tilde{2}}\right\} \tag{2.13}$$

We need to show that the set GB described in Equation 2.12 is consistent with our set notation, such that

$$\{x \in GB\} = \left\{x \in \mathcal{G} \;\middle|\; 0 \leq x < L\right\} \cup \{|x - L|_P \in \mathcal{B}\} \tag{2.14}$$

Proof. This can be demonstrated by considering 3 different ranges of $0 \leq x < P$.

Claim 1: $x \notin GB$ in the range $2 \cdot L \leq x < P$.

Since $x \in \mathcal{G}$ and $|x - L|_P \in \mathcal{G}$, both terms have $|y|_{\tilde{2}} = |y|_2$. As $|L|_2 = 0$ we see that Equation 2.13 cannot be satisfied by any value in this range.

Claim 2: $\left\{x \in GB \;\middle|\; |x - L|_P \in \mathcal{B}\right\}$ in the range $L \leq x < 2 \cdot L$.

We have $x \in \mathcal{G}$ and know that $\left\{x \notin GB \;\middle|\; |x - L|_P \in \mathcal{G}\right\}$ for the same reason that $x \notin GB$ in the first range. If $|x - L|_P \in \mathcal{B}$ then $\left||x - L|_P\right|_{\tilde{2}} \neq \left||x - L|_P\right|_2$ and so $x \in GB$.

Note that this is the only range for which $|x - L|_P \in \mathcal{B}$ and so the values in this range are the only values that can satisfy the second part of Equation 2.14.

Claim 3: $\left\{x \in GB \;\middle|\; x \in \mathcal{G}\right\}$ in the range $0 \leq x < L$.

We have $|x - L|_P \in \mathcal{G}$. In the previously mentioned ranges this would mean $\left\{x \in GB \;\middle|\; x \in \mathcal{B}\right\}$ and $\left\{x \notin GB \;\middle|\; x \in \mathcal{G}\right\}$.

The situation is reversed in this range. For $0 \leq x, y < P \leq x + y$ we have $\left||x|_P + |y|_P\right|_2 \neq \left||x + y|_P\right|_2$. This occurs because $|P|_2 = 1$, and the result is $\left\{x \in GB \;\middle|\; x \in \mathcal{G}\right\}$.

□

Figure 2.6 is a characterization of the set GB in terms of the possible values of $q\left(|2^g x|_P\right)$ in each bin, and Figure 2.7 gives an empirical view of GB.

2.3.5 Building sets with GB

We can build variations of the set GB by generalizing Equation 2.12.

The basic variations occur through translation, dilation and inversion.

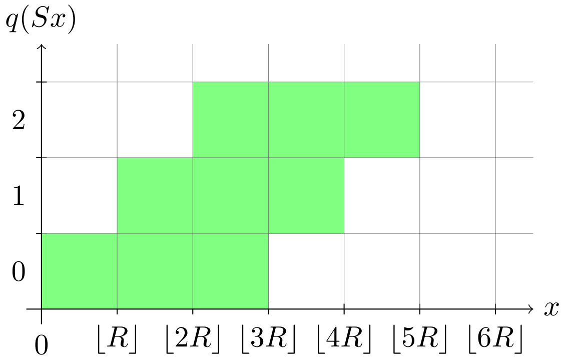


Figure 2.6: The relationship between GB and the bins when $r = 3$.

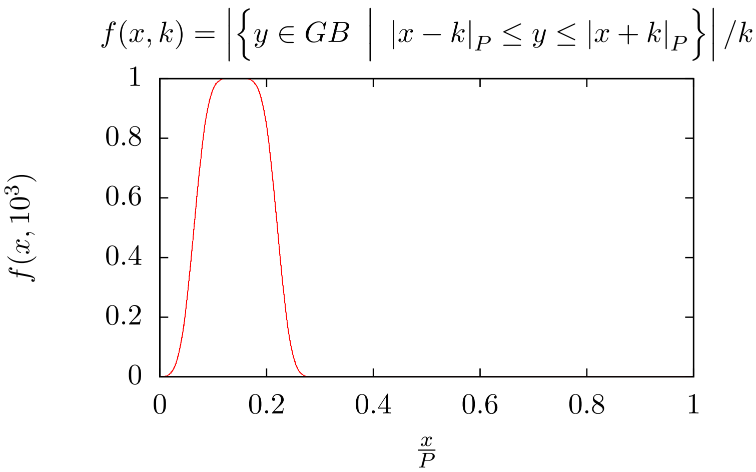


Figure 2.7: Sampled density of GB with $\lceil \log P \rceil = 18$.

We refer to the set GB translated by a and dilated by b as $GB(a, b)$, which is constructed with a modification of Equation 2.12

$$\left\{ x \in GB(a,b) \;\middle|\; ||x-a|_P|_{\tilde{2}} \neq ||x-a-L-b|_{\tilde{2}} + |L+b|_P|_2 \right\} \tag{2.15}$$

and specifies the set $0^a G 1^b B 0^*$.

Figure 2.8 is an example of translation, Figure 2.9 is an example of dilation, and Figure 2.10 demonstrates translation and dilation together.

We use $\overline{GB}$ and $\overline{GB}(a, b)$ to denote the inversion of GB and $GB(a, b)$ respectively. This is as simple as changing the condition in Equation 2.12 from inequality to equality.

The inversion of the set $GB(a, b)$ is always equivalent to some non-inverted set where

$$\overline{GB}(a,b) = GB(a+b+L, P-b-2L)$$

Inversion is demonstrated in 2.11, which sketches the inversion of 2.10.

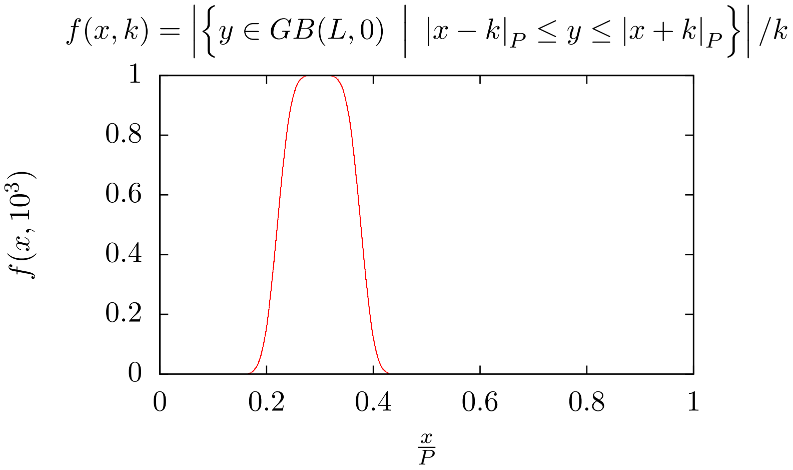


Figure 2.8: Sampled density of $GB(L,0)$ with $\lceil \log P \rceil = 18$.

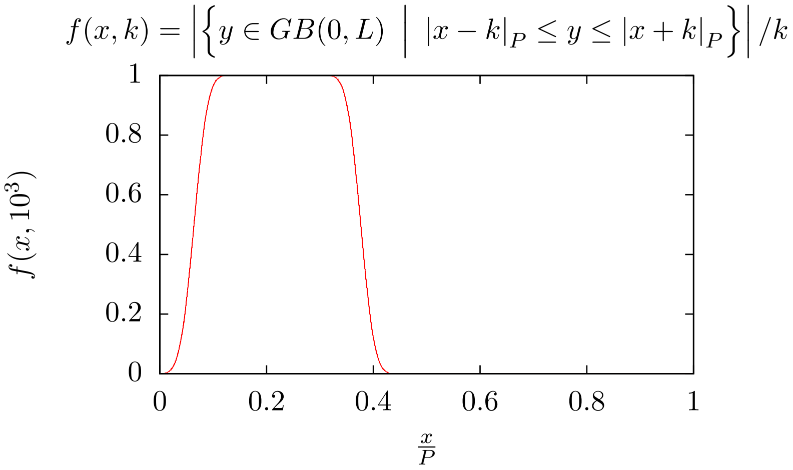


Figure 2.9: Sampled density of $GB(0,L)$ with $\lceil \log P \rceil = 18$.

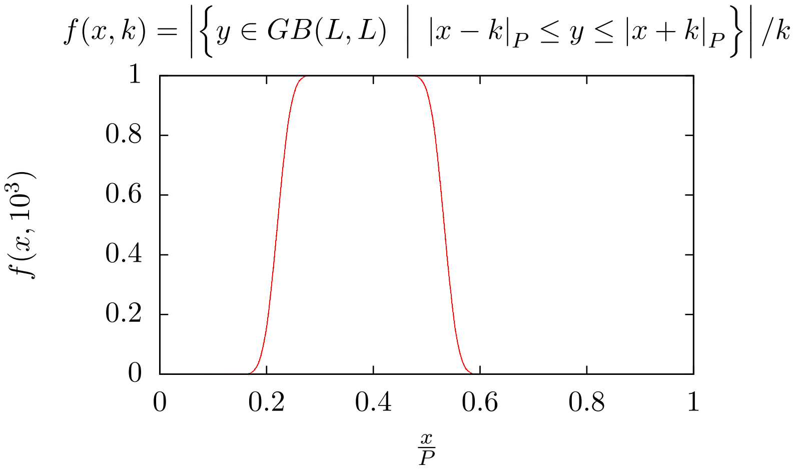


Figure 2.10: Sampled density of $GB(L,L)$ with $\lceil \log P \rceil = 18$.

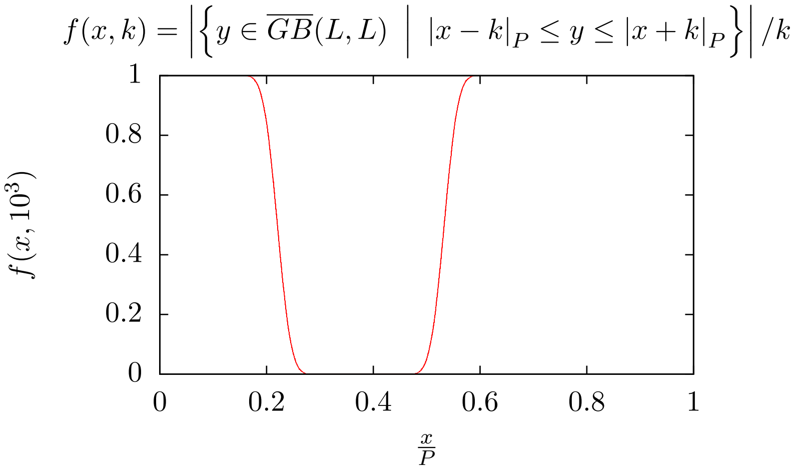


Figure 2.11: Sampled density of $\overline{GB}(L, L)$ with $\lceil \log P \rceil = 18$.

New sets can also be built by taking the union or intersection of existing sets.

It is useful to note that $G \cup B \approx 1^w$ and $G \cap B \approx 0^w$ with $w = L$. These are given as approximate since discrepancies in the rounding may produce $O(1)$ artifacts at the boundaries.

This is not normally an issue, as up to $O(n^k)$ artifacts can be dealt with efficiently by handling those edge cases explicitly for some constant k, but this does require that some care be taken during set manipulation.

The first version of GB we used had $L = \lfloor P/4 \rfloor$. This required even more care and while $G \cup B \approx 1^w$ and $G \cap B \approx 0^w$ still holds the w is hard to determine, which complicate further manipulations. The shift to using $L = \left\lfloor \frac{(r-1)P}{2^g} \right\rfloor$ was a welcome change.

2.3.6 Adding rank

We mention two additional facts about the rank that were discovered during this period of the research.

Lemma 2.1.

$$q\left(|x+y|_P\right) = q\left(x\right) + q\left(y\right) + \left[x + y \neq |x+y|_P\right] - \sum_{i=1}^{r} \left[\langle x \rangle_i + \langle y \rangle_i \neq \langle |x+y|_P \rangle_i\right]$$

Proof. We begin by manipulating $q\left(|x+y|_P\right)$

$$\begin{aligned}
q\left(|x+y|_P\right) &= \sum_{i=1}^{r} \frac{\langle |x+y|_P \rangle_i}{p_i} - \frac{|x+y|_P}{P} \\
&= \sum_{i=1}^{r} \frac{\langle |x+y|_P \rangle_i}{p_i} - \frac{x}{P} - \frac{y}{P} + \left[x+y \neq |x+y|_P\right] \\
&= \sum_{i=1}^{r} \frac{\langle x \rangle_i + \langle y \rangle_i}{p_i} - \frac{x}{P} - \frac{y}{P} + \left[x+y \neq |x+y|_P\right] - \sum_{i=1}^{r} \left[\langle x \rangle_i + \langle y \rangle_i \neq \langle |x+y|_P \rangle_i\right]
\end{aligned}$$

Note that

$$q(x) + q(y) = \sum_{i=1}^{r} \frac{\langle x \rangle_i + \langle y \rangle_i}{p_i} - \frac{x}{P} - \frac{y}{P}$$

The combination of this and the expansion of $q\left(|x+y|_P\right)$ proves the lemma.

□

Lemma 2.2.

$$q(P-x) = r - 1 - q(x) - \sum_{i=1}^{r} \left[\langle x \rangle_i = 0\right]$$

Proof. Consider

$$q(x) + q(P-x) = \sum_{i=1}^{r} \frac{\langle x \rangle_i}{p_i} + \sum_{i=1}^{r} \frac{\langle P-x \rangle_i}{p_i} - \frac{x}{P} - \frac{P-x}{P}$$

Since $|P|_{p_i} = 0$ we get

$$q(q) + q(P-x) = \sum_{i=1}^{r} \frac{\langle x \rangle_i + \langle p_i - x \rangle_i}{p_i} - 1$$

We also have

$$\langle x + (p_i - x) \rangle_i = \langle x \rangle_i + \langle p_i - x \rangle_i - \left[x + (p_i - x) \neq |x + (p_i - x)|_{p_i}\right] \cdot p_i$$

so if $\langle x \rangle_i = 0$ then $\langle x \rangle_i + \langle p_i - x \rangle_i = 0$ and if $\langle x \rangle_i \neq 0$ then $\langle x \rangle_i + \langle p_i - x \rangle_i = p_i$.

From this we get

$$\frac{\langle x \rangle_i + \langle p_i - x \rangle_i}{p_i} = \left[\langle x \rangle_i \neq 0\right]$$

which concludes the proof.

□

2.4 The second census experiment

The first census algorithm was heavily memory bound and gave inexact results. This led to a search for alternatives, and a small number of key insights led to an algorithm which gave exact results and has significantly smaller time and memory requirements.

2.4.1 Theory

Recall that $q(x) = \tilde{q}(x)$ when $x \in \mathcal{G}$ is and $q(x) = \tilde{q}(x) + 1$ when $x \in \mathcal{B}$. We can see that the number of bad integers is given by

$$|\mathcal{B}| = \sum_{x=0}^{P-1} q(x) - \sum_{x=0}^{P-1} \tilde{q}(x) \tag{2.16}$$

We can use the definition of the rank from Equation 2.3 to expand the first sum in Equation 2.16.

$$\begin{aligned} \sum_{x=0}^{P-1} x &= \sum_{x=0}^{P-1} \sum_{i=1}^{r} \frac{\langle x \rangle_i}{p_i} - \sum_{x=0}^{P-1} \frac{x}{P} \\ &= \sum_{i=1}^{r} \sum_{x=0}^{P-1} \frac{\langle x \rangle_i}{p_i} - \frac{P-1}{2} \\ &= \sum_{i=1}^{r} \frac{P}{p_i} \sum_{j=0}^{p_i-1} \frac{j}{p_i} - \frac{P-1}{2} \\ &= \sum_{i=1}^{r} \frac{P}{p_i} \cdot \frac{p_i - 1}{2} - \frac{P-1}{2} \\ &= \frac{P \cdot (r - \sum_1^r 1/p_i)}{2} - \frac{P-1}{2} \end{aligned} \tag{2.17}$$

The expansion of the sum in Equation 2.17 makes use of Gauss' trick and a reordering of the sums to exploit the fact that over the ranges $0 \le x < P$ and $1 \le i \le r$ the value of $\langle x \rangle_i$ will take on each of the values from 0 to $p_i - 1$ exactly $\frac{P}{p_i}$ times.

The pseudorank equation is expanded in a similar manner to the expansion that occurred in Section 2.3.1 to get

$$\tilde{q}(x) = \sum_{i=1}^{r} \left(\frac{\langle x \rangle_i}{p_i} - \frac{|2^g \langle x \rangle_i|_{p_i}}{2^g p_i} \right) - \frac{\beta(x)}{2^g}$$

where

$$\beta(x) = |\alpha(x)|_{2^g} = \left| \sum_{i=1}^{r} \left\lfloor 2^g \frac{\langle x \rangle_i}{p_i} \right\rfloor \right|_{2^g}$$

We begin to expand the second part of the sum in Equation 2.16.

$$\begin{aligned}\sum_{x=0}^{P-1} \tilde{q}\left(x\right) &= \sum_{i=1}^{r}\sum_{x=0}^{P-1}\left(\frac{\left\langle x\right\rangle_i}{p_i} - \frac{\left|2^g\left\langle x\right\rangle_i\right|_{p_i}}{2^g p_i}\right) - \sum_{x=0}^{P-1}\frac{\beta(x)}{2^g}\\ &= \frac{2^g-1}{2^g}\sum_{i=1}^{r}\sum_{x=0}^{P-1}\frac{\left\langle x\right\rangle_i}{p_i} - \sum_{x=0}^{P-1}\frac{\beta(x)}{2^g}\\ &= \frac{2^g-1}{2^g}\frac{P\left(r-\sum_{i=1}^{r}1/p_i\right)}{2} - \sum_{x=0}^{P-1}\frac{\beta(x)}{2^g}\end{aligned}$$

Considering the values of $\left\langle x\right\rangle_i$ over the ranges of x and i used in Equation 2.17 helped to simplify matters, and the same observation applies in this case to the values of $\left\langle x\right\rangle_i$ as well as the values of $\left|2^g\left\langle x\right\rangle_i\right|_{p_i}$. This works because 2^g and p_i are co-prime, and so $\left|2^g\left\langle x\right\rangle_i\right|_{p_i}$ will run through the same values as $\left\langle x\right\rangle_i$, albeit in a different order.

This leads to a simplified restatement of Equation 2.16 as

$$|\mathcal{B}| = \frac{P\left(r-\sum_{i=1}^{r}1/p_i\right)}{2^{g+1}} - \sum_{x=0}^{P-1}\frac{\beta(x)}{2^g}$$

We now turn to a scheme to calculate $\sum\limits_{x=0}^{P-1}\beta(x)$.

For each $1 \le i \le r$, define the polynomial $C_i(z)$ as

$$C_i(z) = \sum_{j=0}^{p_i-1} z^{\left\lfloor 2^g\frac{j}{p_i}\right\rfloor}$$

The polynomial $C(z)$ is defined as

$$C(z) = \prod_{i=1}^{r} C_i(z)$$

and it cannot have more than $2^g r$ terms as it is the result of an r-fold product of polynomials with degree less than 2^g.

These terms come together in the r-fold product such that every number in the CRRS is represented, and so the $C(z)$ be restated as

$$C(z) = \sum_{x=0}^{P-1} z^{\alpha(x)}$$

where $\alpha(x)$ is as defined in Equation 2.5, and comes from the combination of the terms in each of the polynomials C_i with exponent $\left\lfloor 2^g\frac{\left\langle x\right\rangle_i}{p_i}\right\rfloor$.

Since multiple values of x can have the same value for $\alpha(x)$ we can express the jth coefficient of $C(z)$ as

$$C(z)[j] = \sum_{x=0}^{P-1} [\alpha(x) = j]$$

This is all we need to compute the sum of $\beta(x)$ over $0 \leq x < P$ since

$$\sum_{x=0}^{P-1} \beta(x) = \sum_{i=0}^{2^g r} |i|_{2^g} \cdot C(z)[i] \tag{2.18}$$

2.4.2 Practice

All of the calculations involved in the computation of $|\mathcal{B}|$ are trivial in terms of time and memory requirements when compared to the calculation of $\sum_{x=0}^{P-1} \beta(x)$.

We use the polynomial multiplication technique mentioned in Section 2.2.2 and described in detail in Appendix A in the calculation of $\sum_{x=0}^{P} \beta(x)$.

The polynomials used in this census algorithm all have exact integer coefficients that start with value 1. This alone causes a dramatic improvement in the time and memory requirements of the algorithm.

Polynomial multiplication can be viewed as an acyclic convolution of sequences, with the elements of the sequences being the coefficients of the polynomials. Appendix A has more on the topic, and includes a description of the cyclic convolution.

To summarize from the appendix, if $D(z) = E(z) \cdot F(z)$ then the coefficients of the cyclic convolution $\acute{D}(z)$ of length n are

$$\acute{D}(z)[i] = \sum_{k=0}^{\lfloor |D(z)|/n \rfloor} D(z)[i + kn] \text{ for } i = 0, \ldots, n-1$$

We use cyclic convolutions of length 2^g in the census algorithm to compute $C(z)$. This will mean that none of the polynomials we use in the computation will have degree larger than $2^g - 1$, and $C(z)$ will have degree $2^g - 1$ exactly, provided $r > 1$.

The same amount of information is in use so there is no saving in memory. The improvement in the running time from having smaller polynomials to multiply should outweigh the cost of having larger operands in the coefficient arithmetic, especially since this arithmetic is dealt with by the GMP library.

Part of the manipulation of $C(z)$ in Equation 2.18 can be viewed as grouping the terms of $C(z)$ by their exponents modulo 2^g, and this is exactly what the cyclic convolutions are doing for us. This is why such an optimization is possible for this census algorithm, and is also why this optimization cannot be applied to the first census algorithm.

2.4.3 Time and memory bounds

The coefficients of the polynomials are 1-bit integers before the multiplication begins and eventually grow to $O(n)$ bits in length.

We are performing a r cyclic convolutions of length 2^g. Recalling that $4 \cdot r < 2^g \leq 8 \cdot r$ will let us express the time and memory bounds in terms of r and n.

Since the number of terms does not grow the order of the multiplications does not effect the running time of the census, and so the running time will be r times the running time for a cyclic convolution of sequences with length 2^g and $O(n)$-bit integers as elements.

We have

$$r \cdot O(2^g \cdot g \cdot \log g) = O(r^2 \cdot \log r \cdot \log \log r) \cdot A(n)$$

where $A(b)$ is the time required to multiply 2 b-bit integers.

This shows that the difference in running time between this census algorithm and the first census algorithm is mostly the result of the different sized coefficients.

The convolutions are performed sequentially, so we only require enough memory to perform a single cyclic convolution of length 2^g. Note that an acyclic convolution of length n requires the same amount of memory as a cyclic convolution of length $2n$.

We see that at most we will need enough memory to store M_r integers of size $O(n)$, where M_r is bounded by

$$Mr < 8 \cdot r + 16 \cdot \sqrt{2 \cdot r}$$

This translates to a requirement of $O(r \cdot n)$ bits of memory for this census algorithm, where the first census required $O(r^2 \cdot n \cdot \log n)$ bits.

2.4.4 Results

Figure 2.12 shows the exact bad densities in the region from 0 to $\lfloor P/4 \rfloor$ for $3 \leq r \leq 63$. This provides stronger empirical evidence that the bad density in the region $0 \leq x \leq \lfloor rP/2^g \rfloor$ asymptotically approaches $1/2$ as r becomes large.

Figure 2.13 shows the results of the first and second census side by side. The visible difference between the approximate and actual values is coming from the error described in Equation 2.10.

We take a closer look at this difference in Figure 2.14, which shows the absolute error between the actual and approximate census for the 4 combinations of h and $\hat{\pi}$ that were used.

The differences caused by the values of h and $\hat{\pi}$ are dwarfed by the difference that comes from using $Re(e(x/P)) = \cos(\pi x/P)$ as an approximation to 1 for $0 \leq x < P/4$.

When $r/2^g \approx 1/4$ the midpoint of the critical region will be around $P/8$ so the error at the midpoint will be $1 - \cos(\pi/8) \approx 0.0761$ and $r/2^g = 1/8$ the midpoint will be around $P/16$ and the error at the midpoint will be $1 - \cos(\pi/16) \approx 0.0192$

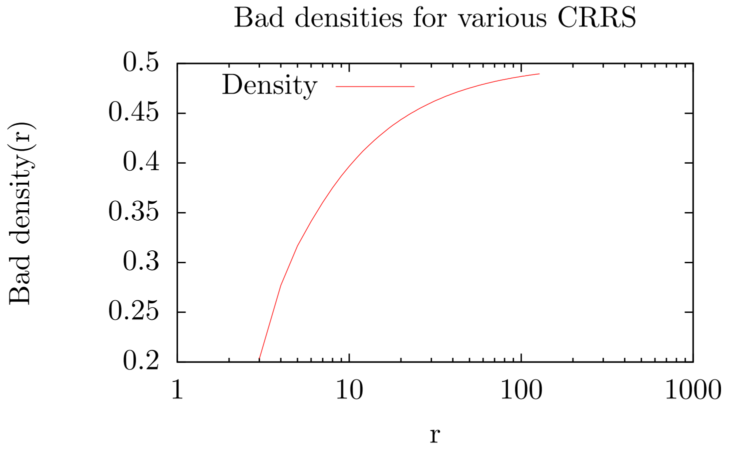


Figure 2.12: Bad densities in the region from 0 to $\frac{rP}{2^g}$.

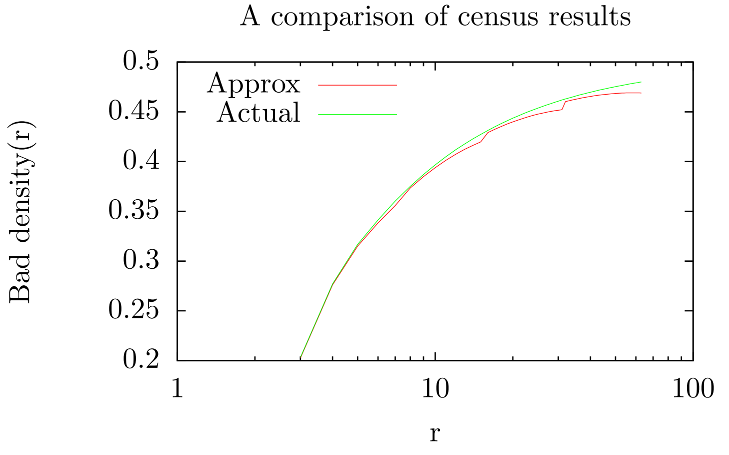


Figure 2.13: Comparison of census results.

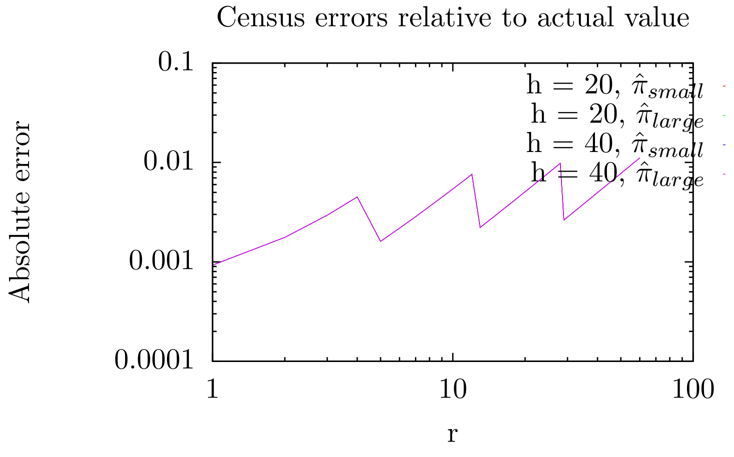


Figure 2.14: Comparison of census errors.

If the bad were uniformly distributed in the critical region then we would expect the error in Figure 2.14 to approach a sawtooth function that moved between of 0.0192 and a maximum of 0.0761. Since the error is less than that we form the hypothesis that the positions of bad integers are skewed towards the lower end of the critical region.

Again, the results appear to be compatible with Figure 2.1. This is a reassuring but informal sign that our hypothesis may not be too far from reality.

We stopped generating data for the first census algorithm when we ran out of memory. This census algorithm only uses a small amount of memory, so we stopped generating data when we thought that extra data was not worth the processing time. This is a much faster algorithm than the first census, although unsurprisingly it slows down when provided with large enough inputs.

2.5 An alternative pseudorank

2.5.1 Definition

Recall that we defined the pseudorank in terms of an approximation factor S, and then set $S = 2^g$. None of the work done in Section 2.3.1 is dependent on the value of S. The intuition was that using a value of S that is co-prime to almost every interesting parameter we deal with had the potential to complicate some of the mathematics we were doing.

Eventually we decided on setting $S = p_1$. Unless otherwise specified this is the value we use from this point forwards. We will describe the consequences of this decision shortly.

At this point in time we had also noticed that using the first r odd primes for our CRRS might be less than ideal. The sizes of each of the moduli vary considerably and so any processing that was done per moduli had inconsistent running times. The use of small primes can also mean that we may have a larger value of r than we would like when working with big CRRS.

A new set of primes solves these problems. They are the primes in the range

$$m \leq p_i < 2m$$

for some integer m.

These primes are all roughly the same size, each being an approximately $\log m$-bit integer. Large values of m will have large primes for p_1, and so we see bigger values of P for a given r than we would have achieved with the old set of primes.

Some care must be taken when experimenting with system created with similar values of m, as the sets of primes are not necessarily unique to m. Ignoring this advice may lead to long running experiments being performed on identical CRRS specified by different m.

We need to work out the size of the CRRS defined with this new set of primes. Dusart [21] provides the bound

$$\sum_{p} \|\ln\, p - m\| \leq 0.0068 \cdot m \cdot \ln\, m$$

where p runs over all of the primes $p \leq m$ and $m > 2.89 \times 10^7$.

If we set $Q(m) = \prod_p p$ we then have $2^m < Q(k) < e^{m \cdot 1.001}$. Since $P = Q(2m)/Q(m)$ we see that

$$2^{(2-1.001 \cdot \log e) \cdot m} < P < 2^{((2.002 \cdot \log e)-1) \cdot m}$$

and so $m^{O(1)} = \log^{O(1)} P$.

The number of primes still satisfies $r = O(n/\log n)$, as shown in Hardy and Wright [30].

2.5.2 Effect on current properties

The immediate changes we see are to the definition pseudorank. We now have

$$\mu_i(x) = \left\lfloor p_1 \frac{\langle x \rangle_i}{p_i} \right\rfloor$$

and

$$\tilde{q}\,(x) = \left\lfloor \left(\sum_{i=1}^{r} \mu_i(x) \right) / p_1 \right\rfloor$$

Aside from the fact that $\mu_1(x) = \langle x \rangle_1$ this does not add anything to what we already know.

We see a larger effect of this change when we look at the discussion of bins in Section 2.3.1. The terms that were $\lfloor P/2^g \rfloor$ become $\lfloor P/p_1 \rfloor$, and since p_1 divides P the floor operation is no longer necessary.

Let $R = P/p_1$, as it will be a frequently used quantity in future discussions.

This is helpful both for the description and the practicalities of the bins. There are now exactly p_1 bins and x is in the kth bin for $k \cdot R \leq x < (k+1) \cdot R$.

We also see that $|p_1 x|_P$ has the same value when x is translated to the same position in another bin, or more formally

$$|p_1 x|_P = |p_1(x + k \cdot R)|_P \;\text{ for } 0 \leq k < p_1 \tag{2.19}$$

2.5.3 The set $\mathcal{BASE}$

Throughout the course of this work we were examining the data that we had, looking for any regularities that we could exploit in order to understand or correct for the bad integers. The most important of these regularities is a set of integers we refer to as $\mathcal{BASE}$.

Define $\mathcal{BASE}$ as the set of x such that $x \in \mathcal{BASE}$ if

$$x = \sum_{i=2}^{r} R \cdot \frac{a_i}{p_i} \text{ where } 0 \leq a_i < p_i \tag{2.20}$$

We use β to denote a member of $\mathcal{BASE}$.

It is clear that $|\mathcal{BASE}| = \prod_{i=2}^{r} p_i = R$. Since the denominators in the sum $\sum_{i=2}^{r} \frac{a_i}{p_i}$ are mutually co-prime then

$$|\beta + k \cdot R|_P \notin \mathcal{BASE} \text{ for } 0 < k < p_1$$

The result of this is that any x can be represented as

$$x = \beta \pm m \cdot R \text{ with } 0 \leq m < p_1 \tag{2.21}$$

and the representation of each x will be unique. Equation 2.21 does not use modular arithmetic and so for each β there are only p_1 values of m for which $0 \leq \beta \pm m \cdot R < P$.

2.5.4 $\mathcal{BASE}$ and the bins

We manipulate Equation 2.20 to get

$$\frac{p_1 \cdot \beta}{P} = \sum_{i=2}^{r} \frac{a_i}{p_i}$$

and noting that $\langle p_1 \cdot x \rangle_1 = 0$ for all x we see that

$$q\left(|p_1 \cdot \beta|_P\right) + \frac{|p_1 \cdot \beta|_P}{P} = \sum_{i=2}^{r} \frac{\langle |p_1 \cdot \beta|_P \rangle_i}{p_i}$$

If we set $a_i = \langle |p_1 \cdot \beta|_P \rangle_i$ for $2 \leq i \leq r$, then we can combine the above equations

$$\frac{p_1 \cdot \beta}{P} = q\left(|p_1 \cdot \beta|_P\right) + \frac{|p_1 \cdot \beta|_P}{P}$$

which becomes

$$\left\lfloor \frac{p_1 \cdot \beta}{P} \right\rfloor = q\left(|p_1 \cdot \beta|_P\right)$$

The mathematics in Section 2.3.1 was independent of the value of S, and so we still have $x \in \mathcal{G}$ if and only if $\lfloor p_1 x/P \rfloor \geq q(|p_1 x|_P)$. It is clear then that all $\beta \in \mathcal{G}$, and furthermore that all $\beta + m \cdot R \in \mathcal{G}$ and all $\beta - m \cdot R \in \mathcal{B}$ for $m > 1$ and where $0 \leq \beta \pm m \cdot R < P$.

Figure 2.15 is a graphical representation of the fact that $\left\lfloor \frac{p_1 \cdot \beta}{P} \right\rfloor = q\left(|p_1 \cdot \beta|_P\right)$ for $x \in \mathcal{BASE}$.

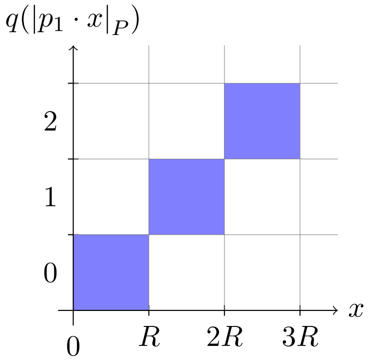


Figure 2.15: The relationship between $\mathcal{BASE}$ and the bins.

2.5.5 $\mathcal{BASE}$ and GB

The critical region in which bad can occur is reduced with the new pseudorank. Since $\langle |p_1 x|_P \rangle_1 = 0$, the maximum possible value of $q\left(|p_1 x|_P\right)$ is $r-2$, and so $L = (r-2)R$.

Looking at Figure 2.15 and Figure 2.16 suggests methods for obtaining the set $\mathcal{BASE}$ from the set GB and vice-versa.

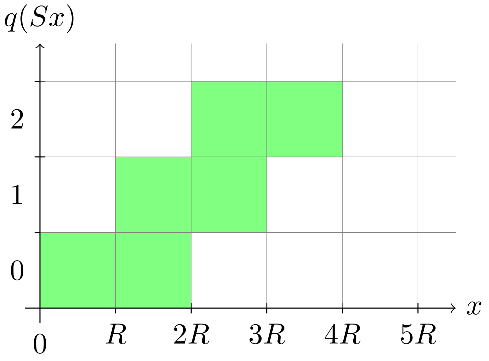


Figure 2.16: The relationship between GB and the bins when $r = 3$ and with the new pseudorank.

The set $\mathcal{BASE}$ is constructed from the set GB with

$$\{x \in \mathcal{BASE}\} = \{x \in GB\} \;\cap\; \{|x + R|_P \notin GB\} \tag{2.22}$$

and the set GB is constructed from the set $\mathcal{BASE}$ with

$$\{x \in GB\} = \bigcup_{k=0}^{r-2} \{|x - k \cdot R|_P \in \mathcal{BASE}\} \tag{2.23}$$

This also hints at a way to create the set $0^a G 1^{b\cdot R} B$

$$\left\{x \in 0^a G 1^{b\cdot R} B\right\} = \bigcup_{k=0}^{r-2+b} \{|x - a - k \cdot R|_P \in \mathcal{BASE}\} \tag{2.24}$$

although the construction $GB(a, b \cdot R)$ is more efficient.

Note that $b \geq p_1 - r + 1$ the "G" and "B" parts of the set will overlap.

A consequence of Equation 2.19 is that we no longer have to concern ourselves with rounding errors when sets are translated by multiples of R, which greatly simplifies the manipulation of most of the sets we are interested in.

2.5.6 $\mathcal{BASE}$ and pseudorank

Taking a second look at the error in the pseudorank with the new value for S revealed that the values of $\alpha(x)$ for $x \in \mathcal{BASE}$ have an interesting property.

Lemma 2.3. $|\alpha(x)|_{p_1} = 0$ *if and only if* $x \in \mathcal{BASE}$

This provides a much faster test for membership in $\mathcal{BASE}$ than using the construction from Equation 2.22. More details are provided in Chapter 3, which covers the computational aspects of all of the CRRS properties.

Proof. We begin by examining the error introduced by the approximation $\mu_i(x)$.

$$\epsilon_i(x) = \frac{p_1 \langle x \rangle_i}{p_i} - \mu_i(x) = \frac{p_1 \langle x \rangle_i}{p_i} - \left\lfloor \frac{p_1 \langle x \rangle_i}{p_i} \right\rfloor = \frac{|p_1 \langle x \rangle_i|_{p_i}}{p_i}$$

and so

$$\sum_{i=1}^{r} \frac{\langle x \rangle_i}{p_i} = q\,(x) + \frac{x}{P} = \sum_{i=1}^{r} \frac{\mu_i(x)}{p_1} + \epsilon(x)$$

where

$$\epsilon(x) = \sum_{i=2}^{r} \frac{\epsilon_i(x)}{p_1} = \sum_{i=2}^{r} \frac{|p_1 \langle x \rangle_i|_{p_i}}{p_1 \cdot p_i}$$

We have

$$\sum_{i=1}^{r} \frac{\mu_i(x)}{p_1} = \tilde{q}\,(x) + \frac{t}{p_1} \text{ where } 0 \leq t < p_1$$

and so we have something very similar to Equation 2.8 but with new $\tilde{q}\,(x)$ and $\epsilon(x)$

$$q\,(x) - \tilde{q}\,(x) + \frac{x}{P} - \frac{t}{p_1} = \epsilon(x)$$

Recall that $\alpha(x) = \sum_{i=1}^{r} \mu_i(x)$, and so $|\alpha(x)|_{p_1} = t$.

Consider the case when $x \in \mathcal{BASE}$. Since $x \in \mathcal{G}$ we have $q\left(x\right) = \tilde{q}\left(x\right)$. Multiplying by P,

$$x - t \cdot R = P \cdot \epsilon(x) = \sum_{i=2}^{r} R \cdot \frac{\left|p_1 \left\langle x \right\rangle_i\right|_{p_i}}{p_i}$$

and we see that $P \cdot \epsilon(x) \in \mathcal{BASE}$.

We have $x \in \mathcal{BASE}$ and $P \cdot \epsilon(x) \in \mathcal{BASE}$ and know that there are no $\beta + k \cdot R \in \mathcal{BASE}$. This means we must have $t = 0$ when $x \in \mathcal{BASE}$ and demonstrates that $\left|\alpha(x)\right|_{p_1} = 0$ if $x \in \mathcal{BASE}$.

To see that $x \in \mathcal{BASE}$ if $\left|\alpha(x)\right|_{p_1} = 0$, set $t = 0$ and look at $P\left(q\left(x\right) - \tilde{q}\left(x\right)\right) + x = P \cdot \epsilon(x)$. If $x \in \mathcal{B}$ we get $P + x = P \cdot \epsilon(x)$, which cannot occur since $0 \leq \epsilon(x) < r/p_1 < 1$. If $x \in \mathcal{G}$ we get $x = P \cdot \epsilon(x)$, and we know that $P \cdot \epsilon(x) \in \mathcal{BASE}$. □

Similarly, if $\left|x - t \cdot R\right|_P \in \mathcal{BASE}$ then $\left|\alpha(x)\right|_{p_1} = t$. This can be used to determine if $x \in GB$ or $x \in GB(a \cdot R, b \cdot R)$.

We get the method to determine if $x \in GB$ from Equation 2.23, which yields

$$x \in GB \Leftrightarrow 0 \leq \left|\alpha(x)\right|_{p_1} < r - 1 \tag{2.25}$$

The test for $x \in GB(a, b \cdot R)$ is then

$$x \in GB(a, b \cdot R) \Leftrightarrow 0 \leq \left|\alpha(x - a)\right|_{p_1} < r - 1 + b$$

We can use also $\left|\alpha(x)\right|_{p_1}$ to determine $\left\lfloor \frac{x}{R} \right\rfloor$, provided that we have an efficient method for computing the rank.

Lemma 2.4.

$$\left\lfloor \frac{x}{R} \right\rfloor = \left|\alpha(x) + q\left(\left|p_1 \cdot x\right|_P\right)\right|_{p_1}$$

Proof. We are after $\left\lfloor \frac{p_1 \cdot x}{P} \right\rfloor$, which is the bin containing x.

If $x \in \mathcal{BASE}$ then x is in bin $q\left(\left|p_1 \cdot x\right|_P\right)$.

We know that $\left|\alpha(x)\right|_{p_1} = m$ such that $\left|x - m \cdot R\right|_P \in \mathcal{BASE}$.

This means that x is in bin

$$\left|\alpha(x)\right|_{p_1} + q\left(\left|p_1 \cdot (x - m \cdot R)\right|_P\right)$$

and since

$$q\left(\left|p_1 \cdot x\right|_P\right) = q\left(\left|p_1 \cdot (x - m \cdot R)\right|_P\right)$$

our claim holds.

The sum is modulo p_1 because there are only p_1 bins. □

Corollary 2.5.

$$x \in \mathcal{B} \textit{ if and only if } p_1 \leq |\alpha(x)|_{p_1} + q\left(|p_1 \cdot x|_P\right)$$

Proof. Recall that $x \in \mathcal{G}$ if and only if $x = \beta + m \cdot R$ for some $\beta \in \mathcal{BASE}$ and $0 \leq \beta + m \cdot R < P$.

From Lemma 2.4 we see that $m = |\alpha(x)|_{p_1}$. If $p_1 \leq |\alpha(x)|_{p_1} + q\left(|p_1 \cdot x|_P\right)$ we know that $P \leq \beta + m \cdot R$ and hence $x \in \mathcal{B}$. □

2.5.7 $\mathcal{BASE}$ is symmetrical

Let β_* be the largest member of $\mathcal{BASE}$. From Equation 2.20 we see

$$\beta_* = \sum_{i=2}^{r} R \cdot \frac{p_i - 1}{p_i} = R \cdot \left(r - 1 - \sum_{i=2}^{r} \frac{1}{p_i}\right)$$

and that $|\beta_* - x|_P \in \mathcal{BASE}$ if and only if $x \in \mathcal{BASE}$ since for $x \in \mathcal{BASE}$

$$|\beta_* - x|_P = \sum_{i=2}^{r} R \cdot \frac{p_i - 1 - a_i}{p_i}$$

and $0 \leq p_i - 1 - a_i < p_i$.

This means that $\mathcal{BASE}$ is symmetrical, which we see empirically in Figure 2.17.

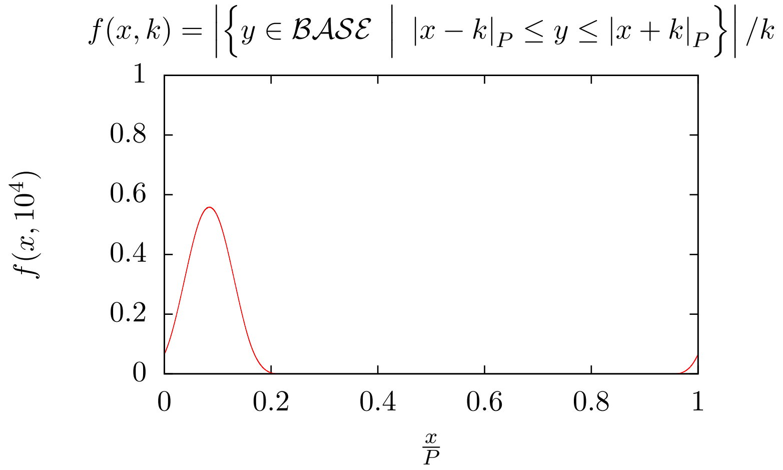


Figure 2.17: Sampled density of $\mathcal{BASE}$ with $\lceil \log P \rceil = 18$.

2.5.8 $\mathcal{BASE}$ and the bad integers

If β_* is the largest member of $\mathcal{BASE}$ then the largest bad integer must then be $W = \beta_* - R$. This is because every $x \in \mathcal{B}$ can be expressed as $\beta - m \cdot R$ for some $\beta \in \mathcal{BASE}$ and some $m \geq 1$.

We can use that to show that for $0 \leq x \leq W$ if $x \in T$ then $W - x \in \bar{T}$ where $T = \{\mathcal{G}, \mathcal{B}\}$.

We begin with $x \in \mathcal{G}$

$$x = \beta + k \cdot R = R \cdot \left(k + \sum_{i=2}^{r} \frac{a_i}{p_i}\right) \text{ for } 0 \leq k < p_i \text{ and } 0 \leq x < P$$

and since

$$W = R \cdot \left(-1 + \sum_{i=2}^{r} \frac{p_i - 1}{p_i}\right)$$

we have

$$W - x = R \cdot \left(-1 - k + \sum_{i=2}^{r} \frac{p_i - 1 - a_i}{p_i}\right)$$

As $0 \leq p_i - 1 - a_i < p_i$ we see that

$$\sum_{i=2}^{r} \frac{p_i - 1 - a_i}{p_i} \in \mathcal{BASE}$$

and regardless of the value of k this means that

$$W - x = \beta - m \cdot R \text{ with } m \geq 1$$

and so $W - x \in B$.

We can use $W - x$ in the place of x to show that $x \in \mathcal{B}$ we have $W - x \in \mathcal{G}$.

We know that $|\mathcal{B}| = \left|\left\{x \in \mathcal{B} \;\middle|\; 0 \leq x \leq W\right\}\right|$. Clearly

$$|\mathcal{B}| = \left|\left\{x \in \mathcal{B} \;\middle|\; 0 \leq x < (W+1)/2\right\}\right| + \left|\left\{x \in \mathcal{B} \;\middle|\; (W+1)/2 \leq x \leq W\right\}\right|$$

but since $W - x \in \mathcal{B}$ implies $x \in \mathcal{G}$

$$|\mathcal{B}| = \left|\left\{x \in \mathcal{B} \;\middle|\; 0 \leq x < (W+1)/2\right\}\right| + \left|\left\{x \in \mathcal{G} \;\middle|\; 0 \leq x < (W+1)/2\right\}\right|$$

and so

$$|\mathcal{B}| = (W+1)/2$$

In addition to being able to directly calculate the number of bad integers in a CRRS, using the new pseudorank causes both the good and the bad density in the region $0 \leq x \leq W$ to be exactly $1/2$.

Chapter 3

Finite automata and CRRS

In this chapter we introduce several kinds of finite automata (FA) in order to establish a model of computation in which we can easily distinguish computations which are "easy" from computation which are "difficult". FA provide a rigorous online computational model for CRRS data.

We also introduce a number of FA for several of the CRRS properties which were introduced in the previous chapter.

3.1 Finite automata preliminaries

3.1.1 Strings, alphabets and languages

An alphabet Σ is a finite set of symbols and a string s is a finite list of symbols drawn from some alphabet. The string s has alphabet Σ if $s = \sigma_1\sigma_2 \cdots \sigma_{n-1}\sigma_n$ and $\sigma_i \in \Sigma$ for $1 \leq i \leq n$. In this case the string has length $|s| = n$.

The notation Σ^k can be used to denote the set of all strings with alphabet Σ and length k. Regardless of the alphabet we have $\Sigma^0 = \epsilon$, where ϵ is the string with no symbols, also known as the empty string. Σ^* is used to refer to all possible strings that can be formed by the symbols with alphabet Σ.

Finally, a set of strings is referred to as a language. We have $L \subseteq \Sigma^*$, and the members of L are usually selected as the strings in Σ^* that satisfy some predicate.

3.1.2 Deterministic finite automata

A deterministic finite automaton (DFA) processes a string of symbols and either accepts or rejects the string.

The main components of a DFA are the states and the transition function. A DFA has a finite set of states, Q, and the initial state $q_i \in Q$ and a set of final states $Q_f \subseteq Q$ are also specified.

$|Q|$ is usually the principle parameter in any equations relating to the time and space bounds of DFA operations. Similarly, $|Q|$ will be the problem size in any discussion involving the computation of DFA properties or functions and complexity classes unless stated otherwise.

We use $Q(F)$ to refer to the states of the DFA F whenever Q would be ambiguous. This convention is used for all of the properties of DFA that we define. We use $\Sigma(F)$, for example, to denote the alphabet of the DFA F when it would not otherwise be made clear by the context in which it was used.

The transitions of a DFA are defined by a transition function $\delta(q_a, \sigma) \rightarrow q_b$, which maps each of the combinations of the states $q_a \in Q$ and symbols $\sigma \in \Sigma$ to some state $q_b \in Q$. A DFA in state q_a will transition to state q_b when the DFA processes the symbol σ.

We generalize the transition function to use strings in the second argument such that

$$\delta(q, s) = \delta(\delta(\ldots \delta(\delta(q, \sigma_1), \sigma_2) \ldots, \sigma_{n-1}), \sigma_n) \text{ where } s = \sigma_1 \sigma_2 \cdots \sigma_{n-1} \sigma_n$$

The previous definition of the transition function is equivalent to the new definition with $s \in \Sigma^1$.

We use q_c to track the current state of a DFA, which is the only memory that a DFA has. We start the processing of the input string $s = \sigma_1 \sigma_2 \cdots \sigma_{n-1} \sigma_n$ by setting $q_c = q_i$. The DFA then processes each of the symbols of the input string, updating the current state with $q_{c(new)} = \delta(q_{c(old)}, \sigma_i)$. The input string is accepted if and only if the $q_c \in Q_f$ after the last symbol in the input string has been processed.

DFA are commonly represented as directed graphs. The states are used as the vertices of the graph and for each $q \in Q$ and $\sigma \in \Sigma$ an edge is drawn between the nodes corresponding to q_a and $\delta(q_a, \sigma)$. The edge is typically labelled with σ. With this in mind we refer to a sequence of states that a DFA transitions through as a path through the DFA.

Figure 3.1 is an example of the directed graph representation of a DFA. The initial state is labelled “start” and the states with the double circles are the final states. This particular DFA accepts all binary strings which have “1” as the third last symbol.

A common notational convenience is to allow $\delta(q, \sigma) = \varnothing$ for some combinations of q and σ, which signifies that a DFA in state q will reject all possible inputs if σ is the first of the remaining symbols to be processed. This is equivalent to creating an additional state $q_r \notin Q_f$ with $\delta(q_r, \theta) = q_r$ for all $\theta \in \Sigma$ and then defining $\delta(q, \sigma) = q_r$.

We use $F(X)$ to denote a DFA F that processes a single input variable x and $F\langle y \rangle$ to refer to the result of using F to evaluate the single input string y. $F\langle y \rangle$ can be used as a predicate indicating whether or not y was accepted by F.

The language of F, denoted by $L(F)$, is

$$L(F) = \left\{ s \;\middle|\; s \in \Sigma(F)^* \;\cap\; F\langle s \rangle \right\}$$

A language that is described by a DFA is known as a regular language.

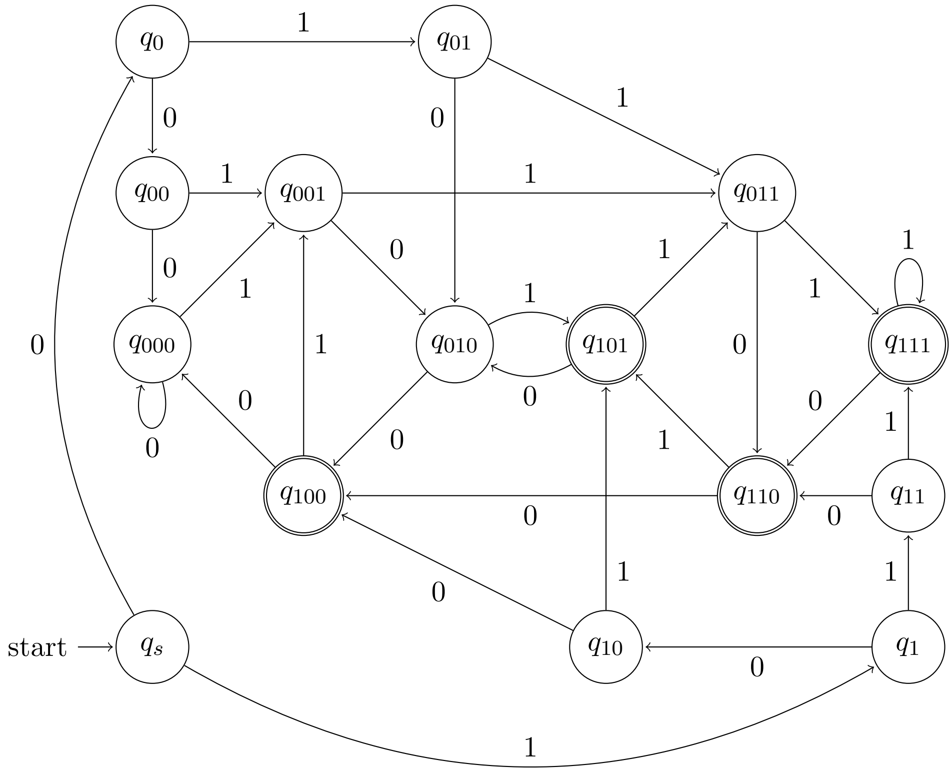


Figure 3.1: An example DFA.

Huffman [36], Mealy [50] and Moore [54] were amongst the first to define and examine DFA.

Equivalence of DFA

Two DFA which accept the same language are said to be equivalent.

The equivalence of states in a DFA is a separate but related concept. We consider the states q_a and q_b to be equivalent if they behave identically when processing identical inputs. This means that $\delta(q_a, s) \in Q_F$ if and only if $\delta(q_b, s) \in Q_F$ for all $s \in \Sigma^*$

Let F_a be a DFA with current state q_a and let F_b be the same DFA with current state q_b. The states q_a and q_b are equivalent if and only if

$$F_a \langle s \rangle = F_b \langle s \rangle \text{ for all } s \in \Sigma^*$$

We test for state equivalence by creating a pairwise list of nonequivalent states NEQ and then checking that list for the pair of states being tested.

We add (q_a, q_b) to NEQ if for all distinct $q_a, q_b \in Q$

$$(q_a \in Q_f \cap q_b \notin Q_f) \, \cup \, (q_a \notin Q_f \cap q_b \in Q_f)$$

This will add pairs of states to NEQ for all DFA except for the DFA with language $L = \varnothing$, which has no final states, and the DFA with language $L = \Sigma^*$, which has no non-final states.

The next step is to check if for any $\sigma \in \Sigma$ we have $(\delta(q_a, \sigma), \delta(q_b, \sigma)) \in NEQ$. If that is the case then (q_a, q_b) is added to NEQ. This process is repeated with all pairs of states not in NEQ until no more pairs are added.

Every pair of states $(q_c, q_d) \notin NEQ$ is equivalent. The equivalence of states is transitive, so if $(q_a, q_b) \notin NEQ$ and $(q_a, q_c) \notin NEQ$ we will have $(q_b, q_c) \notin NEQ$ and $\{q_a, q_b, q_c\}$ will form a set of equivalent states. We define EQ as the set of all state equivalences, so in this example we see that $\{q_a, q_b, q_c\} \in EQ$.

We test for the equivalence between the DFAs F and G by combining the DFAs to form H. The DFA H is created with $Q(H) = Q(F) \cup Q(G)$, $Q_f(H) = Q_f(F) \cup Q_f(G)$. It is clear that if $(q_i(F), q_i(G)) \notin NEQ(H)$ are equivalent then the DFA are equivalent. The text by Hopcroft, Ullman and Motwani contains the formal proof of the equivalence of the DFA [34].

Reachable states

We define the set of reachable states R. We start with $q_i \in R$ and then for all $q \notin R$ we add q to R if $q = \delta(q_r, \sigma)$ for any $q_r \in R$ and $\sigma \in \Sigma$. This process is repeated until no more states are added to R.

A carefully specified DFA will normally have $R = Q$, however unreachable states are commonly produced as the result of various operations on DFA.

Minimization

A DFA F can be minimized, which results in the smallest possible DFA for the language $L(F)$. The first step is to remove all unreachable states, as the states $q \notin R$ can be removed without effecting the behaviour of the DFA.

For every set of equivalent states $Q_E \in E$ we then replace each member of Q_E with a new state. After this step $|E| = 0$, and we have the minimal DFA for the language $L(F)$. Hopcroft, Ullman and Motwani provide a proof that the DFA described is minimal for the language [34].

Watson produced a survey of the many different approaches to DFA minimization and their time and memory requirements [70], including the classic algorithm due to Hopcroft [35].

3.1.3 Non-deterministic finite automata

Non-deterministic finite automata (NFA) are similar to DFA. The main change is that the transition function maps a state and a symbol (or string) to a set of states.

$$\delta(q_a, \sigma) \rightarrow Q_t \text{ where } Q_t = \{q_b, \ldots, q_c\}$$

When the transition function yields a set of states with $|Q_t| \neq 0$ the NFA transitions to each $q \in Q_t$. This adds the requirement that NFA must track the set of current states Q_c instead of a single current state during the evaluation of an input string.

The NFA accepts an input string if, after processing all of the symbols in the input string, at least one of the states in the set of current states is also in the final states. We state this more formally as $Q_c \cup Q_f \neq \varnothing$

There can be multiple paths through a NFA for a given input string and the evaluation of the string will cause the NFA to transition along all of these paths in parallel. When a NFA accepts an input it can give the appearance that the NFA has some prior knowledge of which path to take, which is the reason that NFA are often said to "guess" something about their input.

Relationship to DFA

For every NFA N there is an equivalent DFA D, and so $L(N) = L(D)$. NFA are used in the place of DFA as they allow certain languages to be expressed more concisely.

A NFA can be converted to a DFA by way of the subset construction. This process converts the NFA N with $|Q(N)| = k$ to the DFA D with $|Q(D)| = 2^k$, where the 2^k states of D represent the power set of the states $Q(N)$.

This guarantees that for each set of current states $Q_c(N)$ that occurs during the evaluation of a string with the NFA there will be an analogous current state $q_c(D)$ in the DFA.

In many cases a large proportion of the newly created states will be unreachable and so will be removed, although there are cases where all 2^k states are required. This is the drawback to using NFA where a DFA implementation is required and is also the reason that NFA can be more concise than the equivalent DFA.

Figure 3.2 is the NFA equivalent of Figure 3.1. It is an excellent example of the expressive power of NFA relative to DFA.

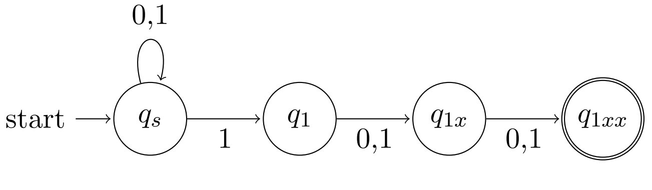


Figure 3.2: An example NFA.

State equivalence in a NFA is not always determinable, and it is no longer true that a unique minimal NFA exists for any given language. Furthermore, Meyer and Stockmeyer showed that finding any minimal NFA for a particular language is PSPACE-complete for the number of states [51].

A NFA can be converted to a DFA and then minimized, but this is not an efficient approach in general because of the risk that the subset construction will create a DFA with an exponential number of states.

Rabin and Scott [59] defined NFA and showed their equivalence to DFA via the subset construction. Their paper also demonstrated a number of properties of FA and the regular languages.

ϵ-NFA

There is an extension to NFA which allow some languages to be described even more concisely. The extension adds spontaneous transitions, known as ϵ-transition, to the NFA. The NFA which include ϵ-transitions are known as ϵ-NFA.

The ϵ-transitions are named after the empty string, ϵ, and occur after an ϵ-NFA transitions to a new set of states but before the ϵ-NFA processes the next symbol. The transitions normally represent additional paths that the ϵ-NFA F could take, and so a transition to the state q_a which has $\delta(q_a, \epsilon) = q_b$ will see the set of current states of F updated to include q_a and q_b. In a similar manner to with regular NFA, we can think of ϵ-NFA as guessing whether or not they should follow an ϵ-transition before processing the next symbol, with the guarantee that the FA will always guess correctly if an accepting path exists.

The ϵ-closure of a state is the set of states that can be reached from the state by following ϵ-transitions. We form the ϵ-closure of the state q, $E(q)$, by setting $E(q) = q$ and then adding $\delta(r, \epsilon)$ to $E(q)$ for each $r \in E(q)$ until there are no more states that can be added.

An ϵ-NFA can be converted to a regular NFA by replacing $\delta(q, \sigma)$ with $E(\delta(q, \sigma))$ for all $q \in Q$ and $\sigma \in \Sigma$.

3.1.4 Matrix notation

We can use the directed graph representation of a FA to derive an equivalent matrix representation.

Set $|Q| = n$ and use $q_{\#j}$ to denote the jth state in Q under some ordering. We define the vectors u and v and the matrices $w(\sigma)$ for each $\sigma \in \Sigma$ such that

- the vector u has size $|u| = \{1, n\}$ and entries $u_j = [q_{\#j} \in E(q_i)]$
- the vector v has size $|v| = \{n, 1\}$ and entries $v_j = [q_{\#j} \in Q_f]$
- each matrix $w(\sigma)$ has size $|w(\sigma)| = \{n, n\}$ and entries $w(\sigma)_{j,k} = [q_{\#k} \in \delta(q_{\#j}, \sigma)]$

For the FA that we have described so far we need to restrict the type of the elements of these vectors and matrices.

We use integers for the elements with each element $e \in \{0, 1\}$, and are similar to Boolean values. This is clear from the definitions of u, v and $w(\sigma)$, which set the values of their elements with $e = [\textit{condition}]$ for some condition. The use of a vector of these elements to indicate

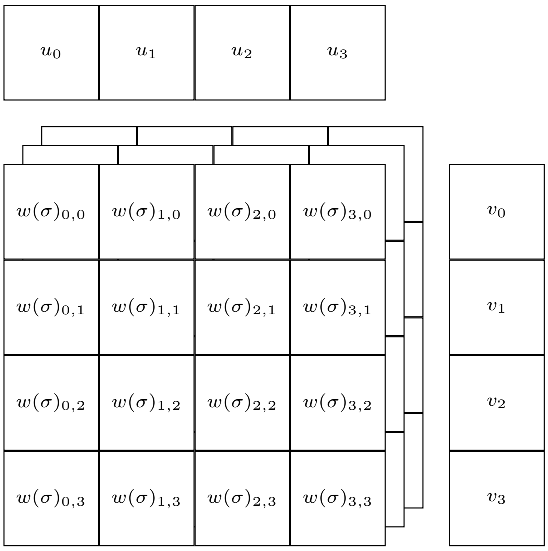


Figure 3.3: Example of FA matrix notation.

membership in the sets of initial, current and final states provides a second demonstration of the integer elements being used with Boolean semantics.

All matrix manipulations see modifications along the same lines. In all arithmetic internal to a matrix operation, we replace addition with the logical OR operation and replace multiplication with the logical AND operation.

To ensure that each state transition in a DFA has only one destination we restrict the matrices $w(\sigma)$ for DFA such that

$$\sum_{k=1}^{n} w(\sigma)_{j,k} \in \{0,1\} \text{ for } 1 \leq j \leq n \tag{3.1}$$

where all arithmetic is carried out with integers rather than Booleans.

This restriction is not necessary for NFA or ϵ-NFA.

It is assumed that the ϵ-transitions of ϵ-NFA have been replaced by the appropriate state transitions either before or during the process of creating the matrix representation of the ϵ-NFA.

The only place where ϵ-transitions are seen in the matrix representation is in the vector u, which represents the ϵ-closure of the initial state of the FA. This results in DFA and regular NFA having exactly 1 non-zero entry in u.

It is clear that the evaluation of an input string $s = \sigma_1\sigma_2 \cdots \sigma_{k-1}\sigma_k$ can be done with the matrix representation of a FA as

$$F\left\langle \sigma_1\sigma_2 \cdots \sigma_{k-1}\sigma_k \right\rangle = u(F) \cdot \left(\prod_{i=1}^{k} w(F, \sigma_i) \right) \cdot v(F) \tag{3.2}$$

If we view the vector $u(F)$ as the set of current states of the FA F before any symbols have been processed, then we will see that

$$Q_c(F, j) = u(F) \cdot \prod_{i=1}^{j} w(F, \sigma_i)$$

is the set of current states of F after j symbols have been processed.

The vector $u(F) \cdot \left(\prod_{i=1}^{k} w(F, \sigma_i) \right) = Q_c(F, k)$ in Equation 3.2 corresponds to the set of current states of F after all of the symbols have been processed. Since $v(F)$ has non-zero entries only for the final states of F, the multiplication by $v(F)$ computes the sum of the states q for which $q \in Q_C(F, k) \cap q \in Q_F$.

3.1.5 Inputs in multiple variables

At this point we have only described finite automata (FA) which process a single input string, however in some cases we would like to process multiple strings. An example would be building a FA that accepts a set of 3 binary strings a, b and c such that $a + b = c$. We would define such a FA as $F(a, b, c)$ and would see that $F \langle 1, 2, 3 \rangle = 1$ and $F \langle 1, 2, 4 \rangle = 0$.

We cannot normally process the multiple strings sequentially. The memory of a FA is the set of current states, so if the FA needs information about each of the strings that cannot be distilled down to a small number of states then processing multiple strings sequentially will not work. If the projection operation, detailed in Section 3.3.4, is used then the output of the operation will not be meaningful if the strings are processed sequentially.

The alternative is to process the strings in parallel, and there are two possible approaches. We will use the example FA F that tests when $a + b = c$ to explain both approaches. We have

$$a = \sum_{i=0}^{n} 2^i \cdot \alpha_i \qquad b = \sum_{i=0}^{n} 2^i \cdot \beta_i \qquad c = \sum_{i=0}^{n} 2^i \cdot \gamma_i$$

with $n = \max(\log a, \log b, \log c)$.

The first approach is to interleave the symbols from each of the strings. A FA that interleaves the symbols of its inputs will process a symbol from each of its inputs in some order. The order in which the example FA will process the symbols will be something like

$$\alpha_0\beta_0\gamma_0\alpha_1\beta_1\gamma_1 \cdots \alpha_{n-1}\beta_{n-1}\gamma_{n-1}\alpha_n\beta_n\gamma_n$$

The alternative approach is to merge the symbols to form a single new string.

For the ith of the n input strings, with $1 \leq i \leq n$, let x_i be the symbol that the FA will process next and X_i be the alphabet associated with the input string. We can map every combination of $x_1 x_2 \cdots x_{n-1} x_n$ to a single value $y \in Y$ where $|Y| = \prod_{i=1}^{n} |X_i|$. This mapping can also be performed in the reverse direction if necessary.

We perform similar operations frequently enough that it is convenient to define a common notation for converting between multiple and single dimensional values.

Let each x_i be an integer in the range $0 \leq x_i < d_i$ for some value d_i. Define

$$D_i = \prod_{j=1}^{i} d_j$$

We use

$$\|x_1 x_2 \cdots x_{n-1} x_n\|_{d_1,d_2,\ldots,d_{n-1},d_n} \Rightarrow y$$

to denote the mapping of the multiple values x_i to the single value

$$y = \sum_{i=1}^{n} D_{i-1} \cdot x_i$$

The reverse mapping

$$y \Rightarrow \|x_1 x_2 \cdots x_{n-1} x_n\|_{d_1,d_2,\ldots,d_{n-1},d_n}$$

sets the single values

$$x_i = |y/D_{i-1}|_{D_i} \tag{3.3}$$

for $1 \leq i \leq n$.

When merging the symbols of multiple inputs a suitable mapping should be used between the symbols $x_i \in X_i$ and the integers $0, \ldots, |X_i| - 1$.

A FA that interleaves the symbols of k strings will use k times as many states as a FA that merges the symbols, although the transition function will be take k times less space to describe.

When FA are represented with matrices we will merge the symbols whenever we need to process multiple input strings. Increasing the number of symbols to be processed by a factor of k will increase the memory usage by a factor k, whereas increasing the number of states by a factor of k will increase the memory usage by a factor of k^2. Additionally, fewer states mean smaller matrices, which generally result in faster FA operations.

This is why we have chosen to merge the symbols of the inputs whenever we have FA that deal with multiple strings.

3.2 Weighted finite automata and finite automata for CRRS

3.2.1 Weighted finite automata

Weighted finite automata (WFA) associate a weighting with the initial state or states and to each transition in a FA.

Each transition between two states in a FA can be uniquely specified by the 3-tuple (q_a, σ, q_b), where $q_b \in \delta(q_a, \sigma)$. We define the weight function $\Omega_i(q)$ to provide the initial weighting for each state and $\Omega_t(q_a, \sigma, q_b)$ to provide a numeric weight for each transition in a FA. Note that $\Omega_i(q) = 0$ if $q \notin Q_i$ and $\Omega_t(q_a, \sigma, q_b) = 0$ if $q_b \notin \delta(q_a, \sigma)$.

For every string $s = \sigma_1\sigma_2 \cdots \sigma_{k-1}\sigma_k$ than a FA processes we are able to describe the sequence or sequences of states that the FA will transition through.

We refer to such a sequence of states as a path, and with $q_x \in Q$ and $0 \leq x \leq k$ the path $q_0, q_1, \ldots, q_{k-1}, q_k$ is described by

$$\begin{aligned} q_0 &= q_i \\ q_j &\in \delta(q_{j-1}, \sigma_j) \end{aligned}$$

For a particular string s we refer to the set of paths that a FA transitions through as path(s) and use $t \in \text{path}(s)$ to denote that the path $t_0, t_1, \ldots, t_{k-1}, t_k$ exists. If $q_k \in Q_f$ we refer to $q_0, q_1, \ldots, q_{k-1}, q_k$ as an accepting path.

Some WFA act as predicates by accepting any string that has an associated weight equal to or greater than some specific weight threshold. The WFA that we use evaluate an input string and return a weight associated with the input string, such that

$$F\langle s\rangle = \sum_{p \,\in\, \text{path}(F,s)} \Omega_i(p_0) \cdot \prod_{i=1}^{k} \Omega(p_{i-1}, s_i, p_i)$$

Since the result of the computation is a numeric value instead of either “accept” or “reject” we are working with an automata variant known as a transducer. In this work the type of the result will always be clear from the context and so we do not need to distinguish between automata and transducers in our explanations.

Since there is at most one path through a DFA for each string, the result for WFA constructed from DFA will either be the product of weights along that path or 0 if the string is rejected.

For the purposes of this work we mostly use of WFA with non-negative integer or rational weights. This is easily achieved with the FA matrix notation by using integers or rationals as the matrix elements and numeric addition and multiplication in the place of the Boolean equivalents that were used for DFA and NFA. We will also briefly discuss the use of indefinite polynomials as weights in Chapter 4.

The only restriction on the weights it that they must belong to a semi-ring. A semi-ring is a set S for which two operations, product ($\times$) and sum ($+$), and two values, 0 and 1, have been defined such that $s \times 1 = 1$, $s + 0 = 0$ and $s \times 0 = 0 \times s = 0$ for all $s \in S$.

The set of integers form a semi-ring, where integer multiplication and addition are used for the product and sum operations. The Boolean semi-ring contains the values false and true, which correspond to the values 0 and 1 in the definition of semi-rings, and use logical AND and OR are used for the product and sum operations respectively.

If the WFA weights are negative or complex numbers and the WFA is derived from a NFA then the weights of several paths through the WFA can potentially cancel the effects of one another. If cancellation is desirable, when working with spectra for instance, then this might be a benefit rather than a drawback, however in other cases some care must be taken to prevent cancellation from causing problems. There are techniques that can be used to avoid cancellation in certain settings, as we shall see when we discuss emptiness in Section 3.3.5.

WFA have been used heavily by Mohir [52], with a focus on speech processing, and by Culik and Kari [17], who used WFA for image compression.

In this work we will not be making use of the full power of WFA. We define the particular restrictions that allow us to avoid some of harder problems associated with WFA in general and integer WFA in particular in Section 3.2.3.

The interested reader is referred to the text by Berstel and Reutenauer [7], which is an update of an older and much cited text [8]. The newer version is freely available from the authors' website and covers the theory of WFA from the initial work by Kleene [40] and Schützenberger [61] through to contemporary results in the field. The other standard text is by Kuich and Salomaa [41], which despite not being as current is still an excellent reference.

Halava and Harju have explored the limitations of integer WFA [29] and the languages accepted by integer WFA and the closure properties of those languages [28]. Their work shows that the restrictions on what can and cannot be computed by integer WFA are much more severe than we see when working with the regular languages.

Instead of treating integer WFA as a model of computation, we choose to view WFA as a method of performing a small amount of arithmetic on a string or set of strings belonging to a regular language. The operations and properties of regular languages are already well understood and are covered in the standard texts on computation [63],[34], and so if we are careful with the size and type of the weights we can use WFA to efficiently uncover additional information about particular regular languages.

Optimization of WFA is a much harder problem than optimization of DFA in the general case. Mohri [53] describes a WFA minimization algorithm that works for WFA defined in the tropical semi-ring. The tropical semi-ring has the union of the natural numbers and infinity as its underlying set. The product operation is integer addition and the sum operation takes the minimum of its arguments, with ∞ and the integer 0 acting as the values of 0 and 1 respectively.

Eisner [23] discusses a more general approach to minimization, describing how to determine which semi-rings allow minimization and how to minimize a WFA in such a semi-ring. Additionally, he shows that finding the minimum numbers of states WFA in any other kind of semi-ring is NP complete and cannot be approximated.

The integer WFA that we will be working with are among the WFA that cannot be minimized. In Section 3.1.3 we define integer WFA with some additional restrictions which allow us to use standard DFA optimization on the restricted WFA.

3.2.2 FA for CRRS

All languages accepted by a FA that operates on CRR strings will have length r and describe sets of integers in the CRRS.

For the input x we would like to use $\langle x\rangle_1 \langle x\rangle_2 \cdots \langle x\rangle_{r-1} \langle x\rangle_r$ as the input string. The symbols are residue values and contain no information about which modulus the residue value is associated with, and so there is no way to distinguish $\langle x\rangle_1$ from $\langle x\rangle_2$ in an input string with $\langle x\rangle_1 = \langle x\rangle_2$.

There are two ways to overcome this ambiguity.

The first method renumbers the symbols to make them unique. A possible renumbering of the symbols is

$$\sigma_{\langle x\rangle_i} = \langle x\rangle_i + \sum_{j=1}^{i-1} p_j$$

A side effect of using this method is that the FA will see an increase in the number of required states. The FA grows so that we can guarantee that the FA will not process any of the CRR moduli out of order.

The states are copied into $r+1$ blocks and the state transitions are modified so that the transition that was from state a to state b will be from the copy of state a in block j to the copy of state b in block $j+1$.

Let $|Q| = n$ and let $\acute{A}$ denote the FA property A after the symbol renumbering. We have

$$|\acute{Q}| = (r+1) \cdot n$$

and, with the transition function modified to use numbers to represent the states under some ordering,

$$\acute{\delta}(k \cdot n + i, \sigma) = \delta(i, \sigma) + (k+1) \cdot n$$

for $0 \leq i < n$ and $0 \leq k \leq r$.

In the ranges $0 \leq j, k < n$ and $0 \leq i \leq r$ the matrix form of the FA becomes

$$\begin{aligned} \acute{u}_j &= u_j \\ \acute{w}(\sigma_{\langle x\rangle_i})_{i \cdot n+j, (i+1) \cdot n+k} &= w(\langle x\rangle_i)_{j,k} \\ \acute{v}_{r \cdot n+j} &= v_j \end{aligned}$$

and elsewhere $\acute{u}_j = \acute{w}_{j,k} = \acute{v}_j = 0$.

Any unreachable states can be removed, and the FA can be optimized if it is based on a DFA.

This method ensures that the FA will only accept strings which represent all r CRR moduli, and that the moduli are processed in some fixed order. The cost is a potential r-fold increase in the number of states required. The large blocks of value 0 in the matrices are a possible indication that we are not getting maximum utility from this increase in states.

The alternative method does not require an increase in the number of states. The FA definition is modified instead, so that there is a transition function for each residue, replacing $\delta(q_a, \langle x\rangle_i) = q_b$ with $\delta(q_a, i, \langle x\rangle_i) = q_b$.

The partitioning of the transition function by moduli causes only a minor change to the matrix representation of FA, in which the w matrices are stored in r separate groups. This simplifies the changes to the transition function for most of the FA operations we will use. An implementer who wants accurate results will need to make sure that the order of the symbols in the input string is correct, as there is nothing in this method which will force this to be true.

The evaluation of the FA in matrix form becomes

$$F\left\langle \sigma_1\sigma_2\cdots,\sigma_{r-1}\sigma_r\right\rangle = u(F)\cdot\left(\prod_{i=1}^{r} w(F, i, \sigma_{\langle x\rangle_i})\right)\cdot v(F) \tag{3.4}$$

The two method are equivalent in that they both result in FA with the same behaviour, however we use the second method exclusively since it leads to a more efficient implementation.

Note that $F\langle x\rangle$ is a shorthand for $F\left\langle |x|_P\right\rangle$, since any value of x outside of the range $0 \leq x < P$ will be brought back into the range by virtue of the representational capacity of the CRRS.

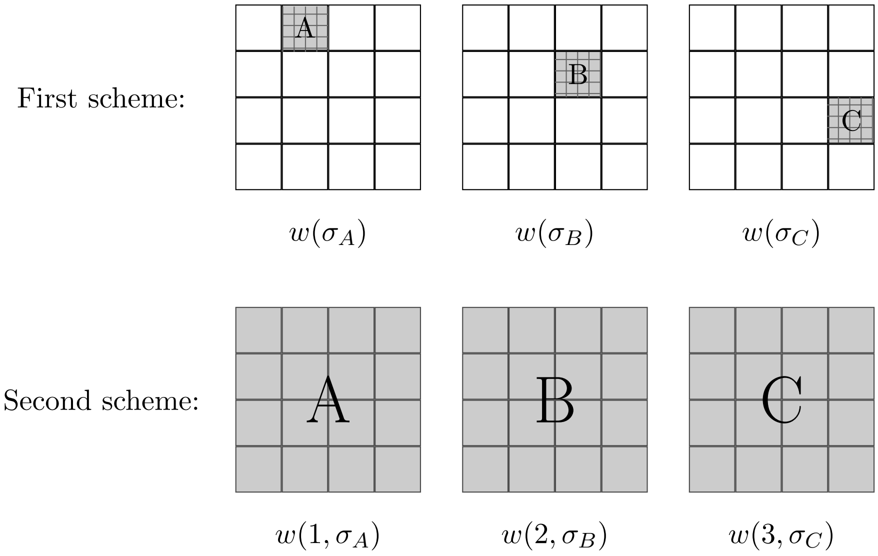


Figure 3.4: A comparison of the two schemes for CRRS FA.

3.2.3 Counting finite automata

We define one more variant of finite automata, which we will refer to as a counting finite automata (CFA). These automata are used exclusively with CRR, and so everything in Section 3.2.2 applies to CFA.

A CFA is simply a DFA or NFA in matrix form where all non-zero entries are the integer 1 and integer addition and multiplication are used for matrix operations. The output of a CFA is the number of accepting paths that an input can take through the FA. We define CFA only for languages with $L \subseteq \Sigma^k$ for some fixed k.

If the input is rejected the output will be 0, and if the input is accepted and the CFA is based on a DFA the output will be 1. If the input is accepted and the CFA is based on a NFA then the results will be $|Q_c \cap Q_f|$ where Q_c is the set of current states after the input string has been processed.

For CFA built from DFA we have FA that we can easily minimize that retain some of the advantages of WFA.

CFA are more useful when used in conjunction with the logical product operation, covered in Section 3.3.4. The logical product of the CFA F we can determine $|L(F)|$, the computation of which is simplified by the restriction $L \subseteq \Sigma^k$.

If the DFA F computes some CRR predicate then the logical product of the corresponding CFA computes $\sum_{x=0}^{P} [F \langle x \rangle]$, and this is a quantity which is of interest to us.

Computing the memory requirements of a CFA in advance is easier than with a general WFA, as we have a precise upper bound of n bits of memory for each entry in the matrices and no possibility of cancellation.

3.3 Operations and properties

In this section we define several FA operations, and discuss their running time and memory usage. We assume that all FA operations are working with FA in matrix form. Although we are primarily concerned with CFA for CRR problems we will discuss any interesting alternatives, especially when they contrast with the CRR CFA case.

If we have an k-state FA in matrix form where each entry in the matrices requires $O(m)$ bits of storage we need

$$\left(k^2 \cdot |\Sigma| + 2 \cdot k\right) \cdot O(m)$$

bits of storage for the matrices.

If the FA is based on a DFA then Equation 3.1 holds and we can compress the w matrices, since each row has at most one non-zero value in it. For each row of a matrix we replace the k $O(m)$-bit values in the row with a single $O(m)$-bit value and a $O(\log k)$-bit index which indicates which column of the row the value occurs in.

From this it is clear that

$$k \cdot \log k \cdot |\Sigma| + (k \cdot |\Sigma| + 2 \cdot k) \cdot O(m)$$

bits of storage are required to store the matrices used by such a FA.

For CRR CFA we have $|\Sigma| = \sum_{i=1}^{r} p_i$ and $O(m) = 1$, so the memory requirements are easy to determine given the number of states of the FA.

When we discuss the memory requirements of an operation in the following sections we are referring to the amount of working space required to carry out the operation.

3.3.1 Evaluation

FA evaluation has been defined and discussed earlier in this work. In this section we discuss the time and memory required to evaluate an input string for different kinds of FA.

For a k-state FA to evaluate and input string we require 2 k-element vectors, which are used to store the old and new sets of current states respectively. If we use c_0 and c_1 to denote the current state vectors, the evaluation of the string $s = \sigma_1\sigma_2 \cdots \sigma_{j-1}\sigma_j$ is performed as follows

$$\begin{aligned} c_0 &= u \\ c_{|i|_2} &= c_{(1-|i|_2)} \cdot w(\sigma_i) \\ F\langle x \rangle &= c_{|j|_2} \cdot v \end{aligned}$$

It it is clear that evaluation requires j vector-matrix multiplications and 1 vector-vector multiplication. If the FA is only going to be evaluated once we could use u as the c_0 vector and save some memory. When this is not the case we will also need to assign u to c_0, which will extend the upper bound of the running time by $O(km)$, where m is the upper bound on the number of bits used to store the entries of the state vector.

The implementation of the vector-matrix multiplication is dependant on whether the FA is based on a DFA or a NFA. The DFA version requires at most k multiplications and additions where the NFA version requires at most k^2 multiplications and additions. The vector-vector multiplication requires at most k multiplications and additions regardless of the type of FA.

Actual DFA and NFA have 1-bit matrix entries that never grow, which means the current state vectors require $2k$ bits of memory and that the arithmetic performed on the matrix entries will take $O(1)$ time. It is clear that a DFA will process a j symbols input string in $O(jk)$ time where a NFA will process the same symbols in $O(jk^2)$ time.

We can derive the time and memory bounds for CFA and WFA from the bounds for DFA and NFA, as the only difference will come from the size of the entries.

If the CFA or WFA current state vectors have entries which are at most h-bits in size then the working memory required will be $O(kh)$ bits. This bound hides an additional $O(h)$-bits of working memory for the multiplications of the matrix elements.

The running times will be

$$O(k \cdot (h + h\log h \log\log h) \cdot (j+1)) = O(j \cdot k \cdot h \log h \log\log h)$$

for DFA based FA and

$$O(k \cdot (h + h\log h \log\log h) \cdot (jk+1)) = O(j \cdot k^2 \cdot h \log h \log\log h)$$

for NFA based FA.

All that remains is to determine the maximum value of h. This is simple for CRR based CFA, which have $h = \log P = n$.

For WFA we can only compute h if we have upper bounds on both the size of the matrix entries and the length of the input strings. If each matrix entry of a WFA is at most m bits in length and the input string contains j symbols or less then $h \leq j \cdot (m + k) + m$.

3.3.2 Union

The union of two FA results in a new FA that accepts any string if either of the original FA accept the string. We use the notation $F \cup G = H$ to denote the union of the FA F and G resulting in the FA H.

The resulting FA H has

$$L(H) = \left\{ x \in \Sigma^* \;\middle|\; x \in L(F) \cup x \in L(G) \right\}$$

Note that

$$H \langle x \rangle = F \langle x \rangle + G \langle x \rangle$$

where the addition is performed with Boolean arithmetic for DFA and NFA and with numeric arithmetic for WFA and CFA.

The result of the union operation will be larger than the union operands, with $|Q(H)| = |Q(F)| + |Q(G)|$.

The matrix representation of H can be derived from the matrix representation of F and G. The initial state vector u is defined as

$$\begin{aligned} u(H)_i &= u(F)_i \text{ for } 0 \leq i < |Q(F)| \\ u(H)_{|Q(F)|+i} &= u(G)_i \text{ for } 0 \leq i < |Q(G)| \end{aligned}$$

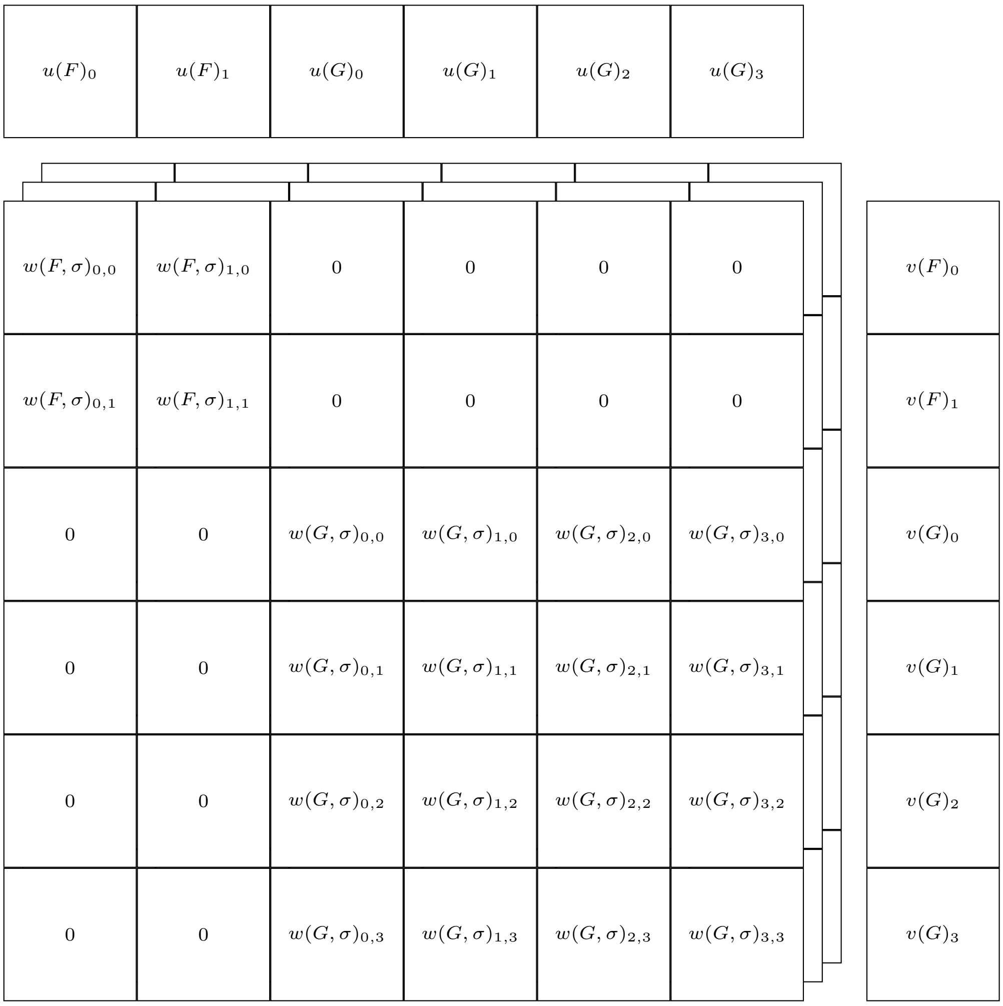
and the final state vector v is created similarly as

$$\begin{aligned} v(H)_i &= v(F)_i \text{ for } 0 \leq i < |Q(F)| \\ v(H)_{|Q(F)|+i} &= v(G)_i \text{ for } 0 \leq i < |Q(G)| \end{aligned}$$

The transition matrices have all entries set to 0 except for the entries

$$\begin{aligned} w(H,\sigma)_{i,j} &= w(F,\sigma)_{i,j} \text{ for } 0 \leq i,j < |Q(F)| \\ w(H,\sigma)_{|Q(F)|+i,|Q(F)|+j} &= w(G,\sigma)_{i,j} \text{ for } 0 \leq i,j < |Q(G)| \end{aligned}$$

The effect on the matrix representation can be seen most clearly in Figure 3.5.



$u(F)_0$	$u(F)_1$	$u(G)_0$	$u(G)_1$	$u(G)_2$	$u(G)_3$

$w(F,\sigma)_{0,0}$	$w(F,\sigma)_{1,0}$	0	0	0	0
$w(F,\sigma)_{0,1}$	$w(F,\sigma)_{1,1}$	0	0	0	0
0	0	$w(G,\sigma)_{0,0}$	$w(G,\sigma)_{1,0}$	$w(G,\sigma)_{2,0}$	$w(G,\sigma)_{3,0}$
0	0	$w(G,\sigma)_{0,1}$	$w(G,\sigma)_{1,1}$	$w(G,\sigma)_{2,1}$	$w(G,\sigma)_{3,1}$
0	0	$w(G,\sigma)_{0,2}$	$w(G,\sigma)_{1,2}$	$w(G,\sigma)_{2,2}$	$w(G,\sigma)_{3,2}$
0	0	$w(G,\sigma)_{0,3}$	$w(G,\sigma)_{1,3}$	$w(G,\sigma)_{2,3}$	$w(G,\sigma)_{3,3}$

$v(F)_0$
$v(F)_1$
$v(G)_0$
$v(G)_1$
$v(G)_2$
$v(G)_3$

Figure 3.5: Matrices for $F \cup G$, with $|Q(F)| = 2$ and $|Q(G)| = 4$.

3.3.3 Intersection

The intersection of two FA results in a new FA that accepts any string if both of the original FA accept the string. We use the notation $F \cap G = H$ to denote the intersection of the FA F and G resulting in the FA H.

The resulting FA H has

$$L(H) = \Big\{ x \in \Sigma^* \;\Big|\; x \in L(F) \cap x \in L(G) \Big\}$$

Note that

$$H \langle x \rangle = F \langle x \rangle \cdot G \langle x \rangle$$

where the multiplication is performed with Boolean arithmetic for DFA and NFA and with numeric arithmetic for WFA and CFA.

The result of the intersection operation will be larger than the intersection operands, with

$|Q(H)| = |Q(F)| \cdot |Q(G)|$. An increase in the number of states will increase the running time and memory requirements of most FA operations, and so we normally try to find alternatives to intersection operations when possible.

The matrix representation of H can be derived from the matrix representation of F and G. The initial state vector u is defined as

$$u(H)_{i\cdot|Q(G)|+j} = u(F)_i \cdot u(G)_j \text{ for } 0 \leq i < |Q(F)| \text{ and } 0 \leq j < |Q(G)|$$

and the final state vector v is created similarly as

$$v(H)_{i\cdot|Q(G)|+j} = v(F)_i \cdot v(G)_j \text{ for } 0 \leq i < |Q(F)| \text{ and } 0 \leq j < |Q(G)|$$

The transition matrices are created with

$$w(H,\sigma)_{i\cdot|Q(G)|+k,j\cdot|Q(G)|+l} = w(F,\sigma)_{i,j} \cdot w(G,\sigma)_{k,l} \text{ for } 0 \leq i,j < |Q(F)| \text{ and} 0 \leq k,l < |Q(G)|$$

The effect on the matrix representation can be seen most clearly in Figure 3.6.

3.3.4 Projection and the logical product

We can create a new FA by projecting one or more of the input variables of a FA. We use $\pi_x F(x,y) = G(y)$ to denote the projection of the variable x in the FA F, which results in the FA G.

The operation makes x an existentially quantified variable of G. If $[G\langle y\rangle] = 1$ then there exists an $x \in \Sigma_x^*$ for which $[F\langle x,y\rangle] = 1$.

We first describe the case where one or more of the input variables remain after other variables have been projected.

The projection operation only effects the transition function of the FA. The transitions for the symbol

$$\sigma_I = \|\sigma_e, \sigma_f\|_{|\Sigma_a|,|\Sigma_b|}$$

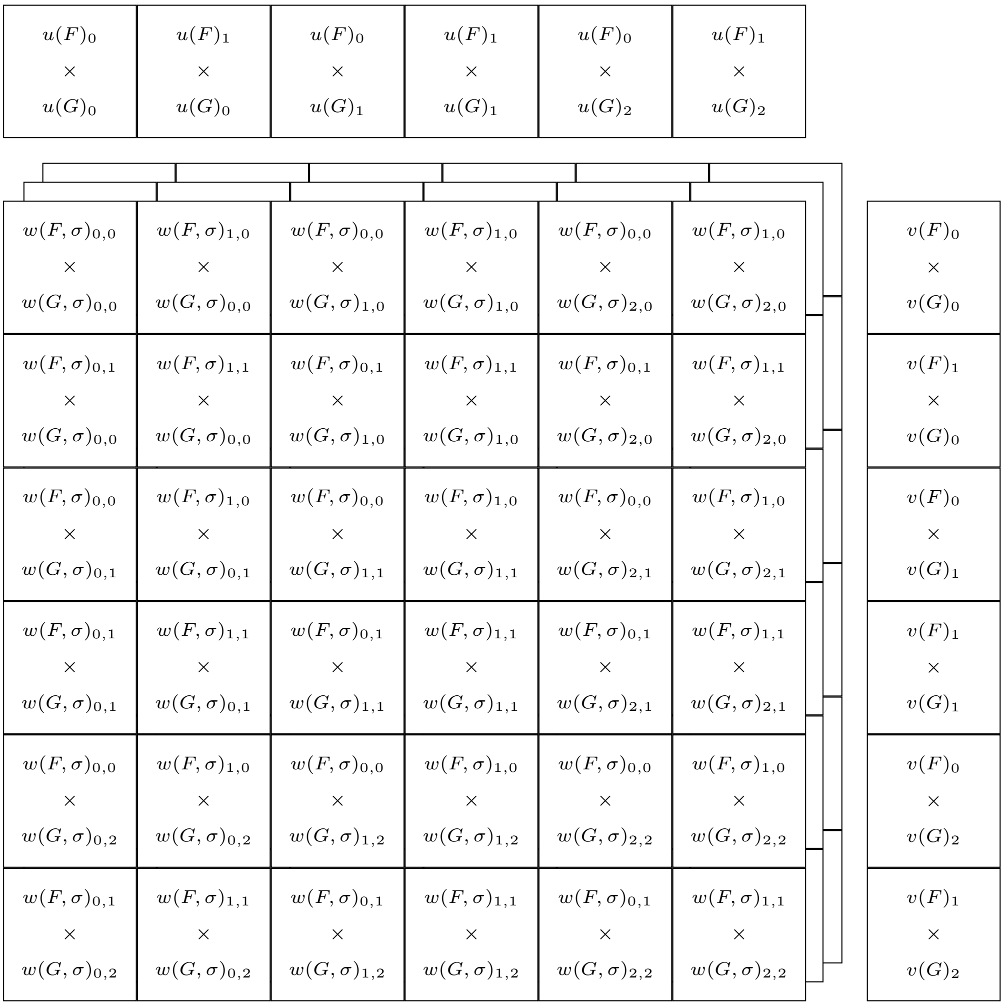
in the FA I will be the combination of the transitions for every symbol

$$\sigma_H = \|\sigma_a, \sigma_b, \sigma_c, \sigma_d\|_{|\Sigma_a|,|\Sigma_b|,|\Sigma_c|,|\Sigma_d|}$$

in the FA H for which $(\sigma_a = \sigma_e) \wedge (\sigma_b = \sigma_f)$.

For the FA that were used to demonstrate the notation for projection,

$$\delta(G,q,\sigma_y) = \bigcup_{x\in\Sigma_x} \delta(F,q,\sigma_{\|x,y\|_{|\Sigma_x|,|\Sigma_y|}})$$



$u(F)_0 \times u(G)_0$	$u(F)_1 \times u(G)_0$	$u(F)_0 \times u(G)_1$	$u(F)_1 \times u(G)_1$	$u(F)_0 \times u(G)_2$	$u(F)_1 \times u(G)_2$

$w(F,\sigma)_{0,0} \times w(G,\sigma)_{0,0}$	$w(F,\sigma)_{1,0} \times w(G,\sigma)_{0,0}$	$w(F,\sigma)_{0,0} \times w(G,\sigma)_{1,0}$	$w(F,\sigma)_{1,0} \times w(G,\sigma)_{1,0}$	$w(F,\sigma)_{0,0} \times w(G,\sigma)_{2,0}$	$w(F,\sigma)_{1,0} \times w(G,\sigma)_{2,0}$	$v(F)_0 \times v(G)_0$
$w(F,\sigma)_{0,1} \times w(G,\sigma)_{0,0}$	$w(F,\sigma)_{1,1} \times w(G,\sigma)_{0,0}$	$w(F,\sigma)_{0,1} \times w(G,\sigma)_{1,0}$	$w(F,\sigma)_{1,1} \times w(G,\sigma)_{1,0}$	$w(F,\sigma)_{0,1} \times w(G,\sigma)_{2,0}$	$w(F,\sigma)_{1,1} \times w(G,\sigma)_{2,0}$	$v(F)_1 \times v(G)_0$
$w(F,\sigma)_{0,0} \times w(G,\sigma)_{0,1}$	$w(F,\sigma)_{1,0} \times w(G,\sigma)_{0,1}$	$w(F,\sigma)_{0,0} \times w(G,\sigma)_{1,1}$	$w(F,\sigma)_{1,0} \times w(G,\sigma)_{1,1}$	$w(F,\sigma)_{0,0} \times w(G,\sigma)_{2,1}$	$w(F,\sigma)_{1,0} \times w(G,\sigma)_{2,1}$	$v(F)_0 \times v(G)_1$
$w(F,\sigma)_{0,1} \times w(G,\sigma)_{0,1}$	$w(F,\sigma)_{1,1} \times w(G,\sigma)_{0,1}$	$w(F,\sigma)_{0,1} \times w(G,\sigma)_{1,1}$	$w(F,\sigma)_{1,1} \times w(G,\sigma)_{1,1}$	$w(F,\sigma)_{0,1} \times w(G,\sigma)_{2,1}$	$w(F,\sigma)_{1,1} \times w(G,\sigma)_{2,1}$	$v(F)_1 \times v(G)_1$
$w(F,\sigma)_{0,0} \times w(G,\sigma)_{0,2}$	$w(F,\sigma)_{1,0} \times w(G,\sigma)_{0,2}$	$w(F,\sigma)_{0,0} \times w(G,\sigma)_{1,2}$	$w(F,\sigma)_{1,0} \times w(G,\sigma)_{1,2}$	$w(F,\sigma)_{0,0} \times w(G,\sigma)_{2,2}$	$w(F,\sigma)_{1,0} \times w(G,\sigma)_{2,2}$	$v(F)_0 \times v(G)_2$
$w(F,\sigma)_{0,1} \times w(G,\sigma)_{0,2}$	$w(F,\sigma)_{1,1} \times w(G,\sigma)_{0,2}$	$w(F,\sigma)_{0,1} \times w(G,\sigma)_{1,2}$	$w(F,\sigma)_{1,1} \times w(G,\sigma)_{1,2}$	$w(F,\sigma)_{0,1} \times w(G,\sigma)_{2,2}$	$w(F,\sigma)_{1,1} \times w(G,\sigma)_{2,2}$	$v(F)_1 \times v(G)_2$

Figure 3.6: Matrices for $F \cap G$, with $|Q(F)| = 2$ and $|Q(G)| = 3$.

In matrix form the change is

$$w(G, \sigma_y) = \sum_{x \in \Sigma_x} w(F, \sigma_{\|x,y\|_{\Sigma_x, \Sigma_y}})$$

where Boolean arithmetic is used for DFA and NFA, and numeric arithmetic is used for WFA and CFA.

This is slightly different in the CRR form because we partition the symbols by the moduli with which they are associated. We have

$$w(G, i, \sigma_{\langle y \rangle_i}) = \sum_{j=0}^{p_i - 1} w(F, i, \sigma_{\|j, \langle y \rangle_i\|_{p_i, p_i}})$$

where the arithmetic used is appropriate to the type of FA.

The memory used to store the transition matrices will be reduced for DFA and NFA since there will be fewer transition matrices and the size of the matrix elements will be unchanged. For CFA or WFA the storage space required will remain the same as the same amount of information will be stored, albeit in fewer matrices.

We use $\pi_* F(x, y, \dots)$ to denote the logical product of F, which is created by projecting all of the variables of F. Since the projection of all the variables prevents the FA from processing an input string, the result is related to the set of all paths through the FA. This is clear both from the definition of projection and from the similarity between projection and existential quantifiers.

To work with $\pi_* F(x)$ in matrix representation, we compute

$$W(F) = \sum_{x \in \Sigma} w(F, \sigma) \tag{3.5}$$

and then use W to gather some additional information about how the FA behaves.

For CRR FA this becomes

$$W(F, i) = \sum_{j=0}^{p_i} w(F, i, j)$$

What $\pi_* F(x, y, \dots)$ evaluates to and how we use W fall under the scope of the next section.

3.3.5 Counting and emptiness

We refer to the problem of determining the cardinality of a language, $|L|$, as the counting problem. If $|L| = 0$ then the language L is said to be empty. It is obvious that determining the cardinality of a language also determines whether the language is empty.

Emptiness is relatively easy to determine. Equation 3.5 described $W(F)$, which describes all of the state transitions that occur in the FA F.

We use $W(F)$ to check if F accepts any strings with length t, with

$$\left[s \in L(F) \cap s \in \Sigma^t\right] = \left[u(F) \cdot W(F)^t \cdot v(F) = 0\right]$$

If $|Q(F)| = k$ then the maximum distance between any reachable states will be k, which means that if the FA reject every string with length less than or equal to k then $L(F)$ is empty.

We can test for the emptiness of $L(F)$ by checking if

$$[|L(F)| = 0] = \left[u(F) \cdot \left(\sum_{t=0}^{k} W(F)^t\right) \cdot v(F) = 0\right]$$

for standard FA, and

$$[|L(F)| = 0] = \left[u(F) \cdot \left(\prod_{i=1}^{r} W(F, i)\right) \cdot v(F) = 0\right]$$

for FA modified for use with CRRS.

Emptiness is still determinable if we allow the weights of a WFA to be negative. The trick is to test if $|L(F \cap F)| = 0$. The intersection of F and F will be F, however performing the intersection on FA in matrix form will square all of the coefficients and remove the possibility that any of the weights will cause cancellation. A similar trick can be used when the weights are complex numbers.

The only generic methods that we have for determining the cardinality of FA work when $L(F) \subseteq \Sigma^t$ for some constant t. We refer to such an $L(F)$ as a fixed-length language.

We determine $|L(F)|$ for DFA and NFA by converting them to CFA, which use numeric entries and arithmetic in the place of Boolean entries and arithmetic but are otherwise identical. The entire purpose of CFA is to determine $|L(F)|$, and this is what the logical product provides, so

$$\pi_* F(x, y, \dots) = |L(F)| = u(F) \cdot W(F)^t \cdot v(F) \tag{3.6}$$

when $L(F) \subseteq \Sigma^t$.

We normally deal with FA for CRRS, so this becomes

$$\pi_* F(x, y, \dots) = |L(F)| = u(F) \cdot \left(\prod_{i=1}^{r} W(F, i) \right) \cdot v(F) \tag{3.7}$$

Equation 3.7 and Equation 3.5 can be combined in two different ways. We use $\pi_* F(x, y, \dots)_{\Sigma,\Pi}$ if the summation of the transition matrices is done before the multiplication by the vector of current states, and use $\pi_* F(x, y, \dots)_{\Pi,\Sigma}$ if we multiply the transition matrix for each symbol by the old vector of current states and accumulate the result into the new vector of current states.

The computation of $\pi_* F(x)_{\Sigma,\Pi}$ can be described by the recurrence

$$\begin{aligned} c_0 &= u(F) \\ c_i &= c_{i-1} \cdot \sum_{j=0}^{p_i - 1} w(F, i, j) \\ \pi_* F(x)_{\Sigma,\Pi} &= c_r \cdot v(F) \end{aligned}$$

and the computation of $\pi_* F(x)_{\Pi,\Sigma}$ can be described by

$$\begin{aligned} c_0 &= u(F) \\ c_i &= \sum_{j=0}^{p_i - 1} c_{i-1} \cdot w(F, i, j) \\ \pi_* F(x)_{\Pi,\Sigma} &= c_r \cdot v(F) \end{aligned}$$

Aside from the usual working memory requirements for evaluating FA, the evaluation of $\pi_* F(x)_{\Pi,\Sigma}$ does not require any working memory. The evaluation of $\pi_* F(x)_{\Sigma,\Pi}$ will require working memory to store the result of the matrix sum, adding a further $O(k^2 \cdot m)$ bits to the requirements for NFA and $O(k \cdot m)$ bits for DFA.

The bulk of the time for either method will be spent updating the vectors of current states.

For $\pi_* F(x)_{\Sigma,\Pi}$ the accumulation of the matrix values takes time $\sum_{i=1}^{r} p_i \cdot k^2 \cdot O(m)$ and the multiplication of the accumulation matrix by the vector of current states takes time $r \cdot k^2 \cdot O(m + m \log m \log\log m)$.

For $\pi_* F(x)_{\Pi,\Sigma}$ the accumulation of the multiplications into the new vector of current states takes time $\sum_{i=1}^{r} p_i \cdot k \cdot O(m)$. The multiplications of the old vector of current states and the transition matrices takes time $\sum_{i=1}^{r} p_i \cdot k^2 \cdot O(m + m \log m \log\log m)$ for NFA and time $\sum_{i=1}^{r} p_i \cdot k \cdot O(m + m \log m \log\log m)$ for DFA.

The method that is used will depend on whether the FA is based on a DFA or a NFA, the way the matrices are stored, and the memory requirements of the problem at hand.

3.3.6 Complement

The complement of a FA results in a new FA that accepts all strings that are rejected by the original FA and rejects all strings that are accepted by the original FA. We use the notation $\overline{F(x)} = G(x)$ to denote the complement of the FA F resulting in the FA G.

The resulting FA G has
$$L(G) = \left\{ x \in \Sigma^* \;\middle|\; x \notin L(F) \right\}$$

For DFA or NFA the complement is created by inverting the set of final states.

$$Q_f(G) = \left\{ q \in Q(F) \;\middle|\; q \notin Q_f(F) \right\}$$

The complement of a WFA makes little sense, however the complement of a CFA can be useful if used with care.

If $G(x) = \overline{F(x)}$ then
$$\pi_* G \langle x \rangle = |L(F)| - \pi_* F \langle x \rangle$$

Care is needed when F is constructed from other FA, as it is easy to mistakenly compute $\overline{H(x) \cap I(x)}$ when $H(x) \cap \overline{I(x)}$ or $\overline{H(x)} \cap I(x)$ was the intended result.

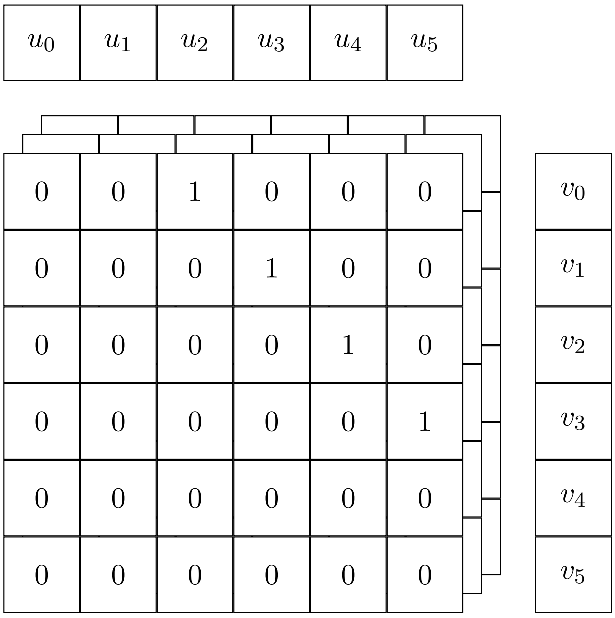
3.4 FA for various CRRS properties

3.4.1 Using FA states for unary addition

When creating FA for CRRS properties we often use the states of the FA to form a unary adder. In some cases the FA performs addition modulo the number of states, and in some cases the FA performs addition and rejects any string that causes the result to be greater than or equal to the number of states. We will sometimes refer to a unary adder as a regular unary adder to distinguish it from a modulo unary adder.

We describe a k-state FA $F(x)$, which is used as a unary adder which rejects results greater than or equal to k. Set $u_d = 1$ for some $0 \leq d < k$ and $u_l = 0$ for $l \neq d$. If the ith residue of x should add $e_i(x)$ to the value of the adder, we set $w(i, \sigma_{\langle x \rangle_i})_{s,t} = [t - s = e_i(x)]$.

In other words, the matrices $w(i, \sigma_{\langle x \rangle_i})$ have non-zero entries only along the $e_i(x)$th upper diagonal.



$$\begin{array}{cccccc|c}
u_0 & u_1 & u_2 & u_3 & u_4 & u_5 & \\
0 & 0 & 1 & 0 & 0 & 0 & v_0 \\
0 & 0 & 0 & 1 & 0 & 0 & v_1 \\
0 & 0 & 0 & 0 & 1 & 0 & v_2 \\
0 & 0 & 0 & 0 & 0 & 1 & v_3 \\
0 & 0 & 0 & 0 & 0 & 0 & v_4 \\
0 & 0 & 0 & 0 & 0 & 0 & v_5
\end{array}$$

Figure 3.7: Transition matrix for a unary adder.

If F is a DFA, then the current state after the input string has been processed will be $c = d + \sum_{i=1}^{r} e_i(x)$.

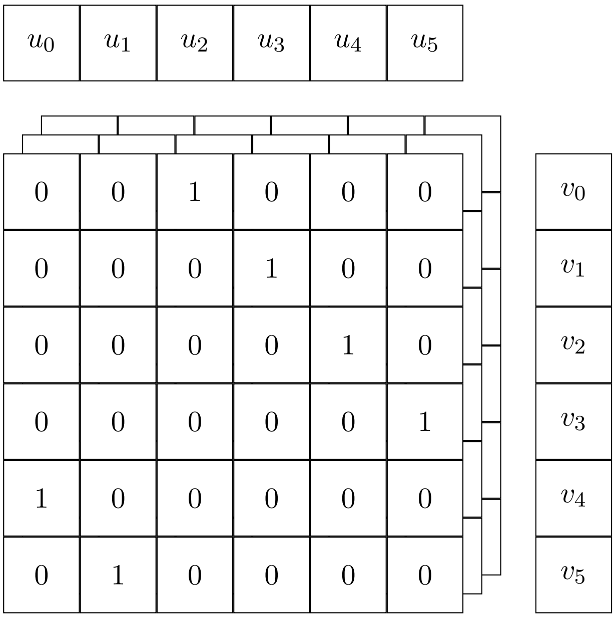
The CRRS FA that we define in this section are all described as DFA. They are often converted to CFA in order to count various sets, and will become NFA if any of the inputs are projected.

A k-state FA which acts as a unary adder modulo k can be created in a similar manner. The transition matrices will be

$$w(i, \sigma_{\langle x \rangle_i})_{s,t} = [t - s \equiv e_i \mod k]$$

and the result will be $c = \left| d + \sum_{i=1}^{r} e_i(x) \right|_k$.

This change means that the matrices $w(i, \sigma_{\langle x \rangle_i})$ now have non-zero entries along the $e_i(x)$th upper diagonal and the $k - e_i(x)$th lower diagonal.



$$\begin{array}{cccccc|c} u_0 & u_1 & u_2 & u_3 & u_4 & u_5 & \\ \hline 0 & 0 & 1 & 0 & 0 & 0 & v_0 \\ 0 & 0 & 0 & 1 & 0 & 0 & v_1 \\ 0 & 0 & 0 & 0 & 1 & 0 & v_2 \\ 0 & 0 & 0 & 0 & 0 & 1 & v_3 \\ 1 & 0 & 0 & 0 & 0 & 0 & v_4 \\ 0 & 1 & 0 & 0 & 0 & 0 & v_5 \end{array}$$

Figure 3.8: Transition matrix for a modulo unary adder.

3.4.2 Pseudorank

The pseudorank of x is

$$\tilde{q}\,(x) = \left\lfloor \sum_{i=1}^{r} \mu_i(x)/S \right\rfloor$$

Since $0 \leq \mu_i(x) < S$ we have $0 \leq \sum_{i=1}^{r} \mu_i(x) < r \cdot S$. We use this fact to create a DFA that can be used for pseudorank calculations.

We describe an $r \cdot S$-state FA $F(x)$ such that $[F\,\langle x \rangle] = [\tilde{q}\,(x) = t]$ for some $0 \leq t < r - 1$.

The FA is a unary adder with $d = 0$ and $e_i = \mu_i(x)$, which corresponds to the matrix descriptions

$$u_i = [i = 0] \qquad\qquad w(i, \sigma_{\langle x \rangle_i})_{s,t} = [t - s = \mu_i(x)]$$

respectively.

After the input string has been processed the current state vector will have a single non-zero entry at position $\sum_{i=1}^{r} \mu_i(x) = \alpha(x)$. We then set $v_i = [t \cdot S \leq i < (t+1) \cdot S]$ to isolate the x with $\tilde{q}\,(x) = t$.

3.4.3 Pseudoparity

The pseudoparity of x is

$$|x|_{\tilde{2}} = \left| \tilde{q}\,(x) + \sum_{i=1}^{r} \langle x \rangle_i \right|_2$$

We can determine whether the pseudorank is even or odd by summing the values of $\mu_i(x)$ modulo $2 \cdot S$. If the adder that calculates this is modified to take into account the parity each $\langle x \rangle_i$ then we have an adder which will let us compute the pseudoparity of x.

We describe an $2 \cdot S$-state FA $F(x)$ such that $[F\langle x \rangle] = [|x|_{\tilde{2}} = t]$ for $t \in \{0, 1\}$.

The FA is a unary adder modulo $2 \cdot S$ with $d = 0$ and $e_i = \mu_i(x) + S \cdot |\langle x \rangle_i|_2$, which corresponds to the matrix descriptions

$$u_i = [i = 0] \qquad\qquad w(i, \sigma_{\langle x \rangle_i})_{s,t} = \big[t - s \equiv \mu_i(x) + S \cdot |\langle x \rangle_i|_2 \mod 2 \cdot S\big]$$

After the input string has been processed the current state vector will have a single non-zero entry c. If $|x|_{\tilde{2}} = 0$ then $0 \leq c < S$ and if $|x|_{\tilde{2}} = 1$ then $S \leq c < 2 \cdot S$.

We set $v_i = [\lfloor i/S \rfloor = t]$ to isolate the x with $|x|_{\tilde{2}} = t$.

3.4.4 GB and BASE

There are several methods we can use to compute if $x \in GB(a, b \cdot R)$ or $x \in \mathcal{BASE}$. The two problems are closely related, as can be seen in Equation 2.23 and Equation 2.22.

We list the different methods in chronological order of discovery, partly to show how our understanding of CRRS' and matrix-based WFA changed over time and partly to demonstrate the performance gains that can be achieved by continually re-examining how a FA operates.

Recall that determining if $x \in GB(a, b \cdot R)$ is equivalent to determining if

$$||x - a|_P|_{\tilde{2}} \neq ||x - L - a - b \cdot R|_{\tilde{2}} + |L + b \cdot R|_P|_2$$

where $L = \lfloor (r - 1) \cdot R \rfloor$

Let $H(x, t)$ be the FA that determines if $|x|_{\tilde{2}} = t$, let $G(x, a, b)$ the FA that determines if $x \in GB(a, b \cdot R)$ and let $F(x)$ be the FA that determines if $x \in \mathcal{BASE}$.

The first versions

The first version of $G(x, a, b)$ can be constructed directly from Equation 2.15

If $||L + b \cdot R|_P|_2 = 0$ then

$$\begin{aligned} G(x, a, b) = \Big(&H(x - a, 0) \;\cap\; H(x - L - a - b \cdot R, 1)\Big) \;\cup \\ &\Big(H(x - a, 1) \;\cap\; H(x - L - a - b \cdot R, 0)\Big) \end{aligned}$$

otherwise

$$G(x,a,b) = \Big(H(x-a,0)\ \cap\ H(x-L-a-b\cdot R,0)\Big) \cup \\ \Big(H(x-a,1)\ \cap\ H(x-L-a-b\cdot R,1)\Big)$$

Since $|Q(H)| = 2\cdot S$ we have $|Q(G)| = 8\cdot S^2$.

We use Equation 2.22 to construct the FA F which determines if $x \in \mathcal{BASE}$

$$F(x) = G(x,0,0) \cap \overline{G(x,-R,0)}$$

and so $|Q(F)| = 64\cdot S^2$.

The second versions

The number of states can be reduced by constructing a smaller FA and modifying the final states.

Let $H_1 = H(x-a,0)$ and $H_2 = H(x-L-a-b\cdot R,0)$. We define

$$G(x,a,b) = H_1 \cap H_2$$

The final states of G are a result of the intersection of H_1 and H_2, with

$$v(G)_i + 2\cdot S\cdot j = v(H_1)_i \cdot v(H_2)_j \text{ for } 0 \le i,j < 2\cdot S \tag{3.8}$$

The main purpose of the FA G is to compare two pseudoparity values for equality or inequality. We make use of the fact that $v(H)_i = [\lfloor i/S\rfloor = t]$ along with the index ordering information from Equation 3.8 in order to modify the final states of G to get the behaviour we are after.

If $\left||L+b\cdot R|_P\right|_2 = 0$ then

$$v(G)_{2\cdot S\cdot i+j} = [\lfloor i/S\rfloor \neq \lfloor j/S\rfloor] \text{ for } 0 \le i,j < 2\cdot S$$

otherwise

$$v(G)_{2\cdot S\cdot i+j} = [\lfloor i/S\rfloor = \lfloor j/S\rfloor] \text{ for } 0 \le i,j < 2\cdot S$$

This reduces $|Q(G)|$ from $8\cdot S^2$ to $4\cdot S^2$ states. We construct F in the same manner as before, and the reduction in the size of G leads to a correspondingly smaller F, with $|Q(F)| = 16\cdot S^4$.

The third versions

Up until this point the FA will work for CRRS with either version of the pseudorank, which is why we have been using the more generic S instead of 2^gor p_1. The next versions of the FA are only valid with the newer version, which has $S = p_1$, although we will continue to use S to maintain a consistent notation amongst the CRRS FA descriptions.

When $S = p_1$ we have $L = (r-1)\cdot R$, and so all of the translations of x that occur when computing if $x \in GB(a, b\cdot R)$ or $x \in \mathcal{BASE}$ will be multiples of R. These translations only effect the value of the residue associated with $p_1 = S$, and we use this to further reduce the number of states required by the FA F and G.

The FA are modified so that the first residue is the last residue to be processed. This could be done by re-ordering the moduli to be $p_2, p_3, \ldots, p_{r-1}, p_r, p_1$, although the only requirement is that $p_1 = S$ is the last modulus to be processed.

With this re-ordering done, the FA F, G, and H process strings in exactly the same manner until the last symbol is processed.

The FA are unary adders modulo $2\cdot S$ with $d = 0$ and $e_i = \mu_i(x-a) + S\cdot |\langle x\rangle_i|_2$, which corresponds to the matrix descriptions

$$u_i = [i = 0] \qquad\qquad w(i, \sigma_{\langle x\rangle_i})_{s,t} = \left[t - s \equiv \mu_i(x) + S\cdot |\langle x\rangle_i|_2 \mod 2\cdot S\right]$$

We refer to this as the basic modulo adder.

For these versions of the FA F and G we set $v(F)_i = v(G)_i = 1$ for $0 \le i < 2\cdot S$, which means that all states are final states. Since all states are final we need to use another mechanism to reject strings.

The FA are modulo unary adders so the matrices $w(i, \sigma_{\langle x\rangle_i})$ have non-zero entries along two diagonals. We reject strings by setting entries on these diagonals to zero.

Despite the fact that it is processed last, we will use $w(1, \sigma_{\langle x\rangle_1})$ to refer to the matrices associated with the modulus p_1.

For $0 \le s < 2\cdot S$, let

$$j(s,x,m) = \left\lfloor \frac{|s + \mu_1(x - m\cdot R) + S\cdot |\langle x - m\cdot R\rangle_i|_2|_{2\cdot S}}{S} \right\rfloor$$

If the FA is in state s after processing $r-1$ symbols of x, then

$$f(s,x,m) = |x - m\cdot R|_{\tilde{2}}$$

This is enough information to produce the FA F and G, as it allows us to determine which entries of the matrices to set to zero.

For $G(x,a,b)$ we begin with the basic modulo adder for $|x-a|_P$ rather than x.

The transition matrix for the symbols associated with p_1 is

$$w(1, \sigma_{\langle x-a\rangle_1})_{s,t} = [t = j(s, x-a, 0)]$$

If $r + b$ is odd, the transition matrix is modified such that

$$w(1, \sigma_{\langle x-a \rangle_1})_{s,t} = [t = j(s, x - a, 0)] \cdot [j(s, x - a, 0) \neq j(s, x - a, r - 1 + b)]$$

otherwise if it is modified to become

$$w(1, \sigma_{\langle x-a \rangle_1})_{s,t} = [t = j(s, x - a, 0)] \cdot [j(s, x - a, 0) = j(s, x - a, r - 1 + b)]$$

For $F(x)$ we use the basic modulo adder and change the transition matrix associated with p_1. It begins as

$$w(1, \sigma_{\langle x \rangle_1})_{s,t} = [t = j(s, x, 0)]$$

If r is odd it is changed such that

$$w(1, \sigma_{\langle x \rangle_1})_{s,t} = [t = j(s, x, 0)] \cdot [j(s, x, 0) \neq j(s, x, r - 1)] \cdot [j(s, x, -1) = j(s, x, r)]$$

otherwise it becomes

$$w(1, \sigma_{\langle x \rangle_1})_{s,t} = [t = j(s, x, 0)] \cdot [j(s, x, 0) = j(s, x, r - 1)] \cdot [j(s, x, -1) \neq j(s, x, r)]$$

Now the number of states is $|Q(F)| = |Q(G)| = 2 \cdot S$. While having FA with size linear in S is an advantage, the alterations to the transition matrices associated with the modulus p_1 prevent us from applying a particular optimization to the FA implementation. This optimization requires that the diagonals of the unary adders are unmodified, and is covered in more detail in Section 3.5.1

The fourth versions

We could have stopped when we got the size to $2 \cdot S$, but there is one more optimization that we can make. The methods described by Equation 2.25 and its variants can be used to implement FA for $x \in GB(a, b \cdot R)$ and $x \in \mathcal{BASE}$ with only S states.

The FA are based on modulo unary adders, and the transition matrices are unmodified which means that we can use the optimization that was not available with the last version of the FA.

The FA are unary adders modulo S with $d = 0$ and $e_i = \mu_i(x)$, which corresponds to the matrix descriptions

$$u_i = [i = 0] \qquad\qquad w(i, \sigma_{\langle x \rangle_i})_{s,t} = [t - s \equiv \mu_i(x) \mod 2 \cdot S]$$

The FA F is straightforward. We only have to set $v(F)_i = [i = 0]$.

For $G(x, a, b)$ we use the adder modified to work with $|x - a|_P$ instead of x and then set

$$v(G)_i = [0 \leq i < r - 1 + b]$$

3.5 Implementation and optimization

3.5.1 Optimizing the unary additions

The majority of the required storage space is used to store the transition matrices. This can be greatly reduced by specializing the matrix representation for the FA which behave as unary adders.

A FA being used as a unary adder will have transition matrices that have non-zero entries on a single upper diagonal of the matrix. If the FA F is being used as a unary adder $\mod |Q(F)|$ there will also be a lower diagonal of non-zero entries.

Consider a k-state FA uses its states as a unary adder and has a transition matrix M that is meant to add a to the total of the adder. The non-zero entries of the matrix will be

$$M_{i,a+i} = 1 \text{ for } 0 \leq i < k - a$$

and if we are dealing with a modulo unary adder we will have the additional non-zero entries

$$M_{k-a+i,i} = 1 \text{ for } 0 \leq i < a$$

It is clear from Figures 3.7 and 3.8 that these matrices have a large amount of structure and a large amount of empty space. Instead of storing k^2 entries for NFA matrices or k entries for DFA matrices we can store the index of the diagonal in $1 + \log k$ bits and whether or not the matrix represents a modulo adder with an additional bit. We will also need storage for the $O(m)$-bit weight if we are working with a WFA.

The change from $k^2 \cdot O(m)$ bits of storage to $\log k + 2 + O(m)$ bits of storage is significant in itself, but the "stripe" representation of the transition matrix also simplifies the vector - matrix multiplications that we see in FA evaluations.

Assume we are dealing with a FA which acts as a unary adder. Let c be a state vector, s be a transition matrix that adds a to the total of the adder and $d = c \cdot s$. For WFA there will be some weighting e associated with the stripe, and for other types of FA we have $e = 1$.

For a regular adder the elements of d will be

$$d_i = 0 \text{ for } 0 \leq i < a$$
$$d_{a+i} = e \cdot c_i \text{ for } 0 \leq i < k - a$$

and for a modulo adder they will be

$$d_i = e \cdot c_{k-a+i} \text{ for } 0 \leq i < a$$
$$d_{a+i} = e \cdot c_i \text{ for } 0 \leq i < k-a$$

In either case the vector-multiplication takes time $O(k \cdot (m + m \log m \log \log m))$, which is the same time bound as matrix-based DFA and an improvement on the time bound for matrix-based NFA. Stripe-based transition matrices reduce the memory requirements of a FA while maintaining or improving the time bound on the main operation they are involved in, and so we use them whenever we can.

The drawback to the stripe-based transition matrices is that operations such as union, intersection and projection will result in FA with transition matrices that cannot be represented with these stripes. In the experimental applications of this work the limiting factor has usually been memory rather than time. With this in mind, we devised an approach that lets us have our stripes and operate on them as well.

3.5.2 Organizing the operations

It is clear from Figures 3.5, 3.6 and 3.7 that the union operation in general and the intersection operation applied to unary adder FA will both result in transition matrices which have a large number of zero-valued entries, which gives us scope to compress the matrices.

We will refer to the FA which implements a unary adder as basic FA and will refer to the FA which are the result of FA operations as derived FA. The stripe optimization detailed in the last section can be applied to all of the basic FA that we deal with when working with CRRS.

In this section we describe how we efficiently represent the derived FA in a manner which allows us to continue to use the stripe optimization for the underlying basic FA. We refer to this alternate representation as the expression tree optimization.

The main operations that we use to create derived FA are union, intersection and projection. Any projection operations are performed in parallel with the evaluation of the FA, and remain almost unchanged.

The intersection operation distributes over the union operation. If we are given a FA derived form basic FA by some combination of union and intersection operations we will always be able to represent the FA as the union of intersections of basic FA.

For instance, if A, B, C and D are basic FA then

$$(A \cup B) \cap (C \cup D) = (A \cap C) \cup (A \cap D) \cup (B \cap C) \cup (B \cap D)$$

Define an intersection list to be an ordered list of basic FA which have had the intersection operation applied to them. Note that an intersection list is also a representation of a particular

FA. Define a union list to be an ordered list of intersection lists which have had the union operation applied to them. Any FA that is the result of any number of union and intersection operations applied to basic FA can be represented by a union list. We refer to the nested list as an expression tree.

The members of the union list can be evaluated sequentially. The time required for any kind of evaluation will be the sum of the time required for the same evaluation on each of the operands to the union. Similarly, the maximum working space required will be the maximum of the working space required by each of the operands.

The intersection will have been performed on a series of single stripes. The vector-matrix multiplications in the FA evaluation can be carried out in a similar manner to the vector-matrix multiplication described when the stripe optimization was introduced.

Let $F = \bigcap_{j=1}^{k} G_j$. We perform the vector-matrix multiplication by performing the k shifts or rotations corresponding to the stripes with a few alterations.

The vector manipulations treat the subsequent intersections as if they were a single FA. As a result of this the hth vector manipulation will be performed as if each block of $\prod_{i=h}^{k} |G_i|$ vector elements were a single element. The $|G_h|$ blocks of states are repeated $\prod_{i=1}^{h-1} |G_i|$ times in the top-level vector, and so the manipulation needs to be performed for each of these repetitions.

It is clear that this will take at most time $k \cdot |Q(F)| \cdot O(m + m \log m \log \log m)$ to perform.

We cannot sum expression trees, and so we are forced to use the slower $\pi_* F(x)_{\Pi,\Sigma}$ variant when determining the cardinality of a language. The faster vector-matrix multiplication counteracts this, to the extent that the running time is largely unchanged.

The main effect of the use of the expression tree is a reduction in the memory requirements. The transition matrices are replaced by a single value per symbol per operand in the expression tree. The initial and final state vectors are components of the FA in the expression tree, which take up less space since than in the matrix representation where the unions and intersections would have caused them to grow.

The current state vectors are now the largest contributor to the memory requirements. We need enough memory for two current state vectors, which will be the size of the largest union operand in the expression tree.

The time required to evaluate FA is slightly altered by these techniques.

Consider $F = (A \cup B) \cap (C \cup D)$, given that the FA A, B, C and D are basic FA. Note that

$$|Q(F)| = (|Q(A)| + |Q(B)|) \cdot (|Q(C)| + |Q(D)|)$$

Let the matrix entries be $O(m)$-bit integers.

Time and memory characteristics of F are presented in Table 3.1 and Table 3.2, which give the values for DFA and NFA respectively. Table 3.3 presents the same information when the stripe and expression tree optimizations are used.

Time ($F\langle x\rangle$)	$(r+1)\cdot\lvert Q(F)\rvert\cdot O(m+m\log m\log\log m)$
Time ($\pi_* F\langle x\rangle$)	$\lvert Q(F)\rvert^2\cdot(\sum_{i=1}^r p_i\cdot O(m)+r\cdot O(m+m\log m\log\log m))$
Memory (evaluation, O(m)-bit entries)	$3\cdot\lvert Q(F)\rvert$
Memory (storage, O(m)-bit entries)	$2\cdot\lvert Q(F)\rvert+\sum_{i=1}^r p_i\cdot\lvert Q(F)\rvert$

Table 3.1: Time and memory characteristics of DFA with matrix representation.

Time ($F\langle x\rangle$)	$(\lvert Q(F)\rvert+r\cdot\lvert Q(F)\rvert^2)\cdot O(m+m\log m\log\log m)$
Time ($\pi_* F\langle x\rangle$)	$\lvert Q(F)\rvert^2\cdot(\sum_{i=1}^r p_i\cdot O(m)+r\cdot O(m+m\log m\log\log m))$
Memory (evaluation, O(m)-bit entries)	$\lvert Q(F)\rvert^2+2\cdot\lvert Q(F)\rvert$
Memory (storage, O(m)-bit entries)	$2\cdot\lvert Q(F)\rvert+\sum_{i=1}^r p_i\cdot\left\lvert Q(F)^2\right\rvert$

Table 3.2: Time and memory characteristics of NFA with matrix representation.

Time ($F\langle x\rangle$)	$4\cdot(2\cdot r+1)\cdot\lvert Q(F)\rvert\cdot O(m+m\log m\log\log m)$
Time ($\pi_* F\langle x\rangle$)	$4\cdot(2\cdot\sum_{i=1}^r p_i+1)\cdot\lvert Q(F)\rvert\cdot O(m+m\log m\log\log m)$
Memory (evaluation, O(m)-bit entries)	$2\cdot\max(\lvert Q(A)\rvert,\lvert Q(B)\rvert)\cdot\max(\lvert Q(C)\rvert,\lvert Q(D)\rvert)$
Memory (storage, O(m)-bit entries)	$2\cdot(\lvert Q(A)\rvert+\lvert Q(B)\rvert+\lvert Q(C)\rvert+\lvert Q(D)\rvert)+\sum_{i=1}^r p_i$

Table 3.3: Time and memory characteristics of FA with the stripe and expression tree optimizations.

3.5.3 Reducing the storage space

The u and v vectors are now the largest contributors to the memory requirements.

The number of states in a CRRS FA is usually a polynomial in one of the CRRS parameters such as r or n, and so the size of the u and v vectors generally increases with the size of the CRRS.

Let n be the parameter that uniquely defines the CRRS. For a basic FA we define the functions $g_u(n)=u$, $g_w(n,i,\sigma)=w(i,\sigma)$ and $g_v(n)=v$. If we represent the FA with the tuple (g_u,g_w,g_v) then the storage space required is the sum of the code size of these functions.

If these functions treat all values of n equally then the functions will require $O(1)$ bytes of storage, since the code size will be independent of the CRRS in use. This reduces the amount of memory required to store a FA. The working space for the operations will be unchanged.

This technique combines the construction of the FA with the evaluation operations, and so it should probably be avoided if a single FA is going to be used to evaluate a large number of input strings.

Without the stripe and expression tree optimizations the memory benefits would be negated, as $g_w(n,i,\sigma)$ would be creating a $\lvert Q(F)\rvert\times\lvert Q(F)\rvert$ matrix every time it was called. This would also require the use of the $\pi_* F(x)_{\Pi,\Sigma}$ variant when determining the cardinality of a language, which would be unacceptably slow.

3.5.4 An interesting alternative

We have so far looked at a set of techniques which exploit the highly structured transition matrices which occur when working with CRRS FA. One of the techniques that was considered but found lacking in performance was interesting enough to warrant a description.

We start by describing the stripe optimization in terms of polynomials.

For some FA F let $n = |Q(F)|$. If c_i is the ith entry in the vector of current states, we define the polynomial

$$C(z) = \sum_{i=0}^{n-1} c_i \cdot Z^i$$

The initial and final state vectors have polynomials defined for them in a similar manner.

A stripe-based transition matrix which uses the states for unary addition can be represented by

$$A(z) = z^a$$

where a is the fixed addend.

The vector-matrix multiplication which updates the state vector is

$$C(z) \cdot A(z) \bmod z^n$$

for the regular adders and

$$C(z) \cdot A(z) \bmod (z^n - 1)$$

for the modulo adders.

The multiplication takes time $O(n \log n \log\log n)$ and requires $O(n)$ elements of temporary memory, and so is worse than the previous scheme by both metrics.

On advantage is that if we pre-allocated all of the elements in the polynomial we can accumulate values into the polynomials, and so the projection operation and the logical product can then both use $\pi F(x)_{\Sigma,\Pi}$ instead of the slower $\pi F(x)_{\Pi,\Sigma}$.

We can also use the polynomial representation of the FA to replace the intersection lists of stripe-based transition matrices.

The polynomial

$$A(z_1, z_2, \ldots, z_{k-1}, z_k) = \prod_{i=1}^{k} {z_i}^{a_i}$$

is functionally equivalent to a k length intersection list of unary adder FA.

This requires that we use k-dimensional matrices as out state vectors, although this is only a conceptual change as the state vectors will contain the same number of elements in either case. In fact, this is only really a change in how we view the state and transition data which allows us to make use of an efficient multiplication algorithm.

Assume that each G_j in $F = \bigcap_{j=1}^{k} G_j$ has $|Q(G_j)| = t$. The multiplications are carried out hierarchically so the running time will be $O\Big((t \log t \log\log t)^k\Big)$. The multiplication will also require $O(t^k)$ element of temporary memory.

The previous scheme has a running time of $O(k \cdot t^k)$ for the same FA and was selected because $k = O(1)$ for all of our work. If k was related to the problem size we would have had to compare and contrast the performance characteristics for that particular problem. The benchmarks in Appendix A appear to indicate that the use of expression trees and stripe-based transition matrices may well be preferable, since the performance of the multivariate polynomial multiplication suffers greatly as the number of variables increases.

Chapter 4

Difficult CRR problems

In this chapter we look at the difficulty of applying the FA techniques from Chapter 3 to the harder CRR problems that were first mentioned in Chapter 2. The literature describing various attempts at efficient implementations for these difficult problems, and will then show that those problems cannot be efficiently solved with the FA model of computation unless $P = NP$.

4.1 An NP complete problem in CRR

4.1.1 Turing machines, P and NP

Turing machines

Turing [68] and Church [13] independently developed methods to formalize the description of any computation that can be carried out by following a finite set of instructions. The two methods are provably equivalent.

Church described these computations in terms of functions, including functions which could operate on and return other functions. The rules which govern the use of these functions are known as the typed lambda calculus. Turing formalized computations in terms of an abstract machine, now known as a Turing machine (TM). The bulk of the theory of computation is defined in terms of Turing machines, and so we will focus out attention on them.

A TM can be viewed as an extension to a FA. The only memory that FA have access to is encoding in the states of the FA, where TMs have access to an infinite amount of sequentially accessed memory This memory is referred to as a tape and is accessed one symbol at a time via a tape head, which is under the control of the TM.

Let $\sigma[i] \in \Sigma$ be the symbol in the ith position of the tape. The tape is normally initialized by writing the input string to the tape starting at position 0. The machine begins in an initial state q_i and with the tape head positioned at the start of the input string.

The behaviour of the machine is defined by a transition function.

$$\delta(q_a, \sigma_b) = \{q_c, \sigma_d, m\}$$

where $m \in \{-1, +1\}$.

A TM in state q_a with its tape head at position i will read the symbol $\sigma_b = \sigma[i]$ from the tape. The TM will write the symbol σ_d to the current position of the tape, change the position of the tape head to $i + m$, and then change its state to q_c. If the TM transitions to one of the final states the computation halts, otherwise it reads another symbol and continues the computation.

A TM can be designed to handle decision problems, which are problems which can be answered "yes" or "no", or search problems, which require a more substantial answer. A TM for a decision problem will divide the final states into accepting and rejecting final states, which will be used to indicate the answer to the problem when the machine halts, where a TM for a search problem simply writes the answer on the tape of the machine immediately before halting.

A universal Turing machine (UTM) is able to simulate any other TM. If the simulated TM halts after n steps the UTM will halt after n^k steps for some constant k.

In the case where a physical computer has at least as much memory as a given algorithm requires, the physical computer can be viewed as a UTM. The result of this is that the analysis of the behaviour of a problem for a TM can provide useful information about how the problem will behave when realized on a physical computer.

Running times, complexity classes and P

The Church-Turing thesis states that any algorithm has an equivalent TM. If we are interested in the running time of an algorithm we can analyse the behaviour of the corresponding TM.

The running time is usually an upper bound on the number of steps required to carry out the algorithm on a Turing machine, expressed as a function in one or more parameters of the problem. The parameter or parameters of such a function define the size of the problem, which is typically something like the number of bits needed to represent a number or the number of elements in a list. Using a function in the problem size to express the running time allows us to reason about the behaviour of the algorithm as the problem size changes.

This does not offer any guidance on what constitutes a reasonable running time. Cobham [14] and Edmonds[22] are credited as the first to seriously consider the problem. The guideline that came from their work was that an algorithm runs in a reasonable amount of time if the worst case running time is expressible as a polynomial in the problem size.

This gave rise to the complexity class known as P. A complexity class is a set of problems with a common property, normally related to the running time or memory requirements of the problem. For running time based complexity classes, the problems are partitioned by characteristics of the function for the running time.

A problem belongs to the complexity class P if the upper bound on the running time is a polynomial in the size of the input, which means that the running time of a problem in P can be expressed in asymptotic notation as $O(n^k)$ for some constant k. We say that such a problem has polynomial running time.

We can define the complexity class EXP in a similar manner, with EXP containing the problems with running time $O(k^n)$ for some constant k. As all problems in the class EXP are also in the class P we say that P is contained in EXP.

Non-determinism and P

In Chapter 3 we described deterministic finite automata and non-deterministic finite automata. NFA are similar to DFA, but have multiple choices that can be made at each step. A NFA "guesses" which choices to make in order to reach an accepting final state whenever possible.

We can think of a NFA as "splitting" into multiple instances whenever it needs to make a choice, and so at any point in the computation we have a set of NFA for every sequence of choices that has been made. This set of virtual NFA accepts an input string if any of these instances accept the input string.

A non-deterministic Turing machine (NTDM) is a similar concept. The transition function of a TM is modified so that the transition function may specify multiple outcomes for each combination of state and symbol.

The complexity class NP is the set of problems that have a running time of $O(n^k)$ on a NDTM for some constant k. NP was first discovered by Cook [15], with a focus on decision problems, and independently by Levin [44], who focused on search problems.

The output of any search problem in the class NP can be verified in polynomial time. This is a consequence of the fact that a NDTM can guess all possible polynomially sized output values in polynomial time and then use the verifier to determine which of the outputs should be accepted. If the output was larger than polynomial in size the NDTM would be unable to generate it in polynomial time, and if the verifier took more than polynomial time the NDTM would be unable to determine which of the guessed outputs to accept.

The ability to make a guess at every computational step means that a NDTM running in polynomial time can make a correct guess from a set that is exponential with respect to the problem size. Any problem in NP can be solved by a DTM that simulates the appropriate NDTM, however any without information about the guesses the DTM will run in EXP time, since in the worst case each of the guessed values will need to be checked.

An alternative interpretation is that the output of a NDTM for a search problem is an encoding of the correct guesses made by the NDTM. A DTM with access to this information would know which paths to take and so would have polynomial running time.

4.1.2 NP-complete and NP-hard

Let $r(b) = a$ be an algorithm in P that will transform the instance b of the problem B into an instance a of the problem A. Note that the size of a is polynomial in the size of b. The mapping r is called a reduction if and only if $b \in B \Leftrightarrow a = r(b) \in A$, and if r takes polynomial time in the problem size then r is a polynomial time reduction from A to B.

If we only know that A is in either P or in EXP, then the existence of a reduction from A to B means that if $A \in P$ then $B \in P$ and if $A \in EXP$ then $B \in EXP$. This is a consequence of the transitivity of polynomial reductions.

In the scope of the complexity classes considered in this thesis, if a problem D has a running time that is at least polynomial in the problem size, then a polynomial time reduction from D to E means that D and E are in the same complexity class, provided that the complexity class is defined in terms of running time.

Karp [39] defined the complexity classes NP-hard and NP-complete in terms of polynomial time reductions. A problem A is NP-hard if every problem in NP is polynomial time reducible to A, and is NP-complete if $A \in NP$ and A is NP-hard.

NP-hard problems are the problems that are at least as hard as the hardest problems in NP. The brute force DTM simulation of a NDTM, for instance, is clearly NP-hard. Since the NP-complete problems are the NP-hard problems that are also in NP, it follows that the NP-complete problems are the hardest of the problems in NP.

The Cook-Levin theorem [15], [44] shows that a NDTM can be simulated by a specially crafted instance of the Boolean satisfiability problem (SAT), and that the problem size of the SAT instance is polynomial in the size of the simulated problem. The result of this is that all problems in NP, and hence all NP-complete problems, are polynomial time reducible to SAT. This makes SAT NP-hard, and since $SAT \in NP$, SAT is also NP-complete.

Most other NP-complete problems are proved to be NP-hard by demonstrating a polynomial time reduction to another NP-complete problem. This can be done since the polynomial time reduction from the problem A to the NP-complete problem B can be combined with the polynomial time reduction from B to SAT, and the combined reduction will also have polynomial running time.

Karp also showed that 21 well known problems were NP-complete. Each of the problems was considered to be intractable at the time. The classification of these problems as NP-complete was a step forward for the study of these problems, as it showed that they were all intractable for similar reasons. Today there are thousands of problems known to be NP-complete, and around 300 are described in the book on NP-completeness by Garey and Johnson [25]

If any of the NP-complete problems was proved to be solvable in polynomial time then the classes P and NP would be equivalent, although nobody has been able to show that $P = NP$ or $P \neq NP$. The usefulness of a polynomial time algorithm for the problems in NP, the theoretical implications of a result in either direction, and the fact that it is a long-standing open question have raised considerable interest in the $P? = NP$ question.

4.1.3 QMOD and QCRR

The $QMOD$ problem is to determine if there exists a non-negative integer x such that

$$x^2 \equiv a \bmod b \ \wedge \ x < c \tag{4.1}$$

We use $QMOD(a, b, c)$ to specify instances of the problem.

It is safe to assume that $a, c, x < b$. If it is clear from Equation 4.1 that $a < b$. Assume that y is a solution to the $QMOD(a, b, c)$ and $k \cdot b \leq y < c$ for some non-negative integer k. This means that

$$(y - k \cdot b)^2 \equiv a \bmod b \ \wedge \ (y - k \cdot b) < (c - k \cdot b)$$

holds and $|y|_b$ is a solution for the $QMOD(a, b, |c|_b)$. We treat the instance with $c, x < b$ as the canonical $QMOD$ instance for a given value of b. A result of this is that we use $\lceil \log b \rceil$ as the problem size of a $QMOD$ instance.

The $QCRR$ problem is the equivalent of the $QMOD$ problem in a CRRS setting if $b^2 < P$. The $QCRR$ problem is to determine if there exists non-negative integers x and y such that

$$\left(\bigwedge_{i=1}^{r} (\langle x^2 \rangle_i = \langle a + b \cdot y \rangle_i) \right) \ \wedge \ (x < c) \ \wedge \ (y < b) \tag{4.2}$$

We use $QCRR(a, b, c)$ to specify an instance of the problem, where the instance is implicitly tied to whichever CRRS we are working with at the time. We assume that $a, c, x, y < b$ for the sames reasons that we assume $a, c, x < b$ for $QMOD$.

Lemma 4.1. *If $b^2 < P$ then $QMOD(a, b, c) \Leftrightarrow QCRR(a, b, c)$.*

Proof. Note that

$$x^2 \equiv a \bmod b$$

is equivalent to

$$x^2 = a + b \cdot y$$

for some non-negative integer y.

If $b^2 < P$ then $x^2 < P$, because $x < b$, and so

$$x^2 = a + b \cdot y \Leftrightarrow x^2 \equiv a + b \cdot y \bmod P$$

A value for y will always exist if $QMOD(a, b, c)$ has a solution.

There will be values of y that exist for the non-solutions of $QMOD(a, b, c)$ where

$$x^2 = a + b \cdot y \ \wedge \ c \leq x < b$$

but they will be removed by the restriction that $x < c$.

It is clear that $y < b$ since $x^2 = a + b \cdot y$ and $x^2 < b^2$. This means that we can find a value for $y < b$ for any instance of $QMOD$ which has a solution.

Since all values are less than P we have

$$\bigwedge_{i=1}^{r} (\langle x^2 \rangle_i = \langle a + b \cdot y \rangle_i)$$

without ambiguity.

This is all that it is required to show that $QMOD$ and $QCRR$ are equivalent when $b^2 < P$ It is clear that equivalence stated by the lemma holds.

□

Manders and Adleman [47] showed that $QMOD$ is NP-complete. They did this by giving a polynomial time reduction from a special form of the $KNAPSACK$ problem to $QMOD$. The paper actually gives a more general result, showing that recognizing a specific set of Diophantine equations is NP-complete.

The modular arithmetic used in the reduction of $KNAPSACK$ to $QMOD$ is very similar to the arithmetic used when working with CRR, although the general approach taken means that the paper never states the problem in terms of $QCRR$.

4.2 Difficult problems in CRR

4.2.1 CRRS FA and Boolean circuits

In previous discussions of CRRS FA we have focused on the time required to evaluate an input string for an CRRS FA with a certain number of states. All of the CRRS FA which we have defined so far have been defined with a number of states and symbols which can be expressed as a polynomial in the CRRS problem size. For the purposes of this discussion we will often use the term polynomial as shorthand for polynomial in the size of a particular CRRS.

What we have not discussed is the time that it takes to construct these CRRS FA. The CRRS FA can be thought of as specialized Boolean circuit, so we borrow some Boolean circuit terminology to talk about the time required for the construction of particular FA.

A Boolean circuit is said to be uniform if some resource bound of the construction of the circuit, typically the time or space required, can be expressed as a computable function of the number of its arguments. If this is not the case then the circuit is said to be nonuniform. We can further classify these circuits by looking at the problem of generating the circuit and the complexity class that this problem belongs to. The P uniform Boolean circuits, for instance, are the Boolean circuits which are able to be constructed in polynomial time.

This means that a P uniform Boolean circuit cannot have more than polynomially many states. Since a Boolean circuit with polynomially many gates can be evaluated in polynomial time, any problem that can be expressed as a P uniform circuit will be in the complexity class P.

We will refer to any FA which can be implemented as a P uniform Boolean circuit as a P uniform FA. All of the CRRS FA which were defined in Section 3.4 are P uniform FA. Additionally, any FA which has polynomially many states and is defined in terms of these basic CRRS FA and the FA operations described in Section 3.3 will be a P uniform FA.

4.2.2 QCRR and the concept of "difficulty"

We can restate $QCRR(a, b, c)$ as the problem of determining if there exists any non-negative integer x in a CRRS such that

$$x < c \ \wedge \ (x^2 - a) < b^2 \tag{4.3}$$

Note that the problem size of a $QCRR$ instance with $b^2 < P$ is polynomial in the size of the CRRS.

The search problem for $QCRR$ will output a value of x for which Equation 4.3 holds, and we can verify the value of x in polynomial time since $QMOD$ and $QCRR$ are NP-complete. The verification requires two CRR comparisons, which implies that there is some method by which we can carry out CRR comparisons in polynomial time.

Consider a hypothetical P uniform FA F for CRR comparison. We can create a CFA from the intersection of the two instance of F corresponding to the comparisons in Equation 4.3. This CFA would be able to be constructed and evaluated in polynomial time, and since $QCRR \subset NPC$ the existence of F would mean that $P = NP$.

Let x be the input to a decision problem A in NP, and let y be some encoding of the guesses made by the NDTM. If x and y are polynomial in the problem size then the problem is equivalent to determining whether a value of y exists that satisfies some relation $R(x, y)$, where R can be determined by a DTM with polynomial running time.

The guessing that occurs when F is used to solve $QCRR$ is very similar, with the exception that the relation $R(x, y)$ must be able to be determined by a P uniform FA.

We have previously described several CRR operations as being "difficult" without saying much about what makes a CRR operation difficult. If the existence of a P uniform FA for a problem implies that P = NP we refer to the problem as difficult. Comparison is one of these operations.

In the next few sections we will look at the concept of "difficulty" in more detail, the various difficult problems and the approaches to these problems in the literature.

4.2.3 The class of difficult problems

Part of this discussion will involve classifying algorithms for CRRS problems as online or offline. An online algorithm is an algorithm which can read each symbol of its input only once, whereas an offline algorithm can read the input symbols multiple times. A FA which processes an input string with polynomial length will be unable to encode the input into polynomially many states, and as such will not have the ability to access any of the symbols of the input string more than once. FA have no memory except for the memory encoded by their states, and so any algorithm for a CRRS FA with polynomial many states will be an online algorithm.

We redefine the concept of a reduction for use with CRRS FA. A problem A is reducible to B if we can construct a FA for A from a FA for B where the FA for A has at most polynomially many more states than the FA for B. If we can define a reduction from A to B and a reduction from B to A we say that A and B are inter-reducible. Note that inter-reducibility is transitive, so if A and B are inter-reducible and B and C are inter-reducible then A and C are inter-reducible.

These reductions can make use of the CRRS FA mentioned in Chapter 3, all of are P uniform. If all of the FA used in the reduction have $O(n^d)$ states then we can use up to $O(n^e)$ union operations or $O(1)$ intersection operations for the reduction, where d and e are constants.

The reductions are defined in terms of P uniform CRRS FA but can also be treated as the description of polynomial time reduction for a TM. This means that if an offline polynomial time algorithm exists for B then the reduction can be used to construct an offline polynomial time algorithm for A.

Section 4.2.4 lists a number of these difficult problems and Section 4.3 examines the techniques that have been applied to these problems in the CRR literature. In Section 4.4 will show that the CRRS problems that are generally recognized to be hard are all "difficult" by our definition. This is done by describing a spanning-tree of inter-reductions between the problems these hard problems. The inclusion of comparison in this spanning-tree and the transitivity of inter-reducibility are enough to show that if any of these problems can be solved by with P uniform FA then P = NP.

In fact, under these conditions we could use the logical product to count the number of solutions to $QMOD$. Any problem which determines the number of accepting paths of a NDTM is in the complexity class $\#P$. The idea of $\#P$-completeness is similar to NP-completeness, with the additional restriction that the reductions must preserve the number of solutions. We do not know if this is the case for $QMOD$, but if it is then a P uniform FA for a difficult CRR problem would imply that $P = \#P$.

4.2.4 Specific difficult CRR problems

Signed CRRS uses the range $0 \leq x \leq \lfloor P/2 \rfloor$ for positive integers and the range $\lceil P/2 \rceil \leq x < P$ for negative integers. Determining the sign of such a number is a difficult problem because it is equivalent to CRR comparison.

We define the FA $COMP(x, y)$ such that

$$[COMP\langle x, y\rangle] = [x < y]$$

Comparison between two operands is actually more than we need. We can define the FA $HALF(x, k)$ such that

$$\begin{aligned}[HALF\langle x, 0\rangle] &= [0 \leq x \leq \lfloor P/2\rfloor] \\ [HALF\langle x, 1\rangle] &= [\lceil P/2\rceil \leq x < P]\end{aligned}$$

and use that for sign detection, which is more efficient than using $COMP$.

Note that detecting additive overflow in a CRRS can be done with several instances of $HALF$. We present a sketch, which ignores $O(1)$ edge cases, of a method for determining if overflow has occurred when computing $x + y = z$.

$$\begin{aligned}HALF(x, 0) \cap HALF(y, 0) &\Rightarrow \text{no overflow} \\ HALF(x, 0) \cap HALF(y, 1) \cap HALF(z, 1) &\Rightarrow \text{no overflow} \\ HALF(x, 1) \cap HALF(y, 0) \cap HALF(z, 1) &\Rightarrow \text{no overflow} \\ HALF(x, 0) \cap HALF(y, 1) \cap HALF(z, 0) &\Rightarrow \text{overflow} \\ HALF(x, 1) \cap HALF(y, 0) \cap HALF(z, 0) &\Rightarrow \text{overflow} \\ HALF(x, 1) \cap HALF(y, 1) &\Rightarrow \text{overflow}\end{aligned}$$

Determining if a given x is bad is also a "difficult" problem, as are determining the rank and the parity of x, and so we define the FA BAD,

$$[BAD\langle x\rangle] = [q(x) \neq \tilde{q}(x)]$$

$RANK$

$$[RANK\langle x, k\rangle] = [q(x) = k]$$

and $PARITY$

$$[PARITY\langle x, k\rangle] = [|x|_2 = k]$$

respectively.

We have seen in Chapter 2 that there is a relationship between the rank and which bin we are in, so it should be no surprise that determining which bin we are in is also a difficult problem. We define the FA

$$[BIN\langle x, k\rangle] = [\lfloor x/R\rfloor = k]$$

for the task.

Finally, we will look at the base extension problem. A CRRS undergoes base extension if an extra modulus is added to the set of moduli. We will refer to the new modulus as p_{r+1}, and will require that p_{r+1} is co-prime with P. Given the residues of x in the original CRRS, the base extension for a CRRS integer is to determine the residues of x in the extended CRRS.

The original residues $\langle x \rangle_i$ need to be modified to include the modular inverse of p_{r+1}. We will use $\langle \hat{x} \rangle_i$ for the extend residues, and have

$$\langle \hat{x} \rangle_i = \left| \langle x \rangle_i \cdot {p_{r+1}}^{p_i - 2} \right|_{p_i} \text{ for } 1 \leq i \leq r$$

Since p_{r+1} is co-prime to the original moduli the $\left|{p_{r+1}}^{p_i-2}\right|_{p_i}$ factor will act as a generator. This means that the first r extended residues will still uniquely encode an integer $0 \leq x < P$. It is clear that for every set of residues $\langle \hat{x} \rangle_i$ with $1 \leq i \leq r$ there will be exactly one value of the residue $\langle \hat{x} \rangle_{r+1}$ such that the set of $r+1$ extended residues encodes an integer from the original CRRS.

The value of the new residue is

$$\langle \hat{x} \rangle_{r+1} = \left| |x|_{p_r} \cdot P^{p_r - 2} \right|_{p_{r+1}}$$

which relies on

$$|x|_{p_{r+1}} = \left| \left(\sum_{i=1}^{r} \frac{P}{p_i} \langle x \rangle_i \right) - q\left(x\right) \cdot P \right|_{p_{r+1}}$$

All of the arithmetic can be carried out modulo p_{r+1}, and so the use of the rank is the most computationally expensive part of the base extension.

We define the FA $EXTEND(x, p_{r+1}, x_{r+1})$ such that x is accepted if and only if $\langle \hat{x} \rangle_{r+1} = x_{r+1}$ in the CRRS extended by the modulus p_{r+1}.

4.3 Historical approaches

4.3.1 Motivations

For any of these difficult problems, we would like to be able to define a FA which has polynomial many states and an efficient method for computing the logical product. We will need to encode the intermediate values of any calculations into the states of the FA, and so these values will need to be quite small or at least compressible.

The CRR literature appears to recognize that meeting these requirements is not an easy task. The focus in the literature has been on approaches that maximize the speed of hardware implementations. This is unsurprising given the potential gains that can be realized from using CRR to parallelize arithmetic operations. The price paid for these gains is the need for an overflow detection technique that avoids an overall decrease in performance.

Those interested in hardware implementations of CRR arithmetic have their own priorities. In general, they are looking for solutions which can be implemented with high parallelism and low latency. Look-up tables are used when appropriate, as is normally the case for the additions modulo small primes. Redundant information is used when it is faster than any alternatives.

In this section we look at a number of interesting approaches that have been taken over the years.

4.3.2 Selected techniques from the literature

The earliest and most cited text on CRR is by Tanaka and Szabó [66]. As we discussed in Chapter 2, Szabó also showed that sign detection requires access to every bit of $\langle x \rangle_i$ for each $p_i < \sqrt{P}$.

Some of the earlier work in this area focused on the detection and correction of hardware errors. This is less of a problem in modern hardware implementations and is a non-issue for theoretical uses of CRR. These approaches made use of redundant moduli, as adding t extra moduli will allow for the detection of t errors and the correction of $\lfloor t/2 \rfloor$ errors. This work is interesting to us because the error detection mechanisms could also be used to detect arithmetic overflow.

The techniques all involve the effect of base extension operations which include redundant moduli and testing for values in the redundant moduli that indicate that a particular integer is outside of the non-redundant range.

Mandelbaum [46] provided two methods for error detection and correction which were extended and improved by Barsi and Maestrini [4]. The improvements included a tightening of the conditions governing the choice of extra moduli and the ability to distinguish between an error and additive overflow where possible. Yau and Liu [71] independently developed the same conditions for a similar scheme. Their focus was solely on error correction and so they make no mention of additive overflow, although in settings where errors are either not possible or very unlikely the technique can be used to detect additive overflow without modification.

Unfortunately the use of redundant moduli limits the cases where the logical product can be applied sensibly. In all of the cases that we have seen, whenever an integer is identified as "good" by a particular set of redundant moduli there will be another set of redundant moduli that will identify the integer as "bad". This severely limits our ability to include redundant moduli in any online algorithm for the difficult CRR problems.

Dimauro, Impedovo and Pirlo [20] have a unique approach to CRR comparisons. Their approach defines and makes heavy use of the quantity $SQ = \sum_{i=1}^{r} \frac{P}{p_i}$.

After examining the relationship between SQ and several other properties of the CRRS, they define a function $D(x)$ that increases monotonically with x. This means that $x < y$ if and only if $D(x) < D(y)$, and so $D(x)$ can be used for comparison.

The technique is faster than naive CRR comparison, although the magnitude of SQ makes this technique less practical than other alternatives. Their paper is less concerned with the size of the operands to the intermediate calculations than most others, although the authors address this near the end of the paper and provide a sketch of a variation which operates with smaller values.

The variation does not cover all cases, and so if $D(x) = D(y)$ in the variation then the original version of the algorithm must be used. This is similar to several other methods that operate on CRR properties with higher and higher precision until a definitive answer is found. The idea behind the variation can be reused to arbitrarily limit the size of the intermediate values, however this is only implied by the paper.

Shenoy and Kumaresan [62] present a clever method of computing the rank in a paper discussing efficient methods for CRR base extension.

Their approach uses a redundant modulus or redundant moduli which are larger than the maximum value of the rank in the non-redundant part of the CRRS. A calculation is specified which guarantees that the redundant residue or residues will be equal to the rank. They then describe a method which makes use of the rank to efficiently compute base extensions.

The arithmetic is all performed with $O(\log p_r)$-bit integers, which appeals to us. While their method uses less redundancy than the earlier approaches that used redundant CRRS, it is enough to rule out the possibility of making use of their technique to implement an online rank algorithm.

Approximations to the rank

Hung and Parhami [37] performed an analysis on the errors introduced by approximations to the rank.

The approximations considered are similar to the original pseudorank as discussed in Chapter 2, which was based on the paper by Vu [69]. Where the pseudorank that we defined only truncates the approximation Hung and Parhami consider both approximations made by truncation and approximations made by rounding to the nearest integer. The distribution of the errors and the worst case errors are derived in terms of the number of bits used for the approximation.

The worst case error was shown to be close to the upper bound derived by Vu. Based on the assumption that the error term for each modulus is uniformly distributed, the magnitude of the errors was shown to follow a normal distribution. This will be examined in further detail in Chapter 5

Orton, Peppard and Tavares [56] present two approaches to overflow detection, and their first approach uses redundant moduli and an approximation to the rank.

They present an analysis of the rounding errors for their particular approximation of the rank, and then show that approximation is sufficiently accurate when in the presence of the redundant moduli. The goal of this approach is to have similar performance to other approaches which use redundant moduli while having a smaller range of values in the “dead zone”.

An algorithm is given to convert a set of CRR moduli into a single quantity which is then used for overflow detection. A certain number of the high bits of the generated quantity are guaranteed to have the same value if the moduli represent a number in the non-redundant range.

While still worth mentioning as an approach to a difficult CRR problem, this technique is not very useful for us. Even if it did not make use of redundant moduli, the generation of a number from all of the moduli rules out this technique for use in an online algorithm, as it as computationally expensive as computing the rank.

Their second approach uses redundant moduli and a base extension. The rank is calculated by a process similar to that of Shenoy and Kumaresan, and then is used to perform a consistency check of the extended moduli in a similar manner to the method described by Yau and Liu.

Division algorithms

The problem of integer division in CRRS, determining $z = \lfloor x/y \rfloor$, is more involved than most of the difficult problems. There has been considerable interest in the problem for use in hardware implementation and for reasoning about the complexity of integer division in general.

A paper by Lu and Chiang [45] describes an algorithm for CRR division. The use the techniques presented by Vu to find the rank of a number, from which they compute the parity of the number. The parity is used to perform comparison via the same reduction given in Section 4.4. The precision required to calculate the rank rules these techniques out for our use.

The division algorithm they describe works by performing a binary search that determines the quotient. They determine $Z = \lfloor X/Y \rfloor$ by search for k such that $Y \cdot 2^k \leq X < Y \cdot 2^{k+1}$, and then find the difference between 2^k and the quotient.

Another CRR division algorithm is given by Davida and Litow [19]. The algorithm involves the computation of the rank, parity and CRR comparisons, and makes use of the pseudorank in the same form as we discussed in Chapter 2.

A problem very similar to $HALF(x)$ is calculated from the pseudorank by binary search. The pseudorank of $\left|2^k x\right|_P$ is computed for $0 \leq k < \log P$ in parallel, and the results are used as the leaves of a binary tree. The pseudorank of $2^{k+1}x$ is only used by the algorithm if $2^k x$ was not inside the rank in which the pseudorank is known to be accurate. This is all performed with a Boolean circuit which is used to implement an algorithm for general comparison, and this approach to comparison is shown to be in log-space uniform NC^1.

A problem with size n is in the class NC^1 if it can be solved by a Boolean circuits with $O(n^k)$ logic gates and $O(\log n)$ depth. These problems are particularly suited to parallelization. Log-space uniform NC^1 and NC^1 are often used interchangeably.

The rest of the paper develops a CRR division algorithm which is in P uniform NC^1. The technique used for the division is similar to that of Lu and Chiang in that it uses a binary search to find the bit size of the quotient. The rest of the details are different, mostly because of the focus on the use of Boolean circuits for the implementation.

This algorithm was improved on in a thesis by Chiu [11], which was summarized in the paper by Chiu, Davida and Litow [12].

Chiu presents a NC^1 circuit for rank, which operates on the same principles as the binary tree based computation of $HALF(x)$ described by Davida and Litow. The rank is then used to implement a new division algorithm, which is in log-space uniform NC^1.

Chiu describes NC^1 circuits were described for base extension which are used to solve the CRR scaling problems. CRR scaling is integer division where the divisor is a product of a subset of the CRRS moduli, and the breakthrough that lead to NC^1 circuits for general integer division was to formulate the problem of integer division in terms of CRR scaling.

Another approach to division comes from Hitz and Kaltofen [33]. Their algorithm requires the use of r redundant moduli p_j, with $p_i < p_j$ for $1 \leq i \leq r$ and $r + 1 \leq j \leq 2r$. This makes the redundant part of the CRRS slightly larger than the non-redundant part.

The substantial increase in the redundancy is worthwhile, as it provides the ability to detect and correct for overflow errors resulting from multiplications. The main result of the paper is a CRR implementation of the iterative method of division due to Newton, with a proof that their algorithm converges and an upper bound for the running time. The division algorithm uses an approach similar to the algorithm by Chiu, with the division being carried out in terms of CRR scaling.

The necessary comparisons between x and y are reduced to detecting overflow in $|x - y|_P$. We have already seen very similar methods of detecting overflow using base extensions from Orton et al, and, under the guise of error detection, from Yau and Liu. The CRR scaling is carried out with base extensions, and so the base extension machinery can be reused to detect overflow.

The details of the algorithms by Chiu and by Hitz and Kaltofen are very different, but the broader strokes are very similar, with Chiu using an approximate rank with refinement to overcome the difficult CRR operations where Hitz and Kaltofen use redundancy.

4.3.3 Summary

There are four main techniques that are used, reused and reinvented by those dealing with difficult CRR operations. All of these techniques are useful in various different settings, although none are useful candidates for conversion to an online polynomial size FA that is amenable to the logical product calculation.

The first approach is to use redundant moduli, usually in conjunction with a base extension. This is of no help to us as the schemes we have seen that use redundancy cannot accurately use the logical product.

The second approach is to calculate the rank by using a sufficiently accurate binary approximation to the fractions used in the rank calculation. The number of bits needed for the approximation to be accurate is larger than the number of bits needed to represent P in binary form. Unfortunately we require a technique where all intermediate values are no bigger than $O(\log P)$

in size, as we have to encode any memory into polynomially many states.

The third approach, which is less common, is to use an inaccurate approximation to the rank and then declare a portion of the CRRS as unusable, which indirectly introduces redundancy into the system. The unusable zone causes problems with the logical product. We would need to be able to exclude integers that fall in the unusable range from the logical product, and doing this would require CRR comparison or an equivalent difficult CRR operation.

The final approach is to use an inaccurate approximation to the rank which can be refined. This requires some method of determining when the approximation is definitely accurate and a method for improving the approximation for the cases where the accuracy of the approximation is unknown.

This technique can be applied iteratively or recursively, although neither can be used for our purposes. The iterative version will result in a offline algorithm as a direct result of our definition of offline algorithms. The recursive version will not be able to be converted to a FA with polynomial size. This is a result of the facts that the rank approximation requires a FA with polynomial size, online recursion is implemented as FA intersection, and the worst case recursion depth required is polynomial in the size of the CRRS.

4.4 The inter-reductions

In Chapter 3 we defined FA for several properties of a number in a CRRS, which we will make use of in the inter-reductions. They each have $t \cdot S$ states for some small t, and since $S = O(n)$ they are polynomial in size.

Let $PR(x, k)$ be the FA which accepts x if and only if $\tilde{q}\,(x) = k$. The FA has $r \cdot S$ states, and can only be constructed for $0 \leq k < r$

Let $PP(x, k)$ be the FA which accepts x if and only if $|x|_{\tilde{2}} = k$. Clearly $k \in \{0, 1\}$. $PP(x, k)$ has $2 \cdot S$ states.

Let $GB(x, a, b)$ be the FA which accepts x if and only if $x \in GB(a, b \cdot R)$. Using the notation for GB related sets from Chapter 2, this means that x is in the set $0^a G 1^{b \cdot R} B$. $GB(x, a, b)$ has S states.

Finally, $BASE(x)$ is the FA which accepts x if and only if $x \in \mathcal{BASE}$. $BASE(x)$ has S states.

We will use these FA to define the reductions and inter-reductions between various pairs of difficult problems.

Reduction 1.

$$BAD(x) \Rightarrow HALF(x, k)$$
$$HALF(x, k) \Rightarrow BAD(x)$$

We will describe the reduction of $HALF(x, 0)$ to $BAD(x)$ in detail. While a FA for $HALF(x, 1)$

can be created as $HALF(x,1) = \overline{HALF(x,0)}$ it can also be created in a similar manner to $HALF(x,1)$, although there will be $O(1)$ edge effects introduced by the slightly different ranges of $HALF(x,0)$ and $HALF(x,1)$ which will need to be dealt with.

The FA $GB(x,a,b)$ accepts $x \in 0^a G 1^{b \cdot R} B$. The G and B parts of this set each span $r-1$ of the p_1 bins, where each bin has width R. If we had access to the FA $BAD(x)$, we could take the union of $GB(x,a,b)$ and $BAD(|x-a|_P)$ and the resulting FA would accept $x \in 0^a 1^{(b+r-1)\cdot R} B$.

The largest x accepted by $GB(x,0,0)$ will be

$$x_{max} = 2 \cdot (r-1) \cdot R$$

and

$$x_{max} \leq \lfloor P/2 \rfloor \text{ if and only if } 4 \cdot (r-1) < p_1$$

We have two cases to consider.

If $4 \cdot (r-1) < p_1$ and b is the least integer such that $(b+r-1) \cdot R > P/4$ then we can construct

$$G(x) = GB(x,0,b) \cup BAD(x)$$

and it is clear that

$$HALF(x,0) = G(x) \cup G(\lfloor P/2 \rfloor - x)$$

If $p1 < 4 \cdot (r-1)$ we construct the FA

$$H(x) = GB(x,0,0) \cup BAD(x) \cup \overline{BAD(|x-(r-1)\cdot R|_P)}$$

which accepts all $0 \leq x < 2 \cdot (r-1) \cdot R$.

The FA $H(x)$ is then used to create a FA for $HALF(x,0)$

$$HALF(x,0) = H(x) \cap H(|x + 2 \cdot (r-1) \cdot R - \lfloor P/2 \rfloor|_P)$$

For the reduction in the other direction, note that the FA $GB(x,(r-1)\cdot R,0)$ accepts the values of x that are in set $B0^*G$. It is clear that $HALF(x,0)$ can be used to isolate the set of bad integers, using

$$BAD(x) = GB\left(x,(r-1)\cdot R,0\right) \cap HALF(x,0)$$

Reduction 2.

$$BAD(x) \Rightarrow RANK(x,k)$$
$$RANK(x,k) \Rightarrow BAD(x)$$

Both of these follow from the properties of the pseudorank and the definition of bad integers. Recall that $q\left(x\right) = \tilde{q}\left(x\right)$ if $x \in \mathcal{G}$ and $q\left(x\right) = \tilde{q}\left(x\right) + 1$ if $x \in \mathcal{B}$.

This means we can get the rank by modifying the pseudorank based on whether or not x is bad.

$$RANK(x,k) = \Big(PR(x,k) \cap \overline{BAD(x)}\Big) \cup \Big(PR(x,k+1) \cap BAD(x)\Big)$$

Conversely, we can determine whether or not $x \in \mathcal{B}$ by comparing the rank and the pseudorank. Note that there are only polynomially many values to check.

$$BAD(x) = \overline{\left(\bigcup_{i=0}^{r-2} PR(x,i) \cap RANK(x,i)\right)}$$

Reduction 3.

$$BAD(x) \Rightarrow PARITY(x,k)$$
$$PARITY(x,k) \Rightarrow BAD(x)$$

Both of these follow from the properties of the pseudoparity and the definition of bad integers. Recall that $|x|_{\tilde{2}} = |x|_2$ if $x \in \mathcal{G}$ and $|x|_{\tilde{2}} \neq |x|_2$ if $x \in \mathcal{B}$.

This means we can get the parity by modifying the pseudoparity based on whether or not x is bad.

$$PARITY(x,k) = \Big(PP(x,k) \cap \overline{BAD(x)}\Big) \cup \Big(PP(x,1-k) \cap BAD(x)\Big)$$

Conversely, we can determine whether or not $x \in \mathcal{B}$ by comparing the parity and the pseudoparity.

$$BAD(x) = \Big(PP(x,0) \cap PARITY(x,0)\Big) \cup \Big(PP(x,1) \cap PARITY(x,1)\Big)$$

Reduction 4.

$$HALF(x,k) \Rightarrow COMP(x,y)$$
$$COMP(x,y) \Rightarrow HALF(x,k)$$

If x is in the first half of the CRRS and y is in the second half then x must be less than y. Similarly, if x is in the second half and y is in the first half then x must be greater than y.

If x and y are in the same half we can look at the difference between them to determine which is larger. If $x \geq y$ then $|x-y|_P$ will be in the first half of the CRRS, otherwise it will wrap into the second half of the CRRS.

This leads to the transformation from $HALF$ to $COMP$.

$$\begin{aligned} COMP(x,y) = & \Big(HALF(x,0) \cap HALF(y,1)\Big) \cup \\ & \overline{\Big(HALF(x,1) \cap HALF(y,0)\Big)} \cup \\ & HALF(|x-y|_P\,, 0) \end{aligned}$$

The reverse transformation is trivial because $HALF$ is just a comparison with a fixed value.

$$\begin{aligned} HALF(x,0) &= COMP\left(x, \left\lceil \frac{P}{2} \right\rceil\right) \\ HALF(x,1) &= COMP\left(\left\lfloor \frac{P}{2} \right\rfloor, x\right) \end{aligned}$$

Reduction 5.

$$\begin{aligned} HALF(x,k) &\Rightarrow PARITY(x,k) \\ PARITY(x,k) &\Rightarrow HALF(x,k) \end{aligned}$$

We will focus on $HALF(x,0)$, for the same reasons that we focused on $HALF(x,0)$ in the reduction of $HALF(x,k)$ to $BAD(x)$.

The reduction of $HALF(x,0)$ to $PARITY(x,k)$ begins with the observation that

$$||x|_2 + |y|_2|_2 = ||x+y|_P|_2 \text{ if and only if } x+y < P$$

Let $y = \lfloor P/2 \rfloor$ and we will see that

$$|x|_2 + |y|_2 = ||x+y|_P|_2 \text{ if and only if } 0 \leq x \leq \lfloor P/2 \rfloor$$

which gives us the reduction.

If $||P/2\rfloor|_2 = 0$

$$\begin{aligned} HALF(x,0) = & \Big(PARITY(x,0) \cap PARITY\left(|x + \lfloor P/2 \rfloor|_P\,, 0\right)\Big) \cup \\ & \Big(PARITY(x,1) \cap PARITY(|x + \lfloor P/2 \rfloor|_P\,, 1)\Big) \end{aligned}$$

otherwise

$$\begin{aligned} HALF(x,0) = & \Big(PARITY(x,0) \cap PARITY\left(|x + \lfloor P/2 \rfloor|_P\,, 1\right)\Big) \cup \\ & \Big(PARITY(x,1) \cap PARITY(|x + \lfloor P/2 \rfloor|_P\,, 0)\Big) \end{aligned}$$

The reduction in the other direction begins by looking at the even numbers in a CRRS

$$x = 2y \text{ for } 0 \leq y \leq \left\lfloor \frac{P}{2} \right\rfloor$$

and the odd numbers

$$x = 2y + 1 \text{ for } 0 \leq y < \left\lfloor \frac{P}{2} \right\rfloor$$

Note that $\frac{P+1}{2}$ will be an integer, as $|P|_2 = 1$.

If an even number is multiplied by $\frac{P+1}{2}$ we get

$$\left| 2y \cdot \frac{P+1}{2} \right|_P = y$$

and so

$$0 \leq \left| 2y \cdot \frac{P+1}{2} \right|_P \leq \left\lfloor \frac{P}{2} \right\rfloor$$

If an odd number is multiplied by $\frac{P+1}{2}$ we get

$$\left| (2y+1) \cdot \frac{P+1}{2} \right|_P = y + \frac{P+1}{2}$$

and so

$$\left\lceil \frac{P}{2} \right\rceil \leq \left| (2y+1) \cdot \frac{P+1}{2} \right|_P < P$$

This yields the reduction

$$PARITY(x, k) = HALF\left(\left| x \cdot \frac{P+1}{2} \right|_P, k \right)$$

Reduction 6.

$$BIN(x, k) \Rightarrow BAD(x)$$

The FA $GB(x, (r-1) \cdot R, 0)$ accepts the values of x that are in set $B0^*G$. The bad integers are the integers in this set that occur in the first $r-1$ bins.

$$BAD(x) = \bigcup_{i=0}^{r-2} \Big(BIN(x, i) \cap GB(x, (r-1) \cdot R, 0) \Big)$$

Reduction 7.

$$RANK(x, k) \Rightarrow BIN(x, k)$$

The $x \in \mathcal{BASE}$ are the x such that

$$\lfloor x/R \rfloor = q\left(|p_1 \cdot x|_P \right)$$

Recall that

$$q\left(|p_1 \cdot x|_P\right) = q\left(|p_1 \cdot (x + k \cdot R)|_P\right)$$

and that

$$0 \leq q\left(|p_1 \cdot x|_P\right) \leq r - 2$$

We have $|x - t \cdot R|_P \in \mathcal{BASE}$ for some t, and the value of t depends entirely on which bin x is in and on the value of $q\left(|p_1 \cdot x|_P\right)$.

This results in the reduction

$$BIN(x, k) = \bigcup_{i=0}^{r-3} RANK(|p_1 \cdot x|_P\,, i) \cap BASE(x + (k - i) \cdot R)$$

Reduction 8.

$$RANK(x, k) \Rightarrow EXTEND(x, p_{r+1}, x_{r+1})$$

For this reduction to be efficient we require that p_{r+1} is roughly the same size as the other moduli. One possible rule of thumb would be to try for $p_{r+1} < 2 \cdot p_r$.

We define an intermediate FA $F(x, p_{r+1}, x_{r+1}, i)$ which computes

$$\left|\left(\sum_{j=1}^{r} \frac{P}{p_j} \cdot \langle x \rangle_j\right) - i\right|_{p_{r+1}}$$

which will accept if and only if $\hat{\langle x \rangle}_{p_{r+1}} = x_{r+1}$ when $q\left(x\right) = i$.

The FA will be configured to act as a modulo unary adder with p_{r+1} states.

The initial state vector,

$$u_j = [j = p_{r+1} - i] \ \text{ for } 0 \leq j < p_{r+1}$$

transition matrices,

$$w(j, \sigma_{\langle x \rangle_j})_{s,t} = \left[t - s \equiv \frac{P}{p_j} \cdot \langle x \rangle_j \bmod p_{r+1}\right] \ \text{ for } 1 \leq j \leq r \text{ and } 0 \leq s, t < p_{r+1}$$

and final state vector

$$v_j\,[j = x_{r+1}] \ \text{ for } 0 \leq j < p_{r+1}$$

can all be constructed in polynomial time.

The values $\left|\frac{P}{p_j}\right|_{p_{r+1}}$ would normally be precomputed if p_{r+1} was known in advance and the base extension was going to be performed often.

$$EXTEND(x, p_{r+1}, x_{r+1}) = \bigcup_{i=0}^{r-1} \Big(F(x, p_{r+1}, x_{r+1}, i) \cup RANK(x, i)\Big)$$

Reduction 9.

$$EXTEND(x, p_{r+1}, x_{r+1}) \Rightarrow PARITY(x, k)$$

This is significantly simpler than the previous reduction.

$$PARITY(x, k) = EXTEND(x, 2, k)$$

4.5 An efficient offline algorithm for rank

4.5.1 CRR subsystems

Let $C^{(r)}$ be a CRRS with r moduli. We use $C^{(r-1)}$ to refer to the CRRS derived from $C^{(r)}$ by removing the modulus p_j. We refer to $C^{(r-1)}$ as a subsystem of $C^{(r)}$, and refer to the initial $C^{(r)}$ as the root CRRS.

The particular modulus that gets removed is not important in this section, although most uses of CRR subsystems will require that a particular modulus is specified.

For any CRR constant or function c we will use $c^{(k)}$ to indicate that the constant or function is relative to $C^{(k)}$. The exception to this is the moduli, which are referred by their indices in the root CRRS. This means that the moduli of the CRR subsystems do not get re-indexed when the modulus is removed.

We will shortly provide the details an offline algorithm for rank which makes extensive use of CRR subsystems in general and the following lemma in particular.

Lemma 4.2.

$$\left\langle |p_j \cdot x|_P \right\rangle_i^{(r)} = \langle x \rangle_i^{(r-1)} \textit{ for } 1 \leq i \leq r \ \wedge \ i \neq j$$

Proof. Recall that $\left(\frac{P^{(r)}}{p_i}\right)^{p_i - 2}$ is the modular inverse of $\frac{P^{(r)}}{p_i}$ for the modulus p_i, and so

$$\left| \frac{P^{(r)}}{p_i} \cdot \left(\frac{P^{(r)}}{p_i}\right)^{p_i-2} \right|_{p_i} = 1$$

This is the case because

$$\left| a \cdot (a^{p_i - 2}) \right|_{p_i} = 1$$

for any a co-prime with p_i and each of the moduli are pairwise co-prime.

With that in mind we see that

$$\left| \left(\frac{P^{(r)}}{p_i}\right)^{p_i-2} \right|_{p_i} = \left| \prod_{\substack{1 \leq k \leq r \\ k \neq i}} ({p_k}^{p_i - 2}) \right|_{p_i}$$

and so, for for $1 \leq i \leq r \ \wedge \ i \neq j$,

$$\begin{aligned}
\left| p_j \cdot \left(\frac{P^{(r)}}{p_i} \right)^{p_i - 2} \right|_{p_i} &= \left| p_j \cdot \prod_{\substack{1 \leq k \leq r \\ k \neq i}} \left({p_k}^{p_i - 2} \right) \right|_{p_i} \\
&= \left| p_j \cdot {p_j}^{p_i - 2} \cdot \prod_{\substack{1 \leq k \leq r \\ k \neq i, j}} \left({p_k}^{p_i - 2} \right) \right|_{p_i} \\
&= \left| \prod_{\substack{1 \leq k \leq r \\ k \neq i, j}} \left({p_k}^{p_i - 2} \right) \right|_{p_i} \\
&= \left| \left(\frac{P^{(r)}}{p_j \cdot p_i} \right)^{p_i - 2} \right|_{p_i}
\end{aligned}$$

Now consider the residues $\left\langle |p_j \cdot x|_P \right\rangle_i^{(r)}$

$$\begin{aligned}
\left\langle |p_j \cdot x|_{P^{(r)}} \right\rangle_j^{(r)} &= 0 \\
\left\langle |p_j \cdot x|_{P^{(r)}} \right\rangle_i^{(r)} &= \left| p_j \cdot x \cdot \left(\frac{P^{(r)}}{p_i} \right)^{p_i - 2} \right|_{p_i} \quad \text{for } 1 \leq i \leq r \ \wedge \ i \neq j \\
&= \left| x \cdot \left(\frac{P^{(r)}}{p_j \cdot p_i} \right)^{p_i - 2} \right|_{p_i} \quad \text{for } 1 \leq i \leq r \ \wedge \ i \neq j
\end{aligned}$$

Note that $C^{(r-1)}$ only uses makes use of the moduli p_i for $1 \leq i \leq r \ \wedge \ i \neq j$. This means $P^{(r-1)} = \left(\frac{P}{p_j} \right)^{(r)}$, and so it is clear that

$$\langle x \rangle_i^{(r-1)} = \left| x \cdot \left(\frac{P}{p_j \cdot p_i} \right)^{p_i - 2} \right|_{p_i} \quad \text{for } 1 \leq i \leq r \ \wedge \ i \neq j$$

□

A consequence of the lemma is that

Corollary 4.3.

$$q \left(p_j \cdot x \right)^{(r)} = q \left(x \right)^{(r-1)}$$

Proof. Recall that

$$q \left(x \right) = \left(\sum_{i=1}^{r} \frac{\langle x \rangle_i}{p_i} \right) - \frac{|x|_P}{P}$$

and so we are claiming that

$$\left(\sum_{i=1}^{r} \frac{{\langle |p_1 \cdot x|_{P^{(r)}} \rangle_i}^{(r)}}{p_i}\right) - \frac{|p_1 \cdot x|_{P^{(r)}}}{P^{(r)}}$$

in $C^{(r)}$ and

$$\left(\sum_{i=2}^{r} \frac{{\langle x \rangle_i}^{(r-1)}}{p_i}\right) - \frac{|x|_{P^{(r-1)}}}{P^{(r-1)}}$$

in $C^{(r-1)}$ are equivalent.

Using Lemma 4.2 and noting that ${\langle p_1 \cdot x \rangle_1}^{(r)} = 0$ we see that the components of the rank equations that depend on the residues are equivalent.

We know that

$$\begin{aligned} |p_1 \cdot x|_{P^{(r)}} &= \left| p_1 \cdot (x + k \cdot \frac{P^{(r)}}{p_1}) \right|_{P^{(r)}} \\ &= \left| p_1 \cdot |x|_{\frac{P^{(r)}}{p_1}} \right|_{P^{(r)}} \\ &= p_1 \cdot |x|_{\frac{P^{(r)}}{p_1}} \end{aligned}$$

We also know that $P^{(r-1)} = \frac{P^{(r)}}{p_1}$, so

$$\frac{|x|_{P^{(r-1)}}}{P^{(r-1)}} = \frac{p_1 \cdot |x|_{\frac{P^{(r)}}{p_1}}}{P^{(r)}}$$

This is enough to show that the remaining part of the rank equations are equivalent.

□

The new style pseudorank uses $S = p_1$, which causes $q\left(|p_1 \cdot x|_P\right)$ to appear in a number of important equations. We form subsystems by removing the modulus p_1, in order to take advantage of Corollary 4.3.

We modify the definition of S, such that $S^{(r)}$ is the smallest of the moduli in $C^{(r)}$. This allows us to define S for the ith subsystem of C^r, as

$$S^{(r-i)} = p_{1+i} \text{ for } 0 \leq i < r$$

The ith subsystem $C^{(r-i)}$ is therefore the CRRS defined by the primes p_j with $1 + i \leq j \leq r$.

4.5.2 The offline algorithm

Recall that

$$\tilde{q}\,(x) = \left\lfloor \frac{\alpha(x)}{S} \right\rfloor$$

and

$$q\,(x) = \tilde{q}\,(x) + [x \in \mathcal{B}]$$

Corollary 2.5 in Chapter 2 shows that

$$x \in \mathcal{B} \Leftrightarrow p_1 \leq |\alpha(x)|_{p_1} + q\,(p_1 \cdot x)$$

From this we get

$$q\,(x) = \left\lfloor \frac{\alpha(x) + q\,(p_1 \cdot x)}{S} \right\rfloor \tag{4.4}$$

If $x \in \mathcal{G}$ then $q\,(p_1 \cdot x)$ will not effect the floor division. If $x \in \mathcal{B}$ then the addition of $q\,(p_1 \cdot x)$ will increase the result of the floor division by 1.

We can use Corollary 4.3 to restate this as

$$q\,(x)^{(r)} = \left\lfloor \frac{\alpha(x)^{(r)} + q\,(x)^{(r-1)}}{S^{(r)}} \right\rfloor \tag{4.5}$$

This is a recursive definition, and the fact that

$$q\,(x)^{(1)} = \tilde{q}\,(x)^{(1)} = 0$$

acts as the basis of the recursion.

The iterative implementation of this equation leads to an efficient offline algorithm for calculating $q\,(x)$. The algorithm computes $q\,(x)^{(j)}$ for $2 \leq j \leq r$, using Equation 4.5 to compute $q\,(x)^{(j)}$ from $q\,(x)^{(j-1)}$.

Recall that

$$\alpha(x)^{(j)} = \sum_{i=r+1-j}^{r} \left\lfloor p_j \cdot \frac{\langle x \rangle_i^{(j)}}{p_i} \right\rfloor$$

At each step we will need $\log r$ bits of memory to store the rank from the last subsystem and at most $r + \log p_r$ bits for the calculation of $\alpha(x)$.

The algorithm solves Equation 4.5 for $r - 1$ CRR systems. Assume that all arithmetic is performed on $r + \log p_r$ bit integers. For each $2 \leq j \leq r$, the algorithm requires an addition, a floor division, and the calculation of $\alpha(x)^{(j)}$. We compute $\alpha(x)^{(j)}$ with j additions and j calculations of $\mu_i(x)^{(j)}$, each of which requires an integer multiplication, a modular multiplication and a floor division.

Let $T_A(n) = O(n)$ be the time required to perform additions of n bit numbers. Let $T_M(n)$,

$T_{MM}(n)$ and $T_F(n)$ be the time required to perform multiplication, modular multiplication and floor divisions with n bit numbers, respectively. Note that p_r is typically small enough that the asymptotically slower methods of multiplication may outperform the alternatives, and that the small size will make look up tables are a viable option for some systems.

We set $n = r + \log p_r$, and the running time of the algorithm is then

$$\frac{r^2+3r-4}{2} \cdot T_F(n) + \frac{r^2+r-2}{2} \cdot \Big(T_M(n) + T_{MM}(n)\Big) + (r-1) \cdot T_A(n)$$

or

$$O(r^2 \cdot [T_F(n) + T_M(n) + T_{MM}(n)])$$

Brönnimann, Emiris, Pan and Pion have a very similar technique which they apply to the sign detection problem. Their paper [32] presents two solutions to the sign detection problem, with a focus on the use of floating point arithmetic. These solutions use interpolation due to Lagrange and Newton respectively, and probabilistic variants of each are provided.

The method based on Lagrangian interpolation makes use of technique they refer to as the recursive relaxation of the moduli, which is equivalent to our use of CRR subsystems. It should be noted that our algorithm was derived independently of Brönnimann et al, although we did make use of their notation for the description of CRR subsystem.

Chapter 5

The distribution of the rank

5.1 The problem

During the investigation into the various properties of $\mathcal{BASE}$ detailed in Section 2.5 we became interested in the distribution of the members of $\mathcal{BASE}$ amongst the bins.

Recalling that

$$x \in \mathcal{BASE} \Leftrightarrow q\left(p_1 \cdot x\right) = \lfloor x/R \rfloor$$

we see that we can determine the exact number of $x \in \mathcal{BASE}$ that occur in the kth bin with

$$\text{bincount}(k) = \sum_{x=0}^{P-1} \left[q\left(p_1 \cdot x\right) = \lfloor x/R \rfloor = k\right]$$

This requires that the rank of $|p_1 \cdot x|_P$ and the bin in which x occurs are known, which are both difficult CRR problems.

We noticed that $q\left(p_1 \cdot x\right)^{(r)} = q\left(x\right)^{(r-1)}$ and that $\left\lfloor x/R^{(r)} \right\rfloor$ in $C^{(r)}$ is just x in $C^{(r-1)}$. This means that the distribution of the values of the rank function in a given CRRS is the same as the distribution of the members of $\mathcal{BASE}$ amongst the bins in a slightly larger CRRS. Working with $q\left(x\right)$ over the whole rank of x involves computations which are faster and easier to describe than the corresponding computations for $q\left(p_1 \cdot x\right)$ with $x \in \mathcal{BASE}$, and so we study the problem in terms of the distribution of the values of the rank function.

We use

$$\text{count}\left(k\right) = \sum_{x=0}^{P-1} \left[q\left(x\right) = k\right] \tag{5.1}$$

and

$$\text{freq}\left(k\right) = \text{count}(k)/P \tag{5.2}$$

for the number of occurrences and relative frequency of $q\left(x\right) = k$, respectively.

A slight reformulation of Equation 2.3 means that count(k) can be thought of as the number of solutions to

$$k \leq \sum_{i=1}^{r} \frac{\langle x \rangle_i}{p_i} < k+1, \text{ for } 0 \leq x < P \tag{5.3}$$

The values of $\langle x \rangle_i$ are uniformly distributed for each $1 \leq i < r$ and so, as Hung and Parhami [37] noted, we should expect that the values of the rank would be normally distributed as a result of the central limit theorem.

This would certainly be the case if we were dealing with continuous values that were centred around 0. As we are dealing with discrete values which begin at 0 we are inclined to be a little more cautious.

This is especially true as we investigate count (k). Even if we assume that the values are normally distributed, the partitioning of the values described by Equation 5.3 could introduce edge effects as there is no guarantee that the partition occurs at the point of symmetry of the distribution.

The interest in these distributions started with the study of $\mathcal{BASE}$. If the members of $\mathcal{BASE}$ were uniformly distributed amongst the bins we would know that one of the bins close to the "middle" would contain the most values.

The intent was to use this information to work out the average and worst case density of the values in the "middle" bin, which we were planning to use in the investigation of some ideas for probabilistic approaches to the difficult CRRS problems.

If the members of $\mathcal{BASE}$ were not uniformly distributed then we planned to study the irregularities in order to refine our understanding of the structural and behavioural properties of numbers in CRR.

5.2 Investigations

The first technique that was used to study of the distributions of rank values was to simply count the occurrences of each value. This was only useful for the smaller CRRS. For larger CRRS we took the rank of a large number of randomly selected integers, which allowed us to trade computation time for precision. We ensured that the sample sizes used were large enough to support our conclusions.

The frequency of the rank values for the largest CRRS for which we have precise data appear in Table 5.1. The most frequent values for the rank are around $(r-1)/2$ and the least frequent value is $r-1$, followed by 0. Aside from a bias towards the lower values, the distribution appears to be bell-shaped.

The bias might be an effect of the average value of the rank. From

$$q(x) = \left\lfloor \left(\sum_{i=1}^{r} \frac{P}{p_i} \langle x \rangle_i \right) / P \right\rfloor$$

and

$$\overline{\langle x \rangle_i} = \frac{p_i - 1}{2}$$

we get

$$\overline{q(x)} = \left\lfloor \frac{r - \sum_{i=1}^{r} \frac{1}{p_i}}{2} \right\rfloor$$

If the values of the rank do follow a bell-shaped distribution, the fact that the average value of the rank is just under $r/2$ would be enough to bias the distribution toward the lower values.

It is hard to use $\text{count}(k)$ to reason about the distribution data when the CRRS is large or when the data is generated by taking random samples. From this point on we use $\text{freq}(k)$, with the divisor in Equation 5.2 adjusted appropriately when random sampling is used. The rank frequencies will sum to 1 for the fully accurate data sets and to $1 \pm \epsilon$ for the sampled data sets. The small values of ϵ encountered in practice boosted the confidence in our selection of sample sizes.

Table 5.1 has the data for the largest CRRS that we were able to gather by brute force. There was not enough data gathered by this method for us to be very confident in our conclusions, however the data attained through random sampling was much more useful in this regard. We noted that the relative frequency of integers with rank 0 appeared to be approaching $\frac{1}{r!}$. The data in Table 5.1 is less convincing than the data gathered by random sampling, but we will soon present more convincing data.

$q(x)$	$\text{count}(q(x))$	$\text{freq}(q(x))$	$\text{freq}(q(x)) \cdot r!$
0	3164348	0.0023460792	1.6891770467
1	134810738	0.0999500284	71.9640204821
2	604848728	0.4484408918	322.8774420802
3	522111806	0.3870989109	278.7112158747
4	82808537	0.0613950769	44.2044553807
5	1037230	0.0007690127	0.5536891354

Table 5.1: Bin counts for CRRS with primes $22 \leq p_i < 44$.

We define

$$\text{co}\hat{\text{u}}\text{nt}(k) = \frac{\text{count}(k) + \text{count}(r - 1 - k)}{2}$$

and

$$\text{fr}\hat{\text{e}}\text{q}(k) = \frac{\text{freq}(k) + \text{freq}(r - 1 - k)}{2}$$

in an attempt to reduce the effect of the skew towards the lower values. Table 5.2 presents the data when are $\text{co}\hat{\text{u}}\text{nt}(k)$ and $\text{fr}\hat{\text{e}}\text{q}(k)$ are used.

As we used more and more samples to examine the behaviour of larger CRRS it became increasingly clear that the values of $\frac{\text{count}(k)}{P} \cdot r!$ were approaching some kind of well behaved integer sequence. A search of the online encyclopedia of number sequences compiled by Sloane [64] revealed that these were the Eulerian numbers.

$q(x)$	$\widehat{\text{count}}(q(x))$	$\widehat{\text{freq}}(q(x))$	$\widehat{\text{freq}}(q(x)) \cdot r!$
0	2100789	0.0015575460	1.1214330911
1	108809637	0.0806725523	58.0842376645
2	563480267	0.4177699014	300.7943289775
3	563480267	0.4177699014	300.7943289775
4	108809637	0.0806725523	58.0842376645
5	2100789	0.0015575460	1.1214330911

Table 5.2: Averaged bin counts for CRRS with primes $22 \leq p_i < 44$.

5.3 Explanations

The Eulerian numbers are defined by

$$\left\langle {n \atop m} \right\rangle = \sum_{j=0}^{m} (-1)^j \binom{n+1}{j} (m+1-j)^n$$

The combinatorial interpretation involves the number of ascents in a sequence of numbers. If the value of the ith element in the sequence a is represented by a_i, the sequence has an ascent whenever $a_i < a_{i+1}$. The Eulerian number $\left\langle {n \atop m} \right\rangle$ is the number of permutations of the integers from 1 to n which have m ascents.

Table 5.3 presents the Eulerian numbers, the normalized rank frequencies and the ratios between the two. We continue to see the bias towards the lower values of the rank.

Using the identity

$$\sum_{j=0}^{n-1} \left\langle {n \atop j} \right\rangle = n!$$

we see that

$$\frac{\left\langle {r \atop k} \right\rangle}{r!}$$

provides a normalized distribution of the Eulerian numbers.

We compare $\text{freq}(k) \cdot r!$ and $\left\langle {r \atop k} \right\rangle$ in Table 5.3 as the differences between the values would have been less readily apparent if we had used $\text{freq}(k)$ and $\frac{\left\langle {r \atop k} \right\rangle}{r!}$.

$q(x)$	$\left\langle {r \atop q(x)} \right\rangle$	$\text{freq}(q(x)) \cdot r!$	$\text{freq}(q(x)) \cdot r! / \left\langle {r \atop q(x)} \right\rangle$	$\widehat{\text{freq}}(q(x)) \cdot r!$	$\widehat{\text{freq}}(q(x)) \cdot r! / \left\langle {r \atop q(x)} \right\rangle$
0	1	1.6891770467	1.6891770467	1.1214330911	1.1214330911
1	57	71.9640204821	1.2625266751	58.0842376645	1.0190217134
2	302	322.8774420802	1.0691306029	300.7943289775	0.9960077118
3	302	278.7112158747	0.9228848208	300.7943289775	0.9960077118
4	57	44.2044553807	0.7755167611	58.0842376645	1.0190217134
5	1	0.5536891354	0.5536891354	1.1214330911	1.1214330911

Table 5.3: The Eulerian numbers and the distribution of the values of the rank function.

David and Barton [18] showed that the values of $\left\langle {n \atop m} \right\rangle$ for $0 \leq m < n$ approach a normal distribution as n grows large. This means that if we can show that the distribution of the values of rank approaches the distribution of the Eulerian numbers as the size of the CRRS increases we will have also demonstrated that the values of the rank are normally distributed. We could use this information and Equation 5.3 to approximate $\text{count}\,(k)$.

Bradley and Gupta [9] describe the distribution of the sum of uniform random variables having different distributions for both the continuous and discrete cases. This appears to be enough to give use the exact rank counts, however the running time for their calculation is exponential in the problem size and so it is not much better than the brute force counting that we started with.

There is a geometric interpretation of the Eulerian numbers that might be applicable to the rank distribution and the efficient computation of $\text{count}\,(k)$.

Consider an r-dimensional unit hypercube.

A point $y_1, y_2, \ldots, y_{r-1}, y_r$ is on the hyperplane H_k if

$$\sum_{i=1}^{r} y_i = k$$

Each hyperplane is perpendicular to the main diagonal of the hypercube and H_k will pass through exactly $\binom{r}{k}$ of the corners of the hypercube.

Every $0 \leq x < P$ can be specified as a point on the hypercube, where $\frac{\langle x \rangle_i}{p_i}$ specifies the coordinate of the point in the ith dimension. From Equation 5.3 we can see that the points corresponding to the x for which $q\,(x) = k$ will all lie between H_k and H_{k+1}. Some of these points might lie on H_k but none can lie on H_{k+1}. We will refer to the volume containing these points as the kth simplex of the hypercube.

Let the lattice for the hypercube by the points where the coordinate in the ith dimension are constrained to the values $\frac{a_i}{p_i}$ where $0 \leq a_i < p_i$. If we would like to work with a lattice of integral values, we can scale each dimension of the hypercube by a factor of P Regardless of the lattice we use, the density of the lattice points in each of the simplices will be proportional to the volume of the simplex.

Schmidt and Simion [24] showed that the volume of the kth simplex is $\frac{\left\langle {r \atop k} \right\rangle}{r!}$. This is encouraging but not necessarily helpful. Determining which simplex a point is in becomes difficult for the points near the partitioning hyperplanes, and is made more difficult by the atypical lattice we are using.

The edge effects on integer points near simplex edge seems to be open to methods from the large polyhedral combinatorics literature, examples of which are the book by Beck and Robins [5] and papers by Postnikov [58] and Brion and Vergne [10]. These methods lead to a new area of CRRS research that lies beyond the scope of the work described here.

5.4 Estimations

Based on my experience with the rank function I do not think a computationally feasible rank census exists. This lead me to seek an efficient approximation scheme.

We can obtain a lower bound on the number of integers for which $q\left(x\right) = k$ with the logical product of the CFA $LOWER$. The definition of $LOWER$ is

$$LOWER(x, k) = PR(x, k) \cap GB(x, 0, p_1 - 2 \cdot (r - 1))$$

where PR and GB are the FA which were defined in Section 4.4

Assuming that $p_1 > 2 \cdot (r - 1)$. The FA $GB(x, 0, p_1 - 2 \cdot (r - 1))$ accepts x in the largest range that we can efficiently construct which has $q\left(x\right) = \tilde{q}\left(x\right)$. The logical product of the CFA $LOWER$ will give us the number of values of x in this range which have $\tilde{q}\left(x\right) = k$. At most the result will be equal to the number of $0 \leq x < P$ which have $q\left(x\right) = k$, although the values of x implicitly excluded by $GB(x, 0, p_1 - 2 \cdot (r - 1))$ will most likely decrease this value.

We can obtain the upper bound on the number of integers for which $q\left(x\right) = k$ in a similar manner. The upper bound is the result of the logical product of the CFA $UPPER$, defined as

$$UPPER(x, k) = PR(x, k) \cup \left(PR(x, k+1) \cap \overline{GB\left(x, 0, p_1 - 2 \cdot (r - 1)\right)}\right)$$

The FA $PR(x, k)$ accepts $x \in \mathcal{G} \wedge \left(\tilde{q}\left(x\right) = q\left(x\right) = k\right)$ and $x \in \mathcal{B} \wedge \left(\tilde{q}\left(x\right) = q\left(x\right) - 1 = k\right)$. We use $PR(x, k)$ as a component of $UPPER$ in order to count the number of $x \in \mathcal{G}$ with $q\left(x\right) = k$. The results for the $x \in \mathcal{B}$ are one of the sources of the overestimation we see in the results of $UPPER$.

We again assume that $p_1 > 2 \cdot (r - 1)$. The FA $\overline{GB\left(x, 0, p_1 - 2 \cdot (r - 1)\right)}$ accepts all $x \in \mathcal{B}$ and some $x \in G$ in the higher end of the range $0 \leq x < P$. We use this to find the values in $x \in \mathcal{B}$ with $q\left(x\right) = k$, which $PR(x, k)$ was unable to provide. The FA $PR(x, k+1)$ is used to isolate the $x \in \mathcal{B}$ with $q\left(x\right) = k$ and takes advantage of the fact that $q\left(x\right) = \tilde{q}\left(x\right) + 1$ when $x \in \mathcal{B}$. In this case the $x \in G$ contribute to the overestimation present in $UPPER$. When combined with the results of $PR(x, k)$ we can guarantee that the logical product of $UPPER$ will count the number of $0 \leq x < P$ with $q\left(x\right) = k$.

The lower and upper bound rank distributions are displayed in Figures 5.1, 5.2 and 5.3, along with a rank distribution created from the mean of the lower and upper bound values. There appears to be monotonic increase in the rank distribution from the lowest and highest values towards the middle, however it is still unclear if the distribution is Gaussian in nature. As the size of the CRRS increases the peak value of the relative frequencies decreases.

Figures 5.4, 5.5 and 5.6 shows the ratio between the rank data and the Eulerian numbers for the lower and upper bounds and the mean of the two bounds.

The key features are the outlying ratios at either end of the range and the and the descending slope over the well behaved range from ratios just over 1 to ratios just under 1. The outliers

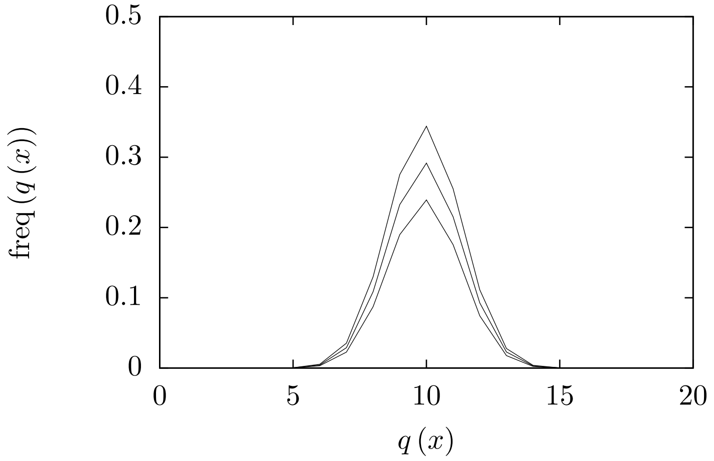


Figure 5.1: Relative frequencies of rank values, $100 \leq p_i < 200$.

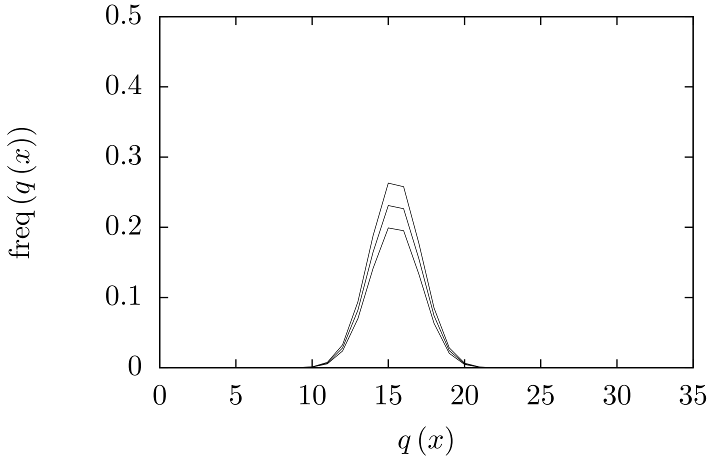


Figure 5.2: Relative frequencies of rank values, $200 \leq p_i < 400$.

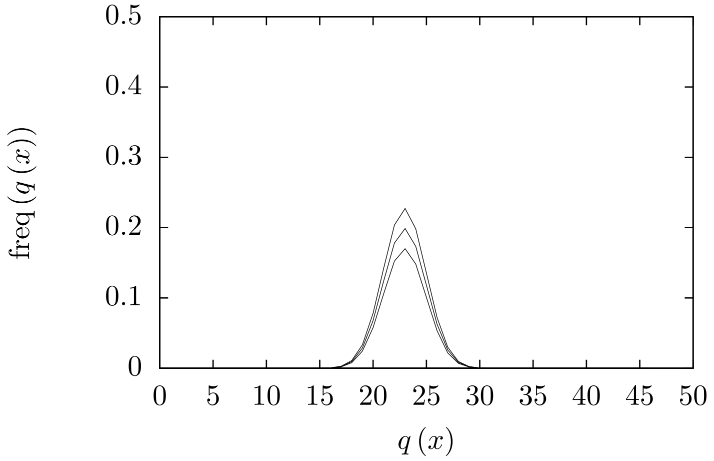


Figure 5.3: Relative frequencies of rank values, $300 \leq p_i < 600$.

are most likely a result of the use of the sets $G1^*B$ and $B0^*G$ in the estimations, and the floor division may be the cause of the bias towards the lower ranks.

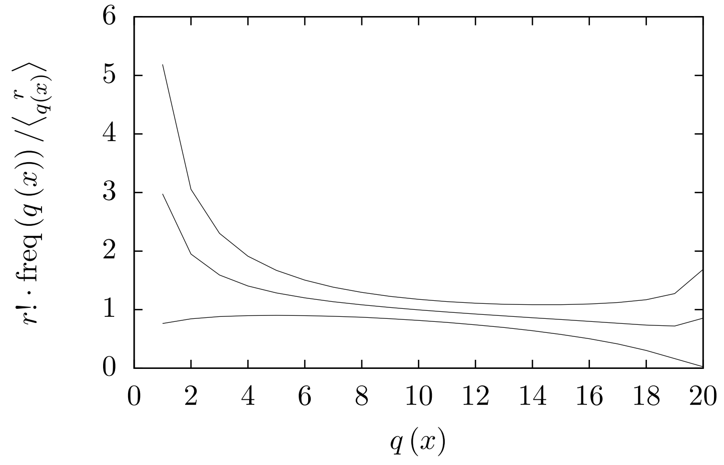


Figure 5.4: Errors relative to the Eulerian numbers, $100 \leq p_i < 200$.

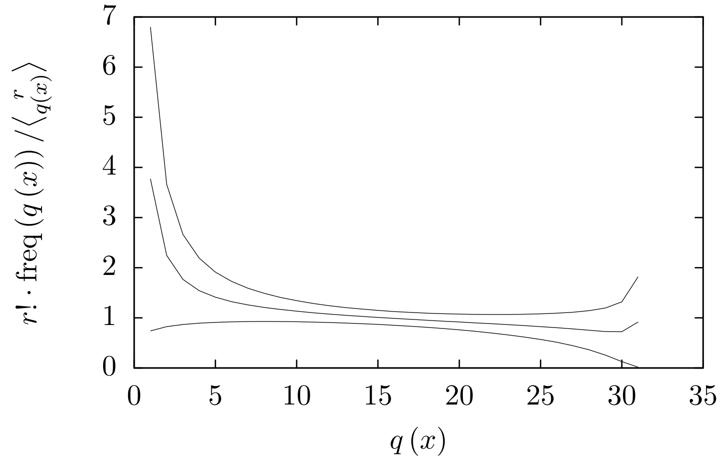


Figure 5.5: Errors relative to the Eulerian numbers, $200 \leq p_i < 400$.

Figure 5.7 and Figure 5.8 display the mean distributions and the ratios between the mean rank data and the Eulerian numbers for several of CRRS in the same plot. These figures are included to that the trends that we identified in the previous figures hold for more than 3 different CRRS. The largest CRRS has smallest peak frequency in Figure 5.7 and the largest error in Figure 5.8, where the smallest CRRS has the largest peak frequency and the smallest error. The different lines in these figures correspond to the different CRRS which are used, and the peak frequency and largest error change monotonically between these two extremes.

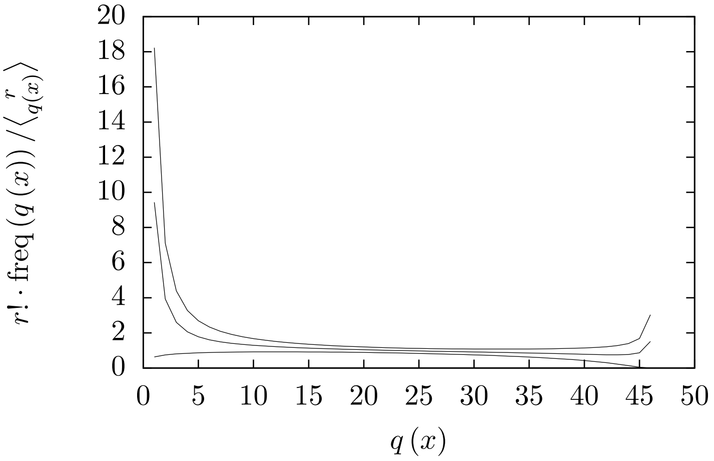


Figure 5.6: Errors relative to the Eulerian numbers, $300 \leq p_i < 600$.

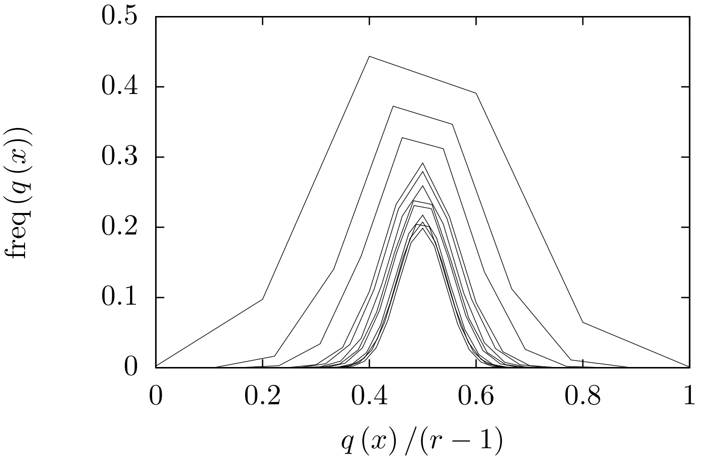


Figure 5.7: Relative frequencies of rank values, $25 \cdot k \leq p_i < 50 \cdot k$, $0 < k \leq 12$.

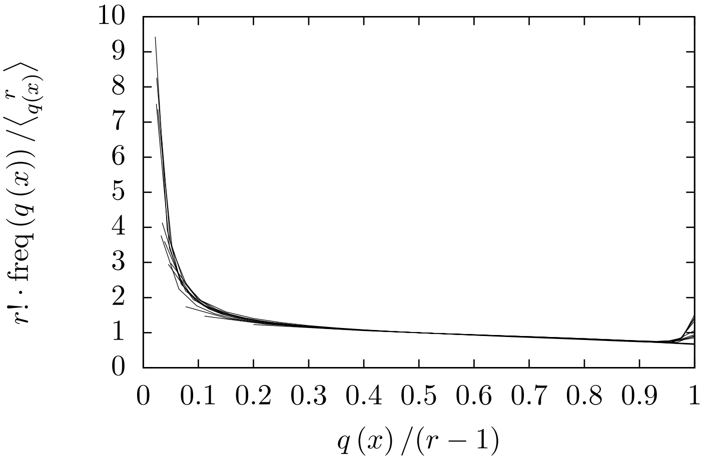


Figure 5.8: Errors relative to the Eulerian numbers, $25 \cdot k \leq p_i < 50 \cdot k$, $0 < k \leq 12$.

Chapter 6

Conclusion

6.1 Summary of results

The census algorithms described in Chapter 2 gave us the ability to determine the number of good and bad integers in a given CRRS. As far as we are aware this is the first effort that yielded such detailed information, mostly because our techniques are able to deal with the very large numbers that occur in these calculations in an efficient manner.

The refinement of the pseudorank and subsequent study of the set $\mathcal{BASE}$ led to a deepened understanding of how and when the pseudorank fails to match the rank.

In Chapter 3 we gave theoretical and practical details on the use of various FA in a CRRS setting. In addition to defining efficient FA for several basic CRRS properties, several options relating to the data structures used in implementations were described and analysed with respect to their effect on the time and memory required by common operations.

In Chapter 4 we demonstrated that a group of CRR problems long held to be difficult are in fact closely related to the NP-complete problems. The new pseudorank was used in order to devise an algorithm which computes the rank of an integer in an efficient but FA-incompatible manner.

The distribution of the values of the rank was investigated and a plausible hypothesis was formed about the behaviour of this distribution in the limiting case.

6.2 Future work

The most obvious goal for future efforts would be to build on the work done in Chapter 5. What we presented was a possible link between a pattern in empirical data and a theoretical explanation, and there is plenty of scope for the link to be explored. The polytope literature and early empirical analyses suggest that the midsection of $\mathcal{BASE}$ is "locally dense", by which we mean that the distance between consecutive members of the set $\mathcal{BASE}$ appears to be bounded by at most n^2.

As an additional area of research, there may also be a relationship between the moduli of a CRRS and the intermediate values which appear during the offline rank algorithm. The moduli need only be odd and mutually co-prime, and a set of moduli may exist such that the offline rank algorithm could be converted to an online rank algorithm with an acceptably small increase in the number of states. Proving whether or not such a set of moduli exist would be a difficult task, however any techniques developed for the study of the problem would most likely be fascinating in their own right.

Bibliography

[1] The GNU multiple precision arithmetic library, 2009. http://gmplib.org.

[2] *The GNU Multiple Precision Arithmetic Library Manual*, 2009. http://gmplib.org/gmp-man-4.3.0.pdf.

[3] I. J. Akushskii, V. M. Burcev, and I. T. Pak. A New Positional Characteristic of Non-Positional Codes and its Applications. In V. M. Amerbaev, editor, *Coding Theory and the Optimization of Complex Systems*. SSR, Alm-Ata: 'Nauka' Kazah, 1977.

[4] F. Barsi and P. Maestrini. Error detection and correction by product codes in residue number systems. *IEEE Transactions on Computers*, C-23(9):915–924, September 1974.

[5] M. Beck and S. Robins. *Computing the continuous discretely. Integer-point enumeration in polyhedra.* Undergraduate Texts in Mathematics. Springer, 2007.

[6] D. J. Bernstein. Multidigit multiplication for mathematicians. http://cr.yp.to/papers/m3.ps, 2001.

[7] J. Berstel and C. Reutenauer. *Noncommutative Rational Series With Applications.* http://www-igm.univ-mlv.fr/~berstel/LivreSeries/LivreSeries.html.

[8] J. Berstel and C. Reutenauer. *Rational series and their languages.* Springer-Verlag, New York, NY, USA, 1988.

[9] D. Bradley and R. Gupta. On the distribution of the sum of n non-identically distributed uniform random variables. *Annals of the Institute of Statistical Mathematics*, 54(3):689–700, September 2002.

[10] M Brion and M. Vergne. Lattice points in simple polytopes. *Journal of the American Mathematical Society*, 10:371–392, 1997.

[11] A. Chiu. Complexity of parallel arithmetic using the chinese remainder representation. Master's thesis, U. Wisconsin-Milwaukee, 1995. G. Davida, supervisor.

[12] A. Chiu, G. Davida, and B. Litow. Division in logspace-uniform NC^1. *Theoretical Informatics and Applications*, 35:259–275, 2001.

[13] A. Church. A note on the Entscheidungsproblem. *Journal of Symbolic Logic*, 1(1):40–41, 1936.

[14] A. Cobham. The intrinsic computational difficulty of functions. In Y. Bar-Hillel, editor, *Proceedings of the 1964 International Conference for Logic, Methodology, and Philosophy of Science*, pages 24–30. North-Holland, 1965.

[15] S. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on theory of computing*, pages 151–158, 1971.

[16] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.

[17] K. Culik II and J. Kari. Image compression using weighted finite automata. *Computer and Graphics*, (17):305–313, 1993.

[18] F. N. David and D.E. Barton. *Combinatorial chance.* Charles Griffen and Co., London, 1962.

[19] G. Davida and B. Litow. Fast parallel arithmetic via modular representation. *SIAM Journal of Computing*, 20(4):756–765, 1991.

[20] Impedovo S. Dimauro, G. and G. Pirlo. A new technique for fast number comparison in the residue number system. *IEEE Transactions on Computers*, 42(5):608–612, May 1993.

[21] P. Dusart. The kth prime is greater than $k(\ln k + \ln\ln k - 1)$ for $k \geq 2$. *Mathematics of Computation*, 68(225):411–415, 1999.

[22] J. Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17:449–467, 1965.

[23] J. Eisner. Simpler and more general minimization for weighted finite-state automata. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology*, pages 64–71, 2003.

[24] F. W. Frank W. Schmidt and R Simion. Some geometric probability problems involving the eulerian numbers. *The Electronic Journal of Combinatorics*, 4(2), 1997.

[25] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.

[26] P. Gaudry, A. Kruppa, and P. Zimmermann. A GMP-based implementation of Schönhage-Strassen's large integer multiplication algorithm. In *ISSAC '07: Proceedings of the 2007 international symposium on Symbolic and algebraic computation*, pages 167–174, 2007.

[27] J. Groβchadl. The chinese remainder theorem and its application in a high-speed RSA crypto chip. In *ACSAC '00: Proceedings of the 16th Annual Computer Security Applications Conference*, page 384, 2000.

[28] V. Halava and T. Harju. Languages accepted by integer weighted finite automata. Technical report, 1998.

[29] V. Halava and T. Harju. Undecidability in integer weighted finite automata. *Fundamenta Informaticae*, 38(1–2):189–200, April 1999.

[30] G. H. Hardy and E. M. Wright. *An introduction to the theory of numbers*. Clarendon Press, Oxford, third edition, 1954.

[31] J. Hee. Fast convolution using polynomial transforms. `http://jenshee.dk/signalprocessing/polytrans.pdf`, 2004.

[32] H. Herv Brnnimann, I. Z. Emiris, V. Y. Pan, and S. Pion. Sign determination in residue number systems. *Theoretical Computer Science*, 210:173–197, 1999.

[33] M. A. Hitz and E. Kaltofen. Integer division in residue number systems. *IEEE Transactions on Computers*, 44(8):983–989, August 1995.

[34] J. Hopcroft, R. Motwani, and J. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley Publishing Company, second edition, 2001.

[35] J. E. Hopcroft. An n log n algorithm for minimizing the states in a finite automaton. In Z. Kohavi, editor, *The Theory of Machines and Computations*, pages 189–196. Academic Press, 1971.

[36] D. A. Huffman. The synthesis of sequential switching circuits. *Journal of the Franklin Institute*, 257:161–190 and 275–303, March and April 1954.

[37] C. Y. Hung and B. Parhami. Error analysis of approximate chinese remainder theorem decoding. *IEEE Transactions on Computers*, 44(11):1344–1348, November 1995.

[38] A. Karatsuba and Y. Ofman. Multiplication of multidigit numbers on automata. *Soviet Physics Doklady*, 7:595–596, 1963.

[39] R. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Communications*, pages 85–103. Plenum Press, 1972.

[40] S. E. Kleene. *Representations of events in nerve nets and finite automata*, pages 3–42. Princeton University Press, 1956.

[41] W. Kuich and A. Salomaa. *Semirings, automata, languages*. Springer-Verlag, London, UK, 1986.

[42] D. Laing and B. Litow. Fast evaluation of iterated multiplication of very large polynomials: An application to chinese remainder theory. In Larson J. W. Read, W. and A. J. Roberts, editors, *Proceedings of the 13th Biennial Computational Techniques and Applications Conference, CTAC-2006*, volume 48 of *ANZIAM Journal*, pages C709–C724, Dec 2007.

[43] D. Laing and B. Litow. Census algorithms for chinese remainder pseudorank. *RAIRO Theoretical Informatics and Applications*, 42(2):309–322, Apr 2008.

[44] L. A. Levin. Universal search problems. *Problemy Peredaci Informacii 9*, pages 115–116, 1973 (in Russian). English Translation in *Problems of Information Transmission 9*, pages 265–266, 1973.

[45] M. Lu and J-S. Chiang. A novel division algorithm for the residue number system. *IEEE Transactions on Computers*, 41(8):1026–1032, August 1992.

[46] D. Mandelbaum. Error correction in residue arithmetic. *IEEE Transactions on Computers*, C-21(6):538–545, June 1972.

[47] K. Manders and L. Adleman. NP-complete decision problems for binary quadratics. *Journal of Computer and System Sciences*, 16:168–184, 1978.

[48] J. B. Martens. Polynomial products by means of generalized number theoretic transforms. *IEEE Transactions on Acoustics, Speech and Signal Processing*, ASSP-32(3):668–670, June 1984.

[49] G. Martinelli. Long convolutions using number theoretic and polynomial transforms. *IEEE Transactions on Acoustics, Speech and Signal Processing*, ASSP-32(5):1090–1092, October 1984.

[50] G. H. Mealy. A method for synthesizing sequential circuits. *Bell Systems Technical Journal*, 34:1045–1079, September 1955.

[51] A. R. Meyer and L. J. Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential space. In *Proceedings of the 13th Annual Symposium on the Foundations of Computer Science*, pages 125–129, 1972.

[52] M. Mohri. Finite-state transducers in language and speech processing. *Computational Linguistics*, 23:269–311, 1997.

[53] M. Mohri. Minimization algorithms for sequential transducers. *Theoretical Computer Science*, 234(1-2):177–201, 2000.

[54] E. F. Moore. Gedanken experiments on sequential machines. In *Automata Studies*, pages 129–153. Princeton University Press, 1956.

[55] H. J. Nussbaumer. *Fast Fourier Transform and Convolution Algorithms.* Springer, second edition, 1982.

[56] G.A. Orton, L.E. Peppard, and S.E. Tavares. New fault tolerant residue number systems. *IEEE Transactions on Computers*, 41(11):1453–1462, November 1992.

[57] S. C. Pei and J. L. Wu. Improved long convolutions using generalized number theoretic and polynomial transforms. *IEEE Transactions on Acoustics, Speech and Signal Processing*, ASSP-33(6):1626–1627, December 1985.

[58] A. Postnikov. Permutohedra, associahedra, and beyond. *International Mathematics Research Notices*, 2009(6):1026–1106, 2009.

[59] M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, (3):114–125, 1959.

[60] I. S. Reed, T. K. Truong, C. S. Yeh, and H. M. Shao. An improved FPT algorithm for computing two-dimensional cyclic convolutions. *IEEE Transactions on Acoustics, Speech and Signal Processing*, ASSP-31(4):1048–1050, August 1983.

[61] M. P. Schtzenberger. On the definition of a family of automata. *Information and Control*, 4:245–270, 1961.

[62] A.P. Shenoy and R. Kumarersan. A novel division algorithm for the residue number system. *IEEE Transactions on Computers*, 38(2):292–297, February 1989.

[63] M. Sipser. *Introduction to the Theory of Computation.* PWS Publishing Company, second edition, 2006.

[64] N. J. A Sloane. The on-line encyclopedia of integer sequences. `http://www.research.att.com/~njas/sequences`.

[65] N. Szabo. Sign detection in nonredundant residues systems. *IRE Transactions on Electronic Computers*, EC-11:494–500, August 1962.

[66] R. Tanaka and N. Szabo. *Residue Arithmetic and its Application to Computer Technology.* McGraw-Hill, 1968.

[67] T. K. Truong, I. S. Reed, R Lipes, and C. Wu. On the application of a fast polynomial transform and the chinese remainder theorem to compute a two-dimensional convolution. *IEEE Transactions on Acoustics, Speech and Signal Processing*, ASSP-29(1):91–99, February 1981.

[68] A. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, (42):230–265, 1936.

[69] T. V Vu. Efficient implementations of the chinese remainder theorem for sign detection and residue decoding. *IEEE Transactions on Computing*, 34(7):646–651, 1985.

[70] B. W. Watson. A taxonomy of finite automata minimization algorithms. Computing Science Note 93/44, Eindhoven University of Technology, The Netherlands, 1993.

[71] S. S-S. Yau and Y-C. Liu. Error correction in redundant residue number systems. *IEEE Transactions on Computers*, C-22(1):5–11, January 1973.

[72] Kim Yen, S-M., S. S. Lim, and S-J. Moon. RSA speedup with chinese remainder theorem immune against hardware fault cryptanalysis. *IEEE Transactions on Computers*, 52(4), April 2003.

Appendix A

Polynomial multiplication

A.1 Notation

A.1.1 Polynomials

Let $P(z)$ be a polynomial in the indefinite variable z. We used $|P(z)|$ to represent the degree of $P(z)$, and we use $P(z)[i]$ to denote the coefficient of the ith term of $P(z)$, and so we have

$$\sum_{i=0}^{|P(z)|} P(z)[i] \cdot z^i$$

We will have occasion to represent various integers as polynomials. For an integer x in base t, we create the $\lfloor \log_t x \rfloor$ degree polynomial

$$P_x(z) = \sum_{i=0}^{\lfloor \log_t x \rfloor} \left| x/t^i \right|_t \cdot z^i$$

and note that evaluating $P_x(t) = x$.

We can interpret $Q(z)$ as the sequence of numbers q with length $|q| = |Q(z)| + 1$ where the ith element of the sequence $q_i = Q(z)[i]$ for $0 \leq i \leq |q|$. We say that the sequence q is zero padded to d terms for some $|q| < d$ if we extend q to length d with $q_i = 0$ for $|q| < i < d$.

A.1.2 Convolutions

We are about to describe a number of operations known as convolutions. For all of the convolutions that we will describe the operand sequences a and b will be zero-padded to length $n = max(|a|, |b|)$. We treat $a_i = 0$ for $i < 0$ and $|a| < i$.

The acyclic convolution of the sequences a and b is defined as

$$c_h = \sum_{k=0}^{n-1} a_k \cdot b_{h-k} \text{ for } h = 0, \ldots, 2n-2$$

where $|c| = 2n - 1$.

This is the most usual form of convolution, and in situations where it would not introduce ambiguity the terms acyclic convolution and convolution can be used interchangeably. Acyclic convolution is equivalent to polynomial multiplication if the sequences are taken to represent polynomials, and so we have $C(z) = A(z) \cdot B(z)$.

The cyclic convolution is defined as

$$c_h = \sum_{k=0}^{h} a_k \cdot b_{h-k} + \sum_{k=h+1}^{n-1} a_k \cdot b_{n+h-k} \text{ for } h = 0, \ldots, n-1$$

and $|c| = n$.

In polynomial form this is expressed as

$$C(z) = A(z) \cdot B(z) \bmod z^n - 1$$

Note that if $\acute{C}(z) = A(z) \cdot B(z)$

$$C(z)[i] = \sum_{k=0}^{\lfloor|\acute{C}(z)|/n\rfloor} \acute{C}(z)[i + kn]$$

The negacyclic convolution is similar to the cyclic convolution

$$c_h = \sum_{k=0}^{h} a_k \cdot b_{h-k} - \sum_{k=h+1}^{n-1} a_j \cdot b_{n+h-k} \text{ for } h = 0, \ldots, n-1$$

and has a similar description in polynomial form

$$C(z) = A(z) \cdot B(z) \bmod z^n + 1$$

Using $\acute{C}(z) = A(z) \cdot B(z)$ we have

$$C(z)[i] = \sum_{k=0}^{\lfloor|\acute{C}(z)|/n\rfloor} -1^k \acute{C}(z)[i + kn]$$

If we zero pad the operand sequences of either type of cyclic convolution to length $2n$ then the operation is equivalent to an acyclic convolution, as the extra space prevents the modular "wrapping" that is characteristic of the cyclic convolutions from occurring.

A.2 Theory

A.2.1 The convolution theorem and the fast Fourier transform

The naive algorithm that performs acyclic convolutions on sequences of length n runs in time $O(n^2)$. There are several algorithms that reduce this to $O(n^k)$ for some $1 < k < 1.5$, the most common being the algorithms due to Karatsuba [38] and Toom and Cook.

We can do better than this polynomial time bound by using the convolution theorem. The convolution theorem states that the discrete Fourier transform (DFT) of the circular convolution of two sequences is equal to the point by point multiplication of the DFTs of the sequences. Formally, with $DFT(x) = X$ transforming the sequence x to the sequence X and $*$ representing a point by point multiplication of two sequences, we have

$$DFT(a \cdot b) = DFT(a) * DFT(b)$$

The DFT is invertible and using $IDFT(X) = x$ to denote the inverse DFT (IDFT) we can restate the convolution theorem as

$$a \cdot b = IDFT(DFT(a) * DFT(b))$$

Although the DFT is used in many fields it can be most easily be explained by describing the role of the DFT in signal processing. The Fourier transform (FT) converts a continuous function describing the value of a signal in time to a continuous function describing the frequency components of the signal.

The DFT is the equivalent for discrete sequences. The input sequence x represents n samples taken from the input sequence at some frequency f, and the output sequence X represents the frequency components of the sampled signal with the kth sample in the output signal is the spectral component with frequency $k \cdot f$. It is usual to define and describe the DFT for use with sequences of complex numbers.

The DFT of the sequence with length n is defined as

$$X_h = \sum_{k=0}^{n-1} x_k \cdot e^{\frac{2\pi i}{n} hk} \text{ for } h = 0, \ldots, n-1 \tag{A.1}$$

and the IDFT is similarly defined as

$$x_k = \frac{1}{n} \sum_{h=0}^{n-1} X_h \cdot e^{\frac{-2\pi i}{n} hk} \text{ for } k = 0, \ldots, n-1$$

The DFT in the complex numbers uses $e^{\frac{2\pi i}{n}}$ as a primitive nth root of unity. An nth root of unity is any number z such that $z^n = 1$. Note that an nth root of unity has period n, and so we normally see roots of unity defined for use in some algebraic ring. If $z^k \neq 1$ for all $1 \leq k < n$

then z is a primitive root of unity, and the following holds:

$$\sum_{k=0}^{n-1} z^k = 0 \text{ for } n > 1$$

Both the DFT and the convolution theorem can be used in other rings by changing the DFT to use an appropriate primitive nth root of unity for the ring. We will describe such a case shortly.

A naive implementation of the DFT for n-length sequences will require $O(n^2)$ complex multiplications and additions, and so implementing polynomial multiplication via the convolution theorem will yield an algorithm with the same asymptotic running time as the naive implementation of long multiplication.

The alternative is to use a fast Fourier transform (FFT), a term used to describe any DFT implementation that only requires the use of $O(n \log n)$ operations. FFT methods are of great interest in diverse areas of research and so there are a wealth of methods and a huge body of literature. A good survey of several FFT methods and their variants is provided by Nussbaumer [55].

As an example of an FFT we describe the radix-2 Cooley-Tukey algorithm, which performs the DFT on n-length sequences for $n = 2^t$ by recursively decomposition [16]. We decompose the time sequence, so this is a decimation in time (DIT) FFT, the alternative being decimation in frequency (DIF). While both methods of decomposition are formally equivalent each presents an implementer with the possibility of applying different optimizations.

We begin with the DFT from Equation A.1 and then split the sum of the time sequence elements into a sum for the elements at even positions and a sum for the elements at odd positions.

For $h = 0, \ldots, n-1$ we have

$$\begin{aligned} X_h &= \sum_{k=0}^{n/2-1} x_{2k} \cdot e^{\frac{2\pi i}{n} h(2k)} + \sum_{k=0}^{n/2-1} x_{2k+1} \cdot e^{\frac{2\pi i}{n} h(2k+1)} \\ &= \sum_{k=0}^{n/2-1} x_{2k} \cdot e^{\frac{2\pi i}{n/2} hk} + e^{\frac{2\pi i}{n} h} \sum_{k=0}^{n/2-1} x_{2k+1} \cdot e^{\frac{2\pi i}{n/2} hk} \end{aligned}$$

This rearranges the DFT to use primitive $n/2$ roots of unity, and since they have period $n/2$ we have

$$\sum_{k=0}^{n/2-1} x_{2k} \cdot e^{\frac{2\pi i}{n/2} hk} = \sum_{k=0}^{n/2-1} x_{2k} \cdot e^{\frac{2\pi i}{n/2} h(k+n/2)} = \sum_{k=0}^{n/2-1} x_{2k} \cdot e^{\frac{2\pi i}{n/2} (h+n/2)k} \text{ and}$$

$$\sum_{k=0}^{n/2-1} x_{2k+1} \cdot e^{\frac{2\pi i}{n/2} hk} = \sum_{k=0}^{n/2-1} x_{2k+1} \cdot e^{\frac{2\pi i}{n/2} h(k+n/2)} = \sum_{k=0}^{n/2-1} x_{2k+1} \cdot e^{\frac{2\pi i}{n/2} (h+n/2)k}$$

and so if we set

$$DFT(even) = \sum_{k=0}^{n/2-1} x_{2k} \cdot e^{\frac{2\pi i}{n/2}hk}$$
$$DFT(odd) = \sum_{k=0}^{n/2-1} x_{2k+1} \cdot e^{\frac{2\pi i}{n/2}hk}$$

we get

$$\begin{aligned} X_h &= DFT(even) + e^{\frac{2\pi i}{n}h} DFT(odd) \text{ for } h = 0, \cdots, n/2-1 \\ &= DFT(even) - e^{\frac{2\pi i}{n}h-n/2} DFT(odd) \text{ for } h = n/2, \cdots, n-1 \end{aligned}$$

From this we see that the n-length DFT performs n complex multiplications and additions and 2 $n/2$-lengths DFTs. This is a recursive algorithm with depth $k = O(\log n)$ where the kth level of recursion performs 2^k length $n/2^k$ DFTs. As this requires n complex multiplications and additions per level of recursion we see that the algorithm requires $\Theta(n \log n)$ complex operations overall.

A.2.2 The Schönhage-Strassen algorithm

Using values of the complex exponential function as the primitive roots of unity has some drawbacks when implementing an FFT based algorithm on modern hardware, especially when dealing with convolutions of long sequences. Using fixed precision approximations to the complex exponential terms can introduce errors that require great care to deal with.

Arbitrary precision arithmetic can reduce this problem at the expense of memory usage, although some imprecision will remain. This can introduce further problems. The convolution theorem cancels terms using the fact that $\sum_{k=0}^{n-1} z^k = 0$ when z is a primitive nth root of unity, and with arbitrary precision arithmetic we instead get large numerical approximations to 0 that can increase the memory usage of the algorithm dramatically.

The Schönhage-Strassen algorithm is a recursive integer multiplication algorithm that avoids these issues by using the convolution theorem in the ring of integers modulo $2^n + 1$ for $n = 2^t$. This gives primitive roots of unity with the form 2^m at each level of recursion, and so the algorithm is exact for computation using binary arithmetic. Most processors provide “shift” instructions which multiply by 2^m and are computationally cheaper than the equivalent multiplication instruction.

The convolution theorem as we have described it is correct for cyclic convolutions, and as the Schönhage-Strassen algorithm uses negacyclic convolutions we need to provide an addendum to the theorem. If we have the primitive $2n$ unity z such that $z^{2n} = 1$, define the sequences s and $\hat{s}$ such that

$$s_i = z^i \text{ for } i = 0, \ldots, n-1 \qquad \bar{s}_i = z^{-i} for i = 0, \ldots, n-1$$

The convolution theorem, corrected for the use of the negacyclic convolution, becomes

$$DFT(\bar{s} * (a \cdot b)) = DFT(s * a) * DFT(s * b)$$

where $a \cdot b$ represents negacylic convolution and $a * b$ represents point by point multiplication of sequences. We refer to the $s * a$ steps as pre-scaling and $\hat{s} * a$ step as post-scaling.

The algorithm begins by dividing the n-bits of each integer operand into 2^k groups, where k is typically a small fixed integer such that 2^k divides n, and treats these groups as the sequences a and b. We then compute the length $\acute{n}$ negacyclic convolution of a and b where $\acute{n}$ is the least multiple of 2^k that will not overflow during the convolution. The pre-scaling by s, the DFTs, the IDFT and the post-scaling is carried out with shifts and adds, using $2^{2\acute{n}/2^k}$ as a 2^kth root of unity. The point by point multiplications are negacylic convolutions of the $n/2^k$-bit elements of the transformed sequences, and this is where the algorithm becomes recursive. The recursion and the extensive use of shifts gives a running time of $O(n \log n \log \log n)$.

The algorithm only outperforms the asymptotically slower multiplication algorithms for very large numbers, and so it is common practice to start using alternative methods of multiplication beyond a certain depth in the recursion. The GMP multiplication function uses the size of the inputs to guess which multiplication algorithm to use and makes use of similar heuristics to switch between the algorithms as they recurse. For details of this see the section of the GMP manual on multiplication algorithms [2]. The paper on the optimizations performed by the GMP team when implementing the Schönhage-Strassen algorithm provides an insight into some of the best practices for optimizing FFT based multiplications of algorithms [26].

A.2.3 The Nussbaumer algorithms

Nussbaumer described a number of different algorithms that perform fast multiplications in a ring of polynomials. Additionally, his book [55] on the topic usually presents a number of variants of each algorithm described. All of the algorithms operate in the ring of polynomials modulo $z^n + 1$, and variants are normally presented for at least the cases where n is prime and n is a power of 2.

Although the Nussbaumer method is not as well known as Schönhage-Strassen, over the years a number of people have worked to refine aspects of the Nussbaumer convolution [57], [48], [49], [67], [60].

To narrow the discussion we will deal with the Nussbaumer polynomial multiplication algorithm described by Hee [31], which focuses on the case where $n = 2^t$. Hee also provides an implementation that we modified and optimized, the details of which will be described shortly. This algorithm is similar to the Schönhage-Strassen algorithm for polynomials and so we will mostly focus on the differences between the two.

Bernstein provides a high level mathematical comparison of several multiplication algorithms including the Schönhage-Strassen and Nussbaumer algorithms [6].

The algorithm can be used to describe acyclic, cyclic and negacyclic convolutions with length 2^t, although Nussbaumer is primarily concerned with the negacylic convolutions. As usual, acyclic convolutions of length m are carried out as cyclic convolutions of length $2m$ with the appropriate zero-padding.

When computing a cyclic convolution of length $2m$

$$C(z) = A(z) \cdot B(z) \bmod z^{2m} - 1$$

we use the fact that $z^{2m} - 1 = (z^m - 1)(z^m + 1)$ to carry out the convolution by performing a length m circular convolution and a length m negacyclic convolution. Since we have $n = 2^t$, we can recursively decompose the cyclic convolutions, halting the recursion by reverting to scalar arithmetic when $n = 1$.

From there we only need an efficient method for computing a negacyclic convolution, which Nussbaumer provides. The first step is the conversion of the 1-dimensional polynomial operands to 2-dimensional polynomials. This step is a purely conceptual manoeuvre and requires no actual computation.

If $n = 2^t$ we define $L_1 = 2^{\lfloor t/1 \rfloor}$ and $L_2 = 2^{\lceil t/2 \rceil}$, and the polynomials are treated as $L_1 - 1$ degree polynomials with $L_2 - 1$ degree polynomials as coefficients. This is analogous to the grouping of bits in the Schönhage-Strassen algorithm, although with a different balance between the amount of work done at each level of recursion and the depth of the recursion. The difference is that the Schönhage-Strassen algorithm has to account for how it modifies the size of the groups of bits as it operates and the Nussbaumer algorithm defers this problem to the polynomial coefficients, which are assumed to be an appropriate size to begin with.

Like other FFT algorithms, the Nussbaumer algorithm proceeds by taking DFTs of the operands. In this case the equivalent to the DFT in the ring of polynomials is referred to as a polynomial transform, which operates on 2-dimensional polynomials.

Normally the point by point multiplication that occurs per the convolution theorem runs along a 1-dimensional sequence and multiplies scalars. In this case the multiplications occur along the dimension with size L_1, which has been zero-padded to length $2L_1$, and the point by point multiplications involve 1-dimensional polynomials with degree $L_2 - 1$.

Due to the way the algorithm operates the multiplications can be carried out using negacylic convolutions. This is how the Nussbaumer algorithm recurses, a process which stops when $n = 2$ and the negacylic convolutions are computed by more direct means.

After the recursive step has been performed the inverse polynomial transform is applied, completing the computation. Unlike the Schönhage-Strassen algorithm, there is no need for pre- or post-scaling to satisfy the requirements of the convolution theorem.

A.2.4 Time and memory bounds

Nussbaumer's algorithm has its origins in signal processing and most of the interest in it has been in the signal processing field. For the most part the bounds on the running time is stated as $O(n \log n)$ operations, which is usually supplemented with a table of the number of multiplications and additions required to perform the convolutions for sequences of different lengths.

We find an upper bound for the time taken by the algorithm in this section partly as a matter of rigour and partly out of curiosity. We also work out the bounds on the amount of memory that the algorithm requires for working space, both out of interest and as it allows an optimization that will be discussed in the next section.

While developing the upper bound on the running time we will prefer to express n, L_1 and L_2 in terms of t such that

$$n = 2^t \qquad\qquad L_1 = 2^{\lfloor t/2 \rfloor} \qquad\qquad L_2 = 2^{\lceil t/2 \rceil}$$

and will refer to operations taking time $O(m)$ for m-bit coefficients simply as operations. These include the addition, shift and assignment operations.

At each level of recursion a negacyclic convolution of length 2^t performs 2 polynomial transform and an inverse polynomial transform, each of which requiring $4 \cdot \log(2 \cdot 2^{\lfloor t/2 \rfloor})$ operations to implement, and $6 \cdot 2^t$ sundry operations. The recursion requires that a further $2 \cdot 2^{\lfloor t/2 \rfloor}$ negacyclic convolutions of length $2^{\lceil t/2 \rceil}$ be carried out.

We use T_t for the number of operations required to implement a negacyclic convolution with length 2^t, and start with

$$T_t = O(t \cdot 2^t + 2 \cdot 2^{\lfloor t/2 \rfloor} \cdot T_{\lceil t/2 \rceil})$$

We can expand this to

$$\begin{aligned} T_t &= O(t \cdot 2^t + 2 \cdot 2^{\lfloor t/2 \rfloor} \cdot \lceil t/2 \rceil \cdot 2^{\lceil t/2 \rceil} + 4 \cdot 2^{\lfloor t/2 \rfloor} 2^{\lfloor \lceil t/2 \rceil /2 \rfloor} \cdot T_{\lceil t/4 \rceil}) \\ &= O(t \cdot 2^t + (t+e) \cdot 2^t + 4 \cdot 2^{\lfloor t/2 \rfloor + \lfloor \lceil t/2 \rceil /2 \rfloor} \cdot T_{\lceil t/4 \rceil}) \end{aligned}$$

where $e \in \{0, 1\}$ is the potential rounding error occurring when we evaluate $2 \cdot \lceil t/2 \rceil$, or more generally when we evaluate $2^k \cdot \lceil t/2^k \rceil$.

Note that we use the fact that

$$\left\lfloor \frac{b-a}{b} \right\rfloor + \left\lceil \frac{a}{b} \right\rceil = 1$$

because

$$\left\lfloor \frac{b-a}{b} \right\rfloor = \frac{b-a}{b} - \frac{|b-a|_b}{b} \text{ and } \left\lceil \frac{a}{b} \right\rceil = \frac{a}{b} + \frac{|b-a|_b}{b}$$

in the expansion

We need to simplify the exponent of the last term to continue.

Lemma A.1.
$$\left\lfloor \frac{2^k-1}{2^k} t \right\rfloor + \left\lfloor \left\lceil \frac{1}{2^k} t \right\rceil /2 \right\rfloor = \left\lfloor \frac{2^{k+1}-1}{2^{k+1}} t \right\rfloor$$

Proof. We restate the lemma as

$$\left\lfloor \frac{2^{k+1}-2}{2^{k+1}} t \right\rfloor + \left\lfloor \left\lceil \frac{2}{2^{k+1}} t \right\rceil /2 \right\rfloor = \left\lfloor \frac{2^{k+1}-1}{2^{k+1}} t \right\rfloor$$

and set $t = u \cdot 2^{k+1} + v$ with $0 \le v < 2^{k+1}$.

We represent the first term in terms of u and v with

$$\left\lfloor \frac{2^{k+1}-2}{2^{k+1}} t \right\rfloor = u \cdot 2^{k+1} + (v - 2u)$$

The second term simplifies to u as follows

$$\left\lfloor \left\lceil \frac{2}{2^{k+1}} t \right\rceil /2 \right\rfloor = \left\lfloor \left\lceil \frac{2 \cdot (u \cdot 2^{k+1} + v)}{2^{k+1}} \right\rceil /2 \right\rfloor = \left\lfloor \left(2u + \left\lceil \frac{2v}{2^{k+1}} \right\rceil \right) /2 \right\rfloor = u$$

because $0 \le v < 2^{k+1}$ and so

$$\begin{aligned} &\text{if } \left\lfloor v/2^k \right\rfloor = 0 \text{ then } \left\lceil \frac{2v}{2^{k+1}} \right\rceil = 0 \text{ and } \lfloor 0/2 \rfloor = 0 \\ &\text{if } \left\lfloor v/2^k \right\rfloor = 1 \text{ then } \left\lceil \frac{2v}{2^{k+1}} \right\rceil = 1 \text{ and } \lfloor 1/2 \rfloor = 0 \end{aligned}$$

Adding these terms completes the lemma since

$$u \cdot 2^{k+1} + (v - u) = \left\lfloor \frac{2^{k+1}-1}{2^{k+1}} t \right\rfloor$$

□

If we expand the original equation for the running time k times we get

$$\begin{aligned} T_t &= O(t \cdot 2^t + (t+e) \cdot 2^t + 4 \cdot 2^{\lfloor t/2 \rfloor + \lfloor \lceil t/2 \rceil /2 \rfloor} \cdot T_{\lceil t/4 \rceil}) \\ &= O(t \cdot 2^t + (t+e) \cdot 2^t + 4 \cdot 2^{\lfloor 3/4t \rfloor} \cdot T_{\lceil t/4 \rceil}) \\ &= \ldots \\ &= O(t \cdot 2^t + k \cdot (t+e) \cdot 2^t + 2^{k+1} \cdot 2^{\lfloor (2^{k+1}-1)/2^{k+1} t \rfloor} \cdot T_{\lceil t/2^{k+1} \rceil}) \end{aligned}$$

If we expand this until we reach $T_{\lceil t/2^{k+1} \rceil} = T_1$, which occurs when $k = \lfloor \log t \rfloor$, we will have $O(2^t \cdot t \cdot \lfloor \log t \rfloor)$ operations in the recursive part of the negacylic convolution. Each of the convolutions corresponding to the $O(2^t \cdot t)$ instances of T_1 are dealt with by a short convolution routine that uses $O(1)$ multiplications and additions.

This means that the upper bound for the running time of 2^t length negacyclic convolution is

$$T_t = O(2^t \cdot t \cdot \log t) \tag{A.2}$$

Cyclic convolutions are performed by recursive decomposition into cyclic and negacylic convolutions. The decomposition for a sequences with length 2^k takes $O(2^k)$ time and so we see

$$\sum_{k=1}^{t-1} O(2^k) = O(2^t)$$

time spend on the decomposition and since

$$\sum_{k=1}^{t-1} 2^k \cdot k \cdot \lfloor \log k \rfloor < 2^t \cdot t \cdot \lfloor \log t \rfloor$$

we see

$$\sum_{k=1}^{t-1} O(2^k \cdot k \cdot \lfloor \log k \rfloor) = O(2^t \cdot t \cdot \log t)$$

time spent on the negacyclic convolutions.

This means that cyclic and negacyclic convolutions have the same asymptotic running time, and the difference in actual running time should be at most $O(2^t)$. As acyclic convolutions are cyclic or negacyclic convolutions performed on sequences that have been zero padded to twice their original length they will require $O(2^{t+1} \cdot (t+1) \cdot \log(t+1)) = O(2^t \cdot t \cdot \log t)$ operations to perform.

The negacyclic convolution of polynomials with length 2^t requires $4 \cdot 2^t$ elements of working space. This working space is initialized with the two input polynomials with each being zero padded to length $2 \cdot 2^t$.

The recursive part of the algorithm computes the $2 \cdot 2^{\lfloor n/2 \rfloor}$ negacyclic convolutions of length $2^{\lceil n/2 \rceil}$ sequentially, so we can state the memory required to perform a 2^t length negacyclic convolution as

$$M_t = 4 \cdot 2^t + M_{\lceil t/2 \rceil} = 4 \sum_{k=0}^{\lfloor \log t \rfloor} 2^{\lceil t/2^k \rceil} \tag{A.3}$$

This will be minimized when $t = 2^j$ as there will be no rounding up during the divisions, and we will see $M_t = 8 \cdot 2^t$. If we consider the case where $t = 2^j - 1$ then all of the divisions round up and we have a maximum value $M_t = 16 \cdot 2^t$.

By noticing that we will never round upwards in Equation A.3 when $k = 0$ we can refine the bounds to

$$4 \cdot 2^t + 8 \cdot 2^{\lceil t/2 \rceil} \leq M_t \leq 4 \cdot 2^t + 16 \cdot 2^{\lceil t/2 \rceil} \tag{A.4}$$

Usually this would be stated in asymptotic notation as $O(2^t)$, but we shall shortly describe how we use the extra information to improve our implementation.

Cyclic convolutions of length 2^t are implemented by a cyclic convolution and a negacyclic convolution, both of length 2^{t-1}. The recursive negacylic convolutions are the only significant contributor to the memory usage of a cyclic convolution, and so we see cyclic convolutions using a peak of M_{t-1} working space.

Acyclic convolutions are performed as cyclic convolutions of sequences that have been zero padded to twice their original length, which means that an acyclic convolution has a peak memory usage of M_t.

A.3 Practice

A.3.1 Implementation and optimizations

We have extended an implementation due to Hee, which computes cyclic and negacylic convolutions for sequences with length 2^t for some integer t. The trivial changes were the made first, and these include the alteration of the code to use GMP for the elements of the sequences as well as some function inlining and code reordering. Most of the less trivial changes described in this section are benchmarked in Section A.3.2

The original implementation due to Hee made use of a predefined array for use as working space. The alternative would be to dynamically allocate and de-allocate the memory required by the algorithm, imposing an unnecessary overhead on the running time.

Hee's implementation sets a maximum length for the sequences, $S = 65536$, and creates an array of length $4S + 300$ for the working space. We did not want to limit the length of our sequences or to allocate more memory than we required. We also noted that Hee's implementation crashed for some longer sequences, which we attributed to the 300 elements of extra space being insufficient in some circumstances.

The memory bound derived in the previous section allows us to allocate enough memory to convolve a sequence of a given length without allocating more memory than is necessary. We also experimented with using the memory pooling facilities from the Boost C++ libraries to reuse a this working space in situations where many convolutions had to be performed in sequence.

The original implementation had a function to compute the negacyclic convolution of 2 element sequences which ends the recursion in the main algorithm. We added a negacyclic convolution function for sequences of 4 elements as an additional point at which the recursion exits. As this function is faster than performing the convolution with the original implementation we see an improvement in the running time for convolutions with size $n \geq 4$.

For the most part convolution only really makes sense for sequences of the same length, and so both Hee's and our implementation of the convolution algorithm assume that this will be the case. We cannot expect the same to be true for polynomial multiplication in general. Zero padding the sequences to the same length will be inefficient, especially if the difference in length is substantial. There is a simple method to deal with this.

Let u and v be the least integers such that $|A(z)| < 2^u$ and $|B(z)| < 2^v$ with $u < v$. We can perform the multiplication by first partitioning $B(z)$ such that

$$B(z) = \sum_{i=0}^{2^{v-u}} B_i(z) \cdot z^{i \cdot 2^u} \text{ where } B_i(z) = \sum_{j=0}^{2^u} B(z)[i \cdot 2^u + j] \cdot z^j$$

from which we get

$$A(z) \cdot B(z) = \sum_{i=0}^{2^{v-u}} A(z) \cdot B_i(z) \cdot z^i$$

Where zero padding the sequences would use T_v time and M_v working space we see that this optimization brings that down to $2^{v-u} T_u$ time and M_u memory. We will look at the effectiveness of this shortly.

The convolution of complex data normally requires 4 times as many real multiplications to compute as a convolution of real data of the same length. A simple mathematical trick brings this down to 3 times as many real multiplications at the cost of an increased number of real additions. The trick is widely used since multiplications are usually more computationally expensive than additions.

Nussbaumer notes that since $z^n = -1 \bmod z^n + 1$, when n is even we have $z^{n/2} = i \bmod z^n + 1$, and describes a way to make use of this so that complex convolutions require twice as many real multiplications as would be used if the data were purely real, while not significantly increasing the number of required real additions [**?**].

Nussbaumer also provides several optimizations for cases that require the use of multi-dimensional sequences. We have occasionally come across interesting problems that require the use of polynomials in multiple variables, which are equivalent to multi-dimensional sequences. Our implementation is in C++ and the polynomial class is implemented as a template class, allowing us to chose between different types to use as the coefficients. This can be used to create polynomials in multiple variables by specifying that polynomial have coefficients that are also polynomials. The infrequent use of and acceptable performance provided by this technique, coupled with the added complexity of optimizing for multi-dimensional sequences, led us not to pursue those optimizations for this work.

A.3.2 Benchmarks

The benchmarks were run on a single core of an AMD Athlon 64 X2 Dual Core processor. Each core runs at 1 GHz and has 64Kb of L1 cache for data and 512 Kb of L2 cache for data and instructions. The machine has 2 GB of RAM, although this is hardly relevant since the benchmarks were never memory bound.

The polynomials used in these benchmarks were randomly generated with each coefficient c being in the range $2^{512} \leq c < 2^{513}$. Each test was run 100 times and the mean and standard deviation of the running time were calculated. The standard deviation was calculated solely to check that the number of samples was sufficient and does not feature in these benchmarks.

All graphs in the section are logarithmic in both axes and display polynomial size or sizes versus time.

The standard/naive, Karatsuba and Nussbaumer algorithms are compared in Figures A.1 and A.2. These figures display the time it takes to multiply two $k = 2^j$ length polynomials with real integer and complex integer coefficients respectively. The Schönhage-Strassen normally only outperforms other algorithms for exceptionally large inputs, and so it is somewhat surprising to see the Nussbaumer algorithm outperforming the other algorithms so quickly.

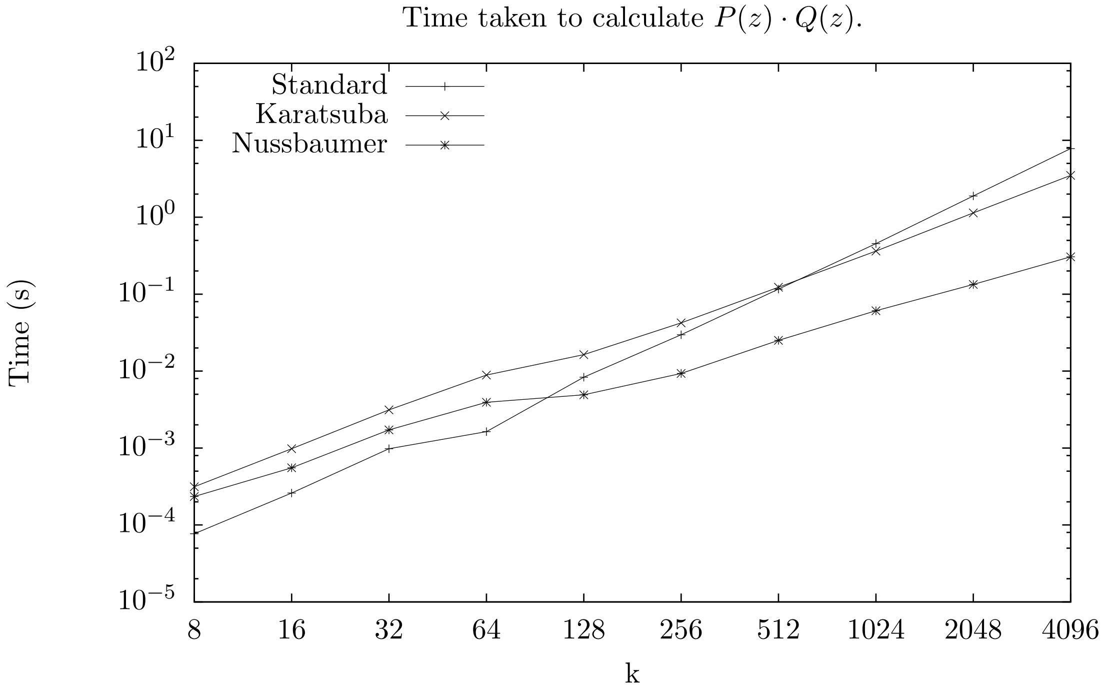


Figure A.1: Various multiplication algorithms, integer coefficients, $|P(z)| = |Q(z)| = k$.

Figure A.2 is not very different from Figure A.1. The multiplications with the complex data take more time than the multiplications with the real data. Other than that the relative performance is very similar.

The case where the operands to the polynomial multiplication have different sizes is interesting. The standard algorithm can avoid doing extra work by scanning the polynomials to find the highest exponent in the polynomial.

The other algorithms get some benefit from the difference in the number of coefficients, but it is significantly worse than the gains made by the standard algorithm. This occurs because the Karatsuba and Nussbaumer algorithms mix the values as they operate, and so any coefficients which are used to zero-pad the polynomials will usually become non-zero and important to the running of the algorithm before very long. This is what dilutes the effect of the size differences on the running time.

Note that the "Optimized Nussbaumer" algorithm in Figure A.3 is the Nussbaumer algorithm with the modification which avoids the need to zero pad operands with different lengths. We

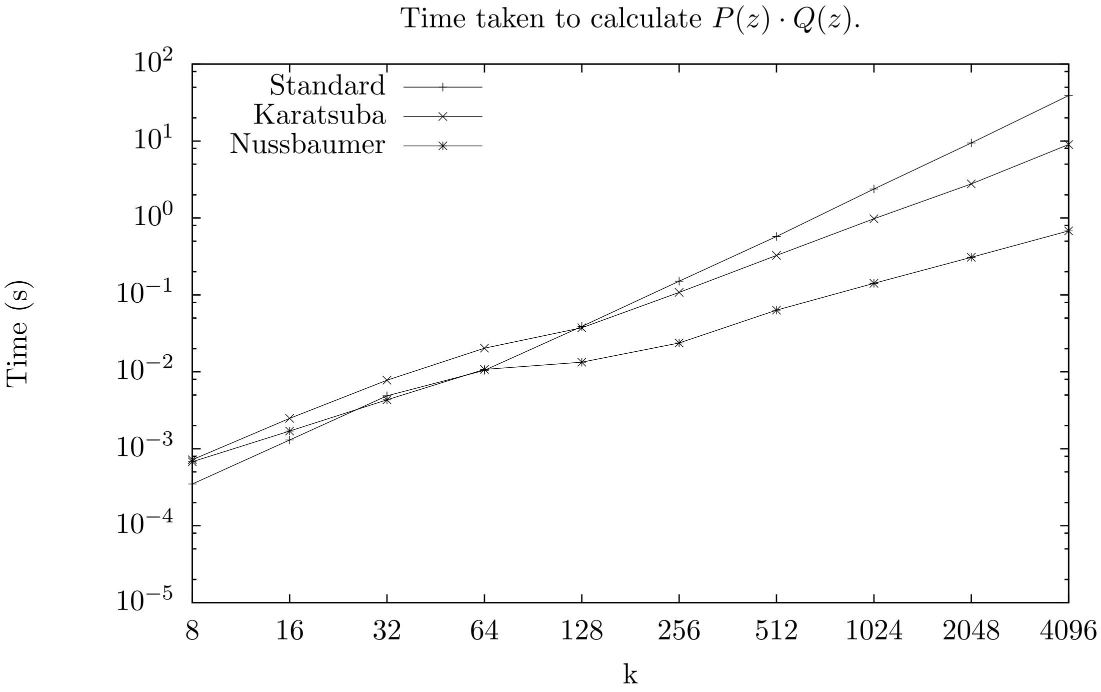


Figure A.2: Various multiplication algorithms, complex integer coefficients, $|P(z)| = |Q(z)| = k$.

see here that this optimization has very little effect on the running time of the multiplication algorithm. This particular optimization reduces the memory required for the multiplication and so it is good to know that the decrease in memory usage does not have an associated increase in running time.

Finally, we looked at the change in running time when polynomials in different numbers of variables were used. This is equivalent to working with multi-dimensional sequences. The running time increases sharply with the number of variables, to the point that we stopped the benchmarks early for polynomials in 3 variables.

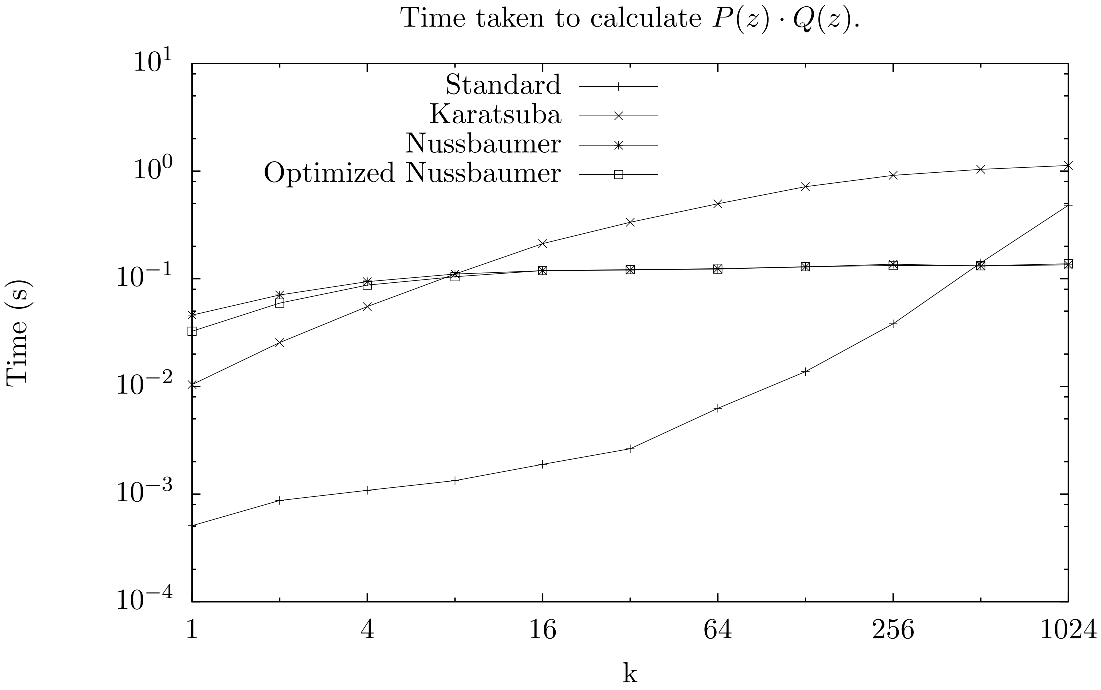


Figure A.3: Nussbaumer multiplication algorithm, integer coefficients, $|P(z)| = 2048$, $|Q(z)| = k$.

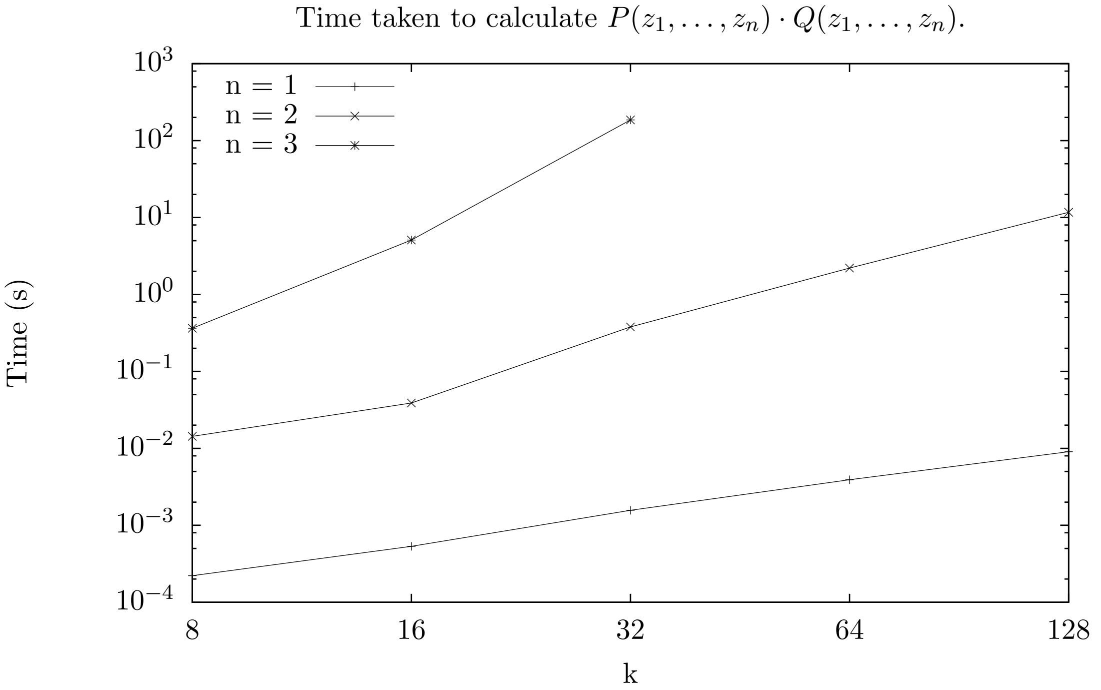


Figure A.4: Nussbaumer multiplication algorithm used for multivariate polynomials, integer coefficients, $|P(z)| = |Q(z)| = k$

Appendix B

Accompanying papers

Two papers were submitted and accepted during the course of this degree.

The first of the papers was published in the ANZIAM Journal as part of the proceedings of the 2006 CTAC conference [42].

The ANZIAM Journal is published by Cambridge University Press, who hold the copyright to the paper, but allow authors to reproduce their papers in subsequent work and on personal and institutional websites without requiring a permission request. The paper is reprinted with permission.

The second paper was published in Theoretical Informatics and Applications in 2008 [43].

Theoretical Informatics and Applications is published by EDP Sciences, who hold the copyright to the paper, but allow authors to reproduce their papers in subsequent works with the publishers PDFs and allow copies to be provided via personal and institutional websites. The original publication is available at `www.edpsciences.org/ita`.

RAIRO-Theor. Inf. Appl. **42** (2007) 309–322
DOI: 10.1051/ita:2007024

Available online at:
www.rairo-ita.org

CENSUS ALGORITHMS FOR CHINESE REMAINDER PSEUDORANK

David Laing[1] and Bruce Litow[1]

Abstract. We investigate the density and distribution behaviors of the chinese remainder representation pseudorank. We give a very strong approximation to density, and derive two efficient algorithms to carry out an exact count (census) of the bad pseudorank integers. One of these algorithms has been implemented, giving results in excellent agreement with our density analysis out to 5189-bit integers.

Mathematics Subject Classification. 11Y99, 68R99.

1. Introduction

Motivation

CRR, chinese remainder representation, has been used as an important tool in parallel arithmetic complexity research. CRRS, CRR systems, have been used in studying parallel complexity of integer division, and also in connection with certain arithmetic circuit complexity problems. See [2,3,10]. A key concept, the CRR pseudorank, introduced in [3], has proved to be very useful in developing $\mathbf{NC}^1$ algorithms for integer comparison. Unlike integer comparison using radix notation, which can be implemented on a DFA (deterministic finite automaton), comparison in CRR is difficult since there is no evident correspondence between ordering and the notation.

Despite its importance in CRR-related research, there appears to be little information on how the pseudorank behaves over a given CRRS. The pseudorank of integer x is known to be either equal to or one less than the rank of x. Rank is a basic CRR function which we will define later on. Knowledge of the rank makes

Keywords and phrases. Chinese remainder representation, rank, pseudorank, pseudorank census algorithms.

[1] School of Information Technology, James Cook University, Townsville, Qld. 4811, Australia; bruce@cs.jcu.edu.au, david.laing1@jcu.edu.au

Article published by EDP Sciences

CRR-intrinsic comparison quite simple. Unfortunately, no efficient CRR-intrinsic method of computing rank is known. This paper gives the first clear indication of how pseudorank compares with rank over a CRRS. In particular, we give a sharp estimate of the density of integers for which rank and pseudorank differ, and show that there are subintervals of a CRRS where either all rank and pseudorank match, or differ. This is the first step in determining how rank-pseudorank mismatches are distributed in a CRRS. We also describe two efficient algorithms for exactly counting the number of rank-pseudorank mismatches. One of these algorithms has been implemented, and results on density based on running it are presented for various CRRS, up to one capable of representing 5189-bit integers. These experimental results are in excellent agreement with our density estimate. Interestingly, our results and algorithms make no essential use of number theory. We hope that the more difficult problem of characterizing the distribution of rank-pseudorank mismatches, which is likely to require number theoretic and other deeper considerations, will be taken up by others. The significance of distribution information will be discussed in Section 2.

This work grows out of an earlier effort, the results of which are summarized in [8]. The algorithm described in that earlier work only gives an estimate of the number of rank-pseudorank msimatches, and although running in polynomial time is more complex and much slower than the newer algorithms described in this paper. However, elements of the original approach can still be found in the implemented algorithm (Algorithm 2) of this paper. The convolution approach to iterated polynomial multiplication that forms the basis of the original algorithm may be of independent interest to some readers. The source for this original implementation is available on request.

Structure of the paper

Section 2 defines CRR and pseudorank, and concludes with some results on rank-pseudorank match and mismatch distribution and density. Section 3 contains the descriptions and analysis of two new census algorithms, and includes tabulated experimental results from an implementation of one of them.

2. CRR and pseudorank

2.1. CRR basic facts

A CRRS can be specified by a single positive integer $P = p_1 \cdots p_r$, where $p_1, \ldots, p_r$ are consecutive odd primes. We will fix $p_1 = 3$. We will simply refer to P as a CRRS, also using CRRS to mean the integers $x < P$. The CRR of an integer x (integer will mean nonnegative integer) is the list $|x|_{p_1}, \ldots, |x|_{p_r}$, where $|x|_z = y$ means that y is the least integer such that $x \equiv y \bmod z$. The weak chinese remainder theorem asserts that each $0 \leq x < P$ is uniquely identified by its CRR. We write $|x|_P$ to indicate that the integer x is given in CRR, rather than in radix notation. On the other hand, $|x|_{p_i}$ indicates that radix notation is used.

The reason for the difference is that by Lemma 2.1, $|x|_{p_i} < \log P$, so its radix notation involves at most about $\log\log P$ bits. Notice that log is the base 2 logarithm, and ln the natural logarithm.

The following two results give some quantitative information about CRRS.

Lemma 2.1. *If $n > 2.89 \times 10^7$, and r is the largest integer such that $p_r < n$, then $r = \Theta(n/\log n)$, and $\exp(n + .001 \times n) > P > 2^n$.*

Proof. The claim on r follows from the prime number theorem. See [5]. The claim for P follows from the bound ($n > 2.89 \times 10^7$)

$$\left|\sum_{i=1}^{r} \ln p_i - n\right| \leq .0068 \cdot n/\ln n.$$

See [4]. From this bound we get

$$\exp(n + .0068n/(1.44 \cdot \log n)) > P > \exp(n - .0068n/(1.44 \cdot \log n)),$$

since $\log n > 1.44 \cdot \ln n$. It is straightforward that $\exp(n + .001 \times n) > P > 2^n$. □

From this point we take the view that we are interested in choosing a CRR that can represent all integers below 2^n. That is, we use n bits as the problem size. By Lemma 2.1, we have that $\log P = \Theta n$ can be imposed, and Θ-notation implies a small range for constants. We could actually obtain smaller values for r yielding $P > 2^n$, and while in practice this could be significant, for our study such a savings is irrelevant. All of our computations are understood to be in the Turing model. By polynomial time, then, we mean $n^{O(1)}$ time on a Turing machine, which is equivalent by Lemma 2.1 to $\log^{O(1)} P$ Turing machine time.

Lemma 2.2.

$$\sum_{i=1}^{r} 1/p_i = O(\log\log n).$$

Proof. See exercise II.9.d in [11]. □

2.2. An observation about parity and comparison in CRR

We first note that given the CRR for $x, y < P$, it is trivial to obtain $|x \diamond y|_P$, where $\diamond$ is addition, subtraction or multiplication. This follows from noting that $|x \diamond y|_{p_1}, \ldots, |x \diamond y|_{p_r}$ is the CRR of $|x \diamond y|_P$. Indeed, this is the chief reason why CRR plays an important role in analyzing parallel complexity of arithmetic operations.

Observe that knowing $|x|_2$ generally makes possible very efficient integer comparison algorithms. First, note that if $x < y < P$, then $|x - y|_P = x - y + P$. This is the "wrap" effect that complicates doing arithmetic in CRR. For example, if $x = 1$ and $y = 2$, we have $|1 - 2|_P = P - 1$. Of course, if $y < x < P$, then $|x - y|_P = x - y$. Next, we have that if $x < y$,

$$||x - y|_P|_2 \equiv |x - y + P|_2 \equiv |x|_2 + |y|_2 + 1 \bmod 2,$$

since $|P|_2 = 1$. This is why in our research, $p_1 = 3$ and not $p_1 = 2$. On the other hand, if $y < x$,

$$||x - y|_P|_2 = |x - y|_2 \equiv |x|_2 + |y|_2 \bmod 2.$$

Thus we can conclude for $x, y < P$, that $x > y$ iff $||x - y|_P|_2 \equiv |x|_2 + |y|_2 \bmod 2$. If parity were easy to compute in CRR (i.e. without some conversion into radix), CRR-intrinsic comparison would also be easy. There appear to be serious obstacles to CRR-intrinsic comparison. For example, see Theorem S, p. 255 in [6].

The next theorem is a strong form of the chinese remainder theorem. Recently, it has been rediscovered in [1]. It is the starting point for the parallel arithmetic work in [2,3].

Theorem 2.3. *There exists a unique integer, $q(x) < r$, such that*

$$x + q(x) \cdot P = \sum_{i=1}^{r} |x \cdot (P/p_i)^{p_i - 2}|_{p_i} \cdot P/p_i \, . \tag{1}$$

Proof. The RHS of equation (1) and x give equal residues $\bmod p_i$, for $i = 1, \ldots, r$. By the chinese remainder theorem we have that the RHS and x differ by a factor of P. Dividing the RHS by P we see that each resulting summand is less than 1, thus the RHS is less than $r \cdot P$. We can conclude that the RHS can be written as $x + q \cdot P$, and letting $q(x) = q$, which is clearly unique, we have the theorem. □

The integer $q(x)$ is called the rank of x. Given the rank, it is easy to compute parity from CRR data. Observe that from equation (1) we have

$$|x|_2 = \left| \sum_{i=1}^{r} x_i + q(x) \right|_2 , \tag{2}$$

where

$$x_i = |x \cdot (P/p_i)^{p_i - 2}|_{p_i},$$

and we have used $|P|_2 = 1$. The integers x_i are all available via CRR, so to compute $|x|_2$ it suffices to know $|q(x)|_2$. However, there is no known way to efficiently compute $|q(x)|_2$ in a CRR intrinsic manner. This difficulty is the motivation for introducing the pseudorank of x. Pseudorank will be defined in such a way that it is easy to compure it directly from CRR data. In fact, pseudorank can be computed by a finite automaton that can be constructed in polynomial time. This will be brought out in Section 3.

2.3. The pseudorank

Dividing through by P, from Theorem 2.3, we can write

$$x/P + q(x) = \sum_{i=1}^{r} x_i/p_i. \tag{3}$$

Let g be the least integer such that $2^g > 4r$. We define integers $\alpha(x), \beta(x)$ by $\alpha(x) < 2^g$ such that

$$2^g \cdot \beta(x) + \alpha(x) = \sum_{i=1}^{r} \lfloor 2^g \cdot x_i/p_i \rfloor. \tag{4}$$

Observe that

$$0 \leq x_i/p_i - \lfloor 2^g \cdot x_i/p_i \rfloor / 2^g < 1/2^g.$$

Thus, to g bits precision, $\beta(x) + \alpha(x)/2^g$ mimics $q(x) + x/P$. We call $\beta(x)$ the pseudorank of x.

The values $\lfloor 2^g \cdot x_i/p_i \rfloor$ can be precomputed, based solely on n, and the consequent choice of P, and stored in a table of diemensions $L \times 2^g$, where $L = \sum_{i=1}^{r} p_i$. We regard this table as indexed for each $i = 1, \ldots, r$ by x_i. We will see in Section 3 that a polynomial time constructible finite automaton can use CRR as input, and properly index into this table. Since $L < r \cdot p_r < r \cdot n$, and $2^g < 8r$, our table has size less than $8r^2 \cdot n \cdot O(\log n) = o(n^3)$ by Lemma 2.1, and the fact that an entry has size at most $g = O(\log r) = O(\log n)$ bits. There is nothing remotely like this economy of data known for computing the rank.

The next result is proved in [3,10].

Theorem 2.4. *If $x > P/4$, then $\beta(x) = q(x)$. If $\beta(x) \neq q(x)$, then $\beta(x) = q(x) - 1$.*

The integers below $P/4$ comprise the critical region of the CRRS. The integer x is said to be good if $q(x) = \beta(x)$, otherwise it is said to be bad. By Theorem 2.5, all bad integers are in the critical region.

We point out that the choice of $4r < 2^g < 8r$ is somewhat arbitrary, in that similar results concerning good and bad integers and the corresponding critical region could be obtained provided:

- $2^g > 2r$ (if $2^g < 2r$ the error in pseudorank compared to rank is too large to be of use), and
- $g = n^{O(1)}$ (otherwise we lose the polynomial size of the table, and other polynomial time bounds for our finite automaton constructions).

Theorem 2.6 shows that the concept of bad density makes sense for any choice $2^\ell \cdot r < 2^g < 2^{\ell+1} \cdot r$, subject to the two items just mentioned. First, it is straightforward to generalize Theorem 2.5 to show that if $2^\ell \cdot r < 2^g < 2^{\ell+1} \cdot r$, then all bad integers are below $P/2^\ell$. Next, the proof of Theorem 2.6 tells us that the ratio of census of bad integers to P is essentially $r/2^{g+1}$. From this and the upper bound of $P/2^\ell$ on bad integers it follows that bad density is always in the range between 1/4 and 1/2.

Our choice $4r < 2^g < 8r$ to leads to some technical simplifications over the "minimal" choice of $2r < 2^g < 4r$ in applications not covered in this paper. The reader may want to consult [1] on the choice $2r < 2^g < 4r$.

To see the relationship between rank and pseudorank for our choice of g, consider a small CRR given by $P = 3 \cdot 5 \cdot 7 = 105$. Here $r = 3$, so $g = 4$ will do since $2^4 = 16 > 4 \cdot 3 = 12$. First choose $x = 14$, by direct calculation of equation (1), we get

$$\sum_{i=1}^{3} (14)_i \cdot 105/p_i = 17 = 14 + q(14) \cdot 105.$$

From this we get $q(14) = 1$. Next, we compute $\beta(14)$ *via*

$$\sum_{i=1}^{3} \lfloor 16 \cdot (14)_i/p_i \rfloor = 16 \cdot \beta(14) + \alpha = 17.$$

It follows that $\beta(14) = 1 = q(14)$, so 14 is good. On the other hand, we get $q(2) = 1$, but

$$\sum_{i=1}^{3} \lfloor 16 \cdot (2)_i/p_i \rfloor = 16 \cdot \beta(2) + \alpha = 15.$$

This means that $\beta(2) = 0$, so 2 is bad.

Let B denote the bad integers. We write $\sum_x$ for $\sum_{x=0}^{P-1}$. The (bad) census is just the indicator sum $\sum_{x \in \mathcal{B}} 1$.

2.4. Distribution and density results

We can say something about the distribution of good and bad integers in extreme parts of the critical region.

Theorem 2.5. *If $x < \frac{r \cdot P}{n \cdot 2^g}$, and x and P are coprime, then x is bad. If $x > P/4 - P/2^g$, then x is good.*

Proof. We treat the bad integer case. We derive a contradiction by assuming $x < \frac{r \cdot P}{n \cdot 2^g}$ and x is good. Let a be the integer for which $x/P = a/2^g + \epsilon$ such that $0 \leq \epsilon < 1/2^g$. Since

$$x/P < \frac{r}{n \cdot 2^g} < 1/n < 1/2^g,$$

we have that

$$\epsilon = x/P < \frac{r}{n \cdot 2^g},$$

and $a = 0$. Since x and P are coprime, $x_i \neq 0$ for $i = 1, \ldots, r$. Let ϵ_i be

$$x_i/p_i - \lfloor 2^g \cdot x_i/p_i \rfloor / 2^g.$$

Note that $0 \leq \epsilon_i$. Also by the upper bound on ϵ and nonnegativity of the ϵ_i, for at least one i,

$$\epsilon_i < \frac{1}{n \cdot 2^g}. \tag{5}$$

It is clear that $x_i \neq 0$ implies that

$$0 < \epsilon_i \cdot p_i \cdot 2^g = x_i \cdot 2^g - \lfloor 2^g \cdot x_i/p_i \rfloor \cdot p_i.$$

On the other hand, if ϵ_i satisfies equation (5), then since $p_i \leq p_r < n$ by Lemma 2.1,

$$0 < x_i \cdot 2^g - \lfloor 2^g \cdot x_i/p_i \rfloor \cdot p_i < 1,$$

which is impossible.

We treat the good integer case, also by contradiction. This result is really a slight refinement of Theorem 2.4. Since by Theorem 2.4, $x > P/4$ implies that x is good, we can assume that $P/4 > x > P/4 - P/2^g$, and that x is bad. By equation (4), Theorem 2.4 and the fact that

$$0 \leq \epsilon_i = x_i/p_i - \lfloor 2^g \cdot x_i/p_i \rfloor / 2^g < 1/2^g,$$

we have

$$1 + x/P - \sum_{i=1}^{r} \epsilon_i = \alpha(x)/2^g.$$

But, this is impossible because $\alpha(x)$ is an integer less than 2^g while

$$\sum_{i=1}^{r} \epsilon_i < r/2^g < 1/4,$$

and $1/4 - 1/2^g < x/P < 1/4$. □

The density of bad integers is defined to be the ratio of the number of bad integers to $P/4$.

Theorem 2.6. *The density of bad integers is*

$$r/2^{g-1} - O(\log^2(n)/2^g).$$

Proof. By Theorem 2.4, the bad integer census, written explicitly as the difference of two sums is

$$\sum_x q(x) - \sum_x \beta(x). \tag{6}$$

We develop expressions for the two sums in equation (6). From equation (3),

$$\sum_x q(x) = \sum_x \sum_{i=1}^{r} x_i/p_i - \sum_x x/P. \tag{7}$$

We write, using twice that for integers a, b, $\lfloor a/b \rfloor = a/b - |a|_b/b$,

$$\beta(x) = \left(\sum_{i=1}^{r} (2^g \cdot x_i - |2^g \cdot x_i|_{p_i})/p_i - \alpha(x) \right) /2^g.$$

Since 2^g and p_i are coprime, multiplying each element of $\{0, 1, \ldots, p_i - 1\}$ by 2^g and reducing $\mathrm{mod} p_i$ results in $\{0, 1, \ldots, p_i - 1\}$, we have

$$\sum_x \beta(x) = \sum_x \sum_{i=1}^{r} x_i/p_i - \frac{1}{2^g} \sum_x \sum_{i=1}^{r} x_i/p_i - \sum_x \alpha(x)/2^g \ .$$

Thus, the census of bad integers can be written as

$$\frac{1}{2^g} \sum_x \sum_{i=1}^{r} x_i/p_i + \frac{1}{2^g} \sum_x \alpha(x) - \sum_x x/P. \tag{8}$$

Let

$$2^g \cdot x_i/p_i = \lfloor 2^g \cdot x_i/p_i \rfloor + \delta_i(x).$$

We can then write

$$\sum_{i=1}^{r} \lfloor 2^g \cdot x_i/p_i \rfloor = \sum_{i=1}^{r} 2^g \cdot x_i/p_i - \sum_{i=1}^{r} \delta_i(x),$$

and using equation (3) we get

$$\sum_{i=1}^{r} \lfloor 2^g \cdot x_i/p_i \rfloor = 2^g \cdot x/P + 2^g \cdot q(x) - \sum_{i=1}^{r} \delta_i(x).$$

Since

$$\alpha(x) = \left| \sum_{i=1}^{r} \lfloor 2^g \cdot x_i/p_i \rfloor \right|_{2^g},$$

we get

$$\alpha(x) = 2^g \cdot x/P - \sum_{i=1}^{r} \delta_i(x). \tag{9}$$

From equations (8) and (9) we can express the bad integer census as

$$\frac{1}{2^g} \sum_x \sum_{i=1}^{r} x_i/p_i - \frac{1}{2^g} \sum_x \sum_{i=1}^{r} \delta_i(x). \tag{10}$$

We analyze the error term

$$\Delta = \sum_x \sum_{i=1}^{r} \delta_i(x)$$

by first estimating $\Delta_i = \sum_x \delta_i(x)$. The possible values for x_i as x ranges from 0 to $P-1$ are, of course, $0, \ldots, p_i - 1$. Each value occurs P/p_i times. Consider the values $k, k+1, \ldots, k+h$ such that

$$\ell < 2^g \cdot k/p_i < \cdots < 2^g \cdot (k+h)/p_i < \ell + 1.$$

An upper bound on the sum of the fractional parts of these fractions is clearly

$$\sum_{j=1}^{h+1} j/p_i.$$

Thus, $\Delta_i < \frac{P}{p_i} \cdot \sum_{j=1}^{h+1} j/p_i$. Also, we have that

$$h < p_i/2^g = O(\log n).$$

It follows that

$$\Delta_i = O(P \cdot \log^2 n/p_i^2).$$

From this we have

$$\Delta = \sum_{i=1}^{r} \Delta_i = O(P \cdot \log^2(n)),$$

since $\sum_{k=1}^{\infty} 1/k^2 = \Theta(1)$.

It is straightforward to show that

$$\sum_x x_i/p_i = 1/2 \cdot (P - P/p_i).$$

Note that $\sum_{i=1}^{r} P/p_i = O(P \cdot \log\log n)$ by Lemma 2.2. Combining this, equation (10), the upper bound on Δ and dividing by $P/4$ to obtain a density, we have the theorem. □

We make two observations about Theorem 2.6. First, the error bound goes to zero rapidly since $2^g = \Theta(n/\log n)$. Second, asymptotically, the density of bad integers oscillates in sawtooth fashion between 1/4 at r a power of 2, and 1/2 at r one less than a power of 2.

3. Two pseudorank census algorithms

3.1. WFA preliminaries

We use weighted finite automata (WFA) to describe the algorithms of this section. WFA is a huge topic. Examples of directions in WFA research are presented in [7,9]. The first general algebraic treatment of WFA appears to be due to Schützenberger. We deal with only those aspects that are relevant to our algorithms in this section.

We work over CRRS P as before. For our CRR oriented purposes a WFA, F, is a tuple Σ, U, V, X_σ, where Σ is a finite alphabet, U is a $1 \times h$ matrix, V is a $h \times 1$ matrix, and for each $\sigma \in \Sigma$, X_σ is a $h \times h$ matrix. All matrix entries are rational coefficient polynomials in the variable t. We regard the indices of these matrices as states. The integer h is the referred to as the number of states. In all of our WFA, h will be bounded above as $n^{O(1)}$. It is sometimes useful to think of the nonzero entries of U as "start" states, and the nonzero entries of V as "final" states. Σ is partitioned as $\Sigma_1 \cup \cdots \cup \Sigma_r$, where the symbols of Σ_i are in bijective correspondence with $0, 1, \ldots p_i - 1$. The r-behavior of F, denoted by $\langle F \rangle_r$ is defined to be

$$U \cdot \left(\sum_{\sigma \in \Sigma} X_\sigma \right)^r \cdot V.$$

We can write $\langle F \rangle_r$ as

$$\sum_{\sigma_{i_1} \cdots \sigma_{i_r}} U \cdot X_{\sigma_{i_1}} \cdots X_{\sigma_{i_r}} \cdot V,$$

and a summand is denoted by $\langle F \rangle_r(\sigma_{i_1} \cdots \sigma_{i_r})$.

All of the square matrices X_σ will be structured so that unless $j = i + 1$, where $\sigma_i \in \Sigma_i$ and $\sigma_j \in \Sigma_j$, then $X_{\sigma_i} \cdot X_{\sigma_j} = 0$. That is we will enforce $\langle F \rangle_r(\sigma_{i_1} \cdots \sigma_{i_r}) = 0$ unless $\sigma_{i_j} \in \Sigma_j$ for $j = 1, \ldots, r$. This means that if $\sigma_{i_1} \cdots \sigma_{i_r}$ does not correspond to a CRR, any WFA that we construct will generate a zero summand for that string. This structure will emerge naturally in each of our constructions.

3.2. Algorithm 1

Recall that all matrix entries are rational coefficient polynomials in the variable t.

Lemma 3.1. *A WFA, F, can be computed in $n^{O(1)}$ time such that*

$$\langle F \rangle_r = \sum_x (1 - \beta(x) \cdot t) \cdot \left(\prod_{i=1}^r (1 + x_i \cdot t/p_i) \right).$$

Proof. The states are ordered pairs of integers (k, f), where $k = 0, \ldots, r$, and $f = 0, \ldots, 2^g \cdot r - 1$. For $\sigma \in \Sigma_i$, a nonzero entry of X_σ is indexed by a row, column pair of states

$$(i-1, f), (i, f + \lfloor 2^g \cdot |\sigma \cdot (P/p_i)^{p_i - 2}|_{p_i}/p_i) \rfloor).$$

The entry at this location is

$$1 + |\sigma \cdot (P/p_i)^{p_i-2}|_{p_i} \cdot t/p_i.$$

The only nonzero entry of U has index $1, (0,0)$ and the entry there is 1. A nonzero entry of V is indexed by $(r, 2^g \cdot \beta + \alpha), 1$, and the entry there is $1 - \beta \cdot t$.

Recalling equation (4), it is clear that a nonzero entry of $(\sum_{\sigma\in\Sigma} X_\sigma)^k$ is indexed by

$$(0,0), \left(k, \sum_{i=1}^{k} \lfloor 2^g \cdot \sigma_i \cdot (P/p_i)^{p_i-2}|_{p_i}/p_i \rfloor\right),$$

where $\sigma_i \in \Sigma_i$. The entry at this location is

$$\prod_{i=1}^{k} 1 + |\sigma_i \cdot (P/p_i)^{p_i-2}|_{p_i} \cdot t.$$

The claim for $\langle F\rangle_r$ follows from this, the chinese remainder theorem, and the definition of V.
The time bound for computing F is clear from the description of its construction. □

Let $B = \langle F\rangle_r$, where F is the WFA in Lemma 3.1, and $B' = \frac{dB}{dt}|_{t=0}$. Here is algorithm 1: compute $B' - (P-1)/2$.

Theorem 3.2. *Algorithm 1 computes the bad census in polynomial time.*

Proof. By Lemma 3.1, equation (3), and calculus,

$$B' = \sum_x \left(-\beta(x) + \sum_{i=1}^{r} x_i/p_i\right) = \sum_x \left(x/P + q(x) - \beta(x)\right).$$

By equation (6),

$$B' = \sum_{x \text{ bad}} 1 + \sum_x x/P,$$

but $\sum_x x/P = (P-1)/2$. The time bound follows from the fact that B is a polynomial in t which is computable in polynomial time, so B' is computable in polynomial time. □

3.3. Algorithm 2

We have implemented and run Algorithm 2. Before describing the main steps of the algorithm, which again counts the bad integers we need to cover some preliminaries. We define a kind of polynomial multiplication, denoted by $\odot$ as follows. Let P, Q be polynomials in the variable z, then $P \odot Q$ is the polynomial obtained from $P \cdot Q$ (ordinary Cauchy product) by reducing all exponents of z $\bmod 2^g$, and collecting like terms. Thus, $P \odot Q$ has degree at most $2^g - 1$. The r-fold $\odot$ product of $P_1, \ldots, P_r$ is denoted by $\bigodot_{i=1}^{r} P_i$. The evaluation of a polynomial P at $z = a$ is denoted by $P(a)$. For $i = 1, \ldots, r$ let

$$\mu_i(x) = \lfloor 2^g \cdot x_i/p_i \rfloor.$$

Here is Algorithm 2.

(1) For $i = 1, \ldots, r$, construct the polynomial C_i in the variable z,

$$\sum_{j=0}^{p_i - 1} z^{\mu_i(j)}.$$

(2) Compute $C = \bigodot_{i=1}^{r} C_i$.
(3) Compute $C' = \frac{d}{dz} C$.
(4) Compute $C'(1)$.
(5) Compute $D = C'(1)/2^g + (r/2^{g+1}) \cdot P - (r/2^{g+1}) \cdot \sum_{i=1}^{r} P/p_i - (P-1)/2$.

We claim that $D = \sum_{x \in \mathcal{B}} 1$.

Lemma 3.3. *Algorithm 2 computes the bad integer census.*

Proof. Start from equation (8). Calculation shows that

$$\sum_{i=1}^{r} \sum_{x} x_i/p_i = (1/2) \left(P - \sum_{i=1}^{r} P/p_i \right),$$

and $\sum_x x/P = (P-1)/2$. It remains to show that $C'(1) = \sum_x \alpha(x)$.

Consider the WFA, F whose states are pairs of integers of the form $(i, \sum_{j=1}^{i} \mu_j(x))$ for $i = 1, \ldots, r$. The start state is $(0,0)$. We regard $(0, \sum_{j=1}^{0} \mu_j(x))$ as $(0,0)$. The only nonzero entries of $X_{|x|_{p_i}}$ are indexed by the state transition pairs

$$\left(i-1, \sum_{j=1}^{i-1} \mu_j(x) \right), \left(i, \sum_{j=1}^{i} \mu_j(x) \right).$$

The common nonzero entry value is 1. The only nonzero entry of matrix $X_{|x|_{p_1}} \cdots X_{|x|_{p_r}}$ will be indexed by

$$(0,0), \left(r, \sum_{j=1}^{r} \mu_j(x) \right).$$

Now, $\sum_{j=1}^{r} \mu_j(x) = a + 2^g \rho(x)$, where $a < 2^g$ is an integer. V has nonzero entries at states $(r, a + 2^g \cdot b)$, where $b < r$. The entry at such a state is a. We can see that the r-output of F is $\alpha(x)$, and hence $\langle F \rangle_r = \sum_x \alpha(x)$.

We now show that $C'(1) = \langle F \rangle_r$. The idea here recasts the original census approximation algorithm described in [8]. Write the polynomial C in step 2 of the algorithm as $C = \sum_{i=0}^{2^g - 1} a_i \cdot z^i$. In step 4 we obtain $\sum_{i=0}^{2^g-1} i \cdot a_i$. We claim that

$$\sum_{i=0}^{2^g - 1} a_i \cdot i = \sum_x \alpha(x).$$

The coefficient a_i is just the number of $x < P$ such that $\alpha(x) = i$. Indeed, this is just a partial summand computed in the r-behavior of the WFA F described in the previous paragraph, and the lemma follows. □

Theorem 3.4. *Algorithm 2 computes the census of bad integers in*

$$O(n^2 \cdot \log\log(n) \cdot \mathcal{A}(n)/\log n)$$

bitwise arithmetic operations, where $\mathcal{A}(n)$ is the time to multiply two n-bit integers.

Proof. Lemma 3.3 establishes that the algorithm computes the bad census. We proceed to the time complexity. Multiplication of the r polynomials C_i, with reduction of all intermediate polynomial degrees modulo 2^g can be done in time dominated by the product of two degree 2^g polynomials with integer coefficients no larger than P. By Lemma 2.1, $P < 2^{2n}$, so coefficient multiplications involve at most $\mathcal{A}(2n)$ bitwise arithmetic operations. Since multiplication is no worse than quadratic time, we can express this cost as $O(\mathcal{A}(n))$. Using FFT, or similar convolution transform methods, the $O(r)$ pairwise polynomial products can each be done using $2^g \cdot g \cdot \log(g) \cdot \mathcal{A}(n)$ operations. By Lemma 2.1, $r = O(n/\log n)$, and since $2^g < 8r$, this cost is bounded above by $O(n \cdot \log\log(n) \cdot \mathcal{A}(n))$. The overall time follows by multiplying this bound by r. □

3.4. Experimental results

We tabulate representative densities calculated from actual counts for the bad integers over a range of values for r. We point out that on a dedicated 2.66 gHz computer with a 3 gB real address space the running time for $r = 511$ is roughly 70 minutes. For $r = 511$, $\lfloor \log P \rfloor = 5189$. In the interest of sanity, we illustrate the actual output only for $r = 127$ at the end of the paper.

```
r                density


8:               .18753305742133311028
15:              .40060892370174538968
16:              .21566463134239888256
24:              .33901848123361139064
31:              .44741712193144528296
32:              .23146403541827736960
40:              .29357038284339453948
63:              .47250524865958437592
64:              .24014639429784329084
127:             .48573776789735434444
128:             .24481932239434431308
511:             .49622202812798288372
```

The reader can check that as r increases, so does agreement with the asymptotic density expression $r/2^{g-1}$. Ultimately, the graph of density versus r is a “sawtooth” oscillating between 1/4 and 1/2.

For $r = 127$, P is

```
4962053072862893011815686085054943
```

and the bad census is exactly

```
3012820729750787433881999941469665831
```

References

[1] D.J. Bernstein and J. Sorenson, Modular exponentiation *via* the explicit chinese remainder theorem. *Math. Comp.* **76** (2007) 443–454.

[2] A. Chiu, G. Davida and B. Litow, Division in logspace-uniform NC^1. *RAIRO-Theor. Inf. Appl.* **35** (2001) 259–275.

[3] G. Davida and B. Litow, Fast parallel arithmetic via modular representation. *SIAM J. Comput.* **20** (1991) 756–765.

[4] P. Dusart, The kth prime is greater than $k(\ln k - \ln\ln k - 1)$ for $k \geq 2$. *Math. Comp.* **68** (1999) 411–415.

[5] G.H. Hardy and E.M.Wright, *An Introduction to the Theory of Numbers.* Oxford Press, USA (1979).

[6] D. Knuth, *The Art of Computer Programming, Vol. II.* Addison-Wesley (1969).

[7] W. Kuich and A. Salomaa, *Semirings, Automata, Languages.* Springer-Verlag (1986).

[8] B. Litow and D. Laing, A census algorithm for chinese remainder pseudorank with experimental results. Technical Report. http://www.it.jcu.edu.au/ftp/pub/techreports/2005-3.pdf

[9] A. Salomaa and S. Soittola, *Automata Theoretic Aspects of Formal Power Series.* Springer-Verlag (1978).

[10] S.P. Tarasov and M.N. Vyalyi, *Semidefinite programming and arithmetic circuit evaluation.* Technical report, arXiv:cs.CC/0512035 v1 9 Dec 2005 (2005).

[11] I.M. Vinogradov, *Elements of Number Theory.* Dover (1954).

Communicated by C. Choffrut.
Received February 17, 2006. Accepted June 18, 2007.