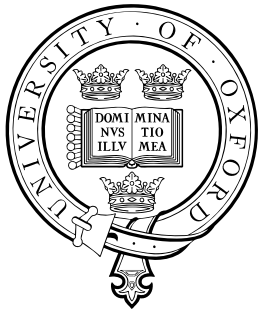
Matching with Mismatches and Assorted Applications



Colin Percival
Wadham College
University of Oxford

A thesis submitted for the degree of
Doctor of Philosophy

Hilary 2006

To the Chemist, the Teacher, and the Composer,
The Medic, the Musician, and the Conductor,
Without whom this would not have happened;

And to the City of Oxford,
For reminding me how short three years can be.

Acknowledgements

In no particular order, I would like to thank:

Graham Percival, for building and repairing computer systems while I've been thousands of miles distant;

Michael Nottebrock, for asking the innocent question on a public mailing list which first made me consider the problem of delta compression;

Robert Muth, for searching through four year old backups to retrieve the "upgrade" corpus used in Table 2.1 and Table 3.1;

My supervisors, for allowing me the freedom to wander between topics until I decided to write *this* thesis, yet being available if I ever wished to discuss anything;

Richard Brent, Richard Bird, and Peter Jeavons for reading and commenting on early draft(s) of this work;

The Commonwealth Scholarship Commission in the United Kingdom, for funding my years in Oxford;

and finally, the dedicatees, who have each helped, in their own way, to keep me sane for the past π years.

Abstract

This thesis consists of three parts, each of independent interest, yet tied together by the problem of matching with mismatches. In the first chapter, we present a motivated exposition of a new randomized algorithm for indexed matching with mismatches which, for constant error (substitution) rates, locates a substring of length m within a string of length n faster than existing algorithms by a factor of $\mathrm{O}(m/\log(n))$.

The second chapter turns from this theoretical problem to an entirely practical concern: delta compression of executable code. In contrast to earlier work which has either generated very large deltas when applied to executable code, or has generated small deltas by utilizing platform and processor-specific knowledge, we present a naïve approach — that is, one which does not rely upon any external knowledge — which nevertheless constructs deltas of size comparable to those produced by a platform-specific approach. In the course of this construction, we utilize the result from the first chapter, although it is of primary utility only when producing deltas between very similar executables.

The third chapter lies between the horn and ivory gates, being both highly interesting from a theoretical viewpoint and of great practical value. Using the algorithm for matching with mismatches from the first chapter, combined with error correcting codes, we give a practical algorithm for "universal" delta compression (often called "feedback-free file synchronization") which can operate in the presence of multiple indels and a large number of substitutions.

Contents

List of Symbols for Chapter 1

log	Natural logarithm
e	Base of natural logarithm
$\pi(x)$	Number of positive prime integers less than or equal to x
Σ	Alphabet
S	String over Σ within which T is being searched
$\bar{S}$	Index generated from S
S'	Substrings of S which approximately match T
n	Length of S
T	String over Σ which is being searched for
m	Length of T
$\delta(x, y)$	Indication of how well symbols x, y match. Usually restricted to $\{0, 1\}$
V	Match count vector; V_i is the sum of $\delta(S_{i+j}, T_j)$
$\bar{V}$	Randomized approximation to V
X	Offsets within S where T approximately matches
$\bar{X}$	Computed estimate of X
t	$\lvert X \rvert$
p	Probability of symbols from S, T matching at offsets in X
k	Permitted edit distance (Section 1.1)
k	Number of projections used
ϵ	Permitted probability of error
L	Minimum useful size of groups onto which projections are made
P	Set of primes in the range $[L, L(1 + 2/\log L))$
p_i	Size of groups onto which projections are made

ϕ_i	Random projections from $\Sigma \to \{-1, 1\}$
$A^{(i)}$	Projection of S via ϕ_i, Z_{p_i} from $\Sigma^n \to \mathbb{R}^{p_i}$
$B^{(i)}$	Projection of T via ϕ_i, Z_{p_i} from $\Sigma^m \to \mathbb{R}^{p_i}$
$X^{(i)}$	Computed approximation of $X \bmod p_i$
$\sigma_{p_i}(n, m, k)$	$\lvert\{(i, j) < (n, m) \colon i \equiv j + k \bmod p_i\}\rvert$
$c_{i,j}$	$\lvert X \cap (j + p_i\mathbb{Z})\rvert$

Preface

This thesis is, in large part, a tribute to the irrepressibility of the human mind. When I started my D.Phil studies in October 2001, I was planning to work on parallel computing — specifically, attacking large computational problems[1] over the Internet using spare computing power, storage, and network bandwidth. I was making steady progress towards this when illness intervened, first sending me to hospital in January 2003, and subsequently making it difficult to concentrate on my work for the following few months.

It was during this period that the first seeds which would later become this thesis were planted. Adopting a Feynmanesque attitude — 'if I can't work, I might as well play' — I turned to a problem which had been plaguing the FreeBSD operating system [18] for years: security patches. Apart from a few abortive attempts, and despite many repeated pleas on the mailing lists, FreeBSD had no existing binary update system; instead, security fixes were distributed as source code patches, and each system administrator would recompile and reinstall the entire operating system. Since this process took several hours, and required manual intervention at half a dozen points, this often meant that important security fixes were ignored entirely.

A few weeks after I finished putting together a working binary security update system for FreeBSD [44], I read an email on a public mailing list, from someone who was unaware of my existing work, asking about the potential for using delta compression in distributing binary security patches; in particular, he asked about using Xdelta [35]. This was something I had been considering, mostly for financial reasons — at 30MB per installation, it wouldn't take many people updating their systems before the bandwidth started costing "real money" — but I hadn't yet found

[1] 2^{40}-element fast Fourier transforms, for example.

time for this. Some 36 hours and 281 lines of code later, however, I was already producing patches 35% smaller than those from Xdelta; over the following month I (while feeling increasingly guilty about neglecting the work I was "supposed" to be doing) made significant further improvements, which I released to the open source community and used in my binary security update system.[2]

Over the second half of 2003, this thesis gradually took form: My investigations into delta compression of executable code led me to a new algorithm for matching with mismatches; with this I made further improvements to my delta compression work; and then while walking around Wadham college in October 2003, I stumbled across the surprising algorithm for "universal" delta compression which forms the core of the third chapter of this thesis. From there onwards, for the duration of 2004, all that remained was the tedious process of filling in details.

If a mathematician is a machine for turning coffee into theorems, a computer scientist is a machine for converting caffeine into algorithms. As with mathematicians and theorems, the output of these machines may bear little resemblance to that which was originally sought, but I hope the reader will find this particular body of output to be both interesting and useful.

[2]In this manner, FreeBSD became the first commodity operating system, by a margin of over a year, to use delta compression for binary security patches. Microsoft Windows added similar capabilities in the summer of 2004 as part of the XP SP2 service pack, and starting with the OS X 10.3.4 → 10.3.5 upgrade, Apple has been using the code I wrote for FreeBSD to reduce the bandwidth consumed by 10.x.y → 10.x.(y + 1) upgrades.

Chapter 0

Introduction

In the first chapter of this thesis, we introduce a novel approach to string matching problems, namely that of projecting vectors onto subspaces of relatively prime dimension. Providing that we are operating on sufficiently random inputs, we demonstrate that these projections can provide us with useful information about the original strings, while (thanks to their reduced length) allowing for much faster computation. In this manner, we construct a new — and vastly improved — algorithm for matching with mismatches, first in its most obvious form, and later in a more computationally efficient manifestation.

This first chapter is not for the faint of heart: It draws on a wide array of results, including the Fast Fourier Transform [13] over non-power-of-two lengths [7], the Chinese Remainder Theorem [30, 54], priority queues, the well-known probability bounds of Hoeffding [24] and Chernoff [12], and even results concerning the distribution of prime numbers [15].

The remaining two chapters concern themselves with applications of the algorithm from Chapter 1, and here the reader will likely find an easier road. In the second chapter, we present a *naïve* method for delta compression of executable code; that is, a method for constructing patches between two versions of a (compiled) computer program, which takes advantage of the structure of such files while remaining entirely platform-agnostic. This is theoretically interesting — the possibility of a naïve method producing patches of size comparable to those produced by a non-naïve method indicates that knowing the instruction set of a processor is rather less useful than one might expect — but is primarily of practical importance: Because this ap-

proach is naïve, it can be applied to any platform desired, allowing highly effective delta compression to be used on esoteric platforms for which it would otherwise be unavailable.

The third chapter extends the concept of delta compression into the concept of *universal* delta compression: Rather than having two files and attempting to construct a patch between them, we have one file and an estimate of the similarity between the two files. To many people, this result is profoundly counter-intuitive — the idea of constructing a "patch" which allows a given file to be reconstructed from any reference file is combinatorially absurd — but the problem is made feasible by the fact that the party applying the patch already has a file which is similar to the target, even if this file is unknown to the party constructing the patch.

We expect that these two applications will be of significant practical importance in the coming years, and that, consequentially, a large proportion of those reading this thesis will be concerned more with the applications than with the beautiful algorithm for matching with mismatches which underlies them. We strongly recommend that these readers start reading at Chapter 2, and take for the moment as given that an algorithm for matching with mismatches exists with the properties required in the later chapters. Even on second reading, those concerned merely with the applications will not need to read all of Chapter 1; of that chapter, the final two sections alone can be taken as presenting a practical algorithm, even if they fail to provide any justification or motivation.

But to those who are more interested in algorithms than applications: Welcome; and let us now begin the first chapter.

Chapter 1

Matching with Mismatches

1.1 Introduction

The problem of approximate matching with respect to edit distance (also known as Levenshtein distance) — that is, given two strings S, T of lengths n, m over an alphabet Σ, $n > m$ (and usually $n \gg m$), to find all substrings[1] S' of S such that S' can be transformed into T via a sequence of at most k substitutions, insertions, and deletions[2] — has been studied extensively [39]. Of particular theoretical interest are "filtering" algorithms which, for low error rates ($k < \alpha m$, for some α depending upon $|\Sigma|$), operate in $\mathrm{O}(n(k + \log m)/m)$ time, which is equal to the proven lower bound for non-indexed algorithms (those which are not permitted to perform any pre-processing of the string S) [11].

The problem of matching with mismatches — a more restricted problem, where the only transformations permitted are substitutions — has received considerably less attention, probably due to its perceived lack of relevance. There are few naturally occurring problems where substitutions occur in the complete absence of insertions and deletions (hereafter, "indels"); however, there are many cases where substitutions are far more common than indels: A visual inspection of a number of related protein sequences suggests that substitutions are 10–20 times as common as indels, while computer programs compiled from closely related source code (e.g., before and after

[1] Some authors use the term "substring" to refer to what is more properly termed a "subsequence". In this thesis, we adopt the more conventional definition that a substring of S is a string $S_i S_{i+1} S_{i+2} \cdots S_j$ for some $i \leq j$.

[2] For large values of k, the number of such substrings grows as $\mathrm{O}(nk)$; we therefore take "finding" the substrings to mean "enumerating the positions i in S where the substrings start".

a security patch) often have substitution rates several thousand times their indel rates. In light of this, it seems clear that the problem of matching with mismatches is worth considering.

The "problem" of matching with mismatches is in fact a number of related problems. Taking $\delta\colon \Sigma \times \Sigma \to \mathbb{R}$ as a function which indicates how closely two symbols match, and defining

$$V_i = \sum_{j=0}^{m-1} \delta(S_{i+j}, T_j),$$

the following problems are frequently considered:

1. Compute V_i for all $0 \leq i \leq n - m$.

2. Given some $k \in \mathbb{R}$, find all integers $i \in [0, n - m]$ satisfying $V_i > k$.

3. Given some $t \in \mathbb{N}$, find values $x_1 \dots x_t$ such that V_{x_i} takes on the t largest possible values.

4. Let $t \in \mathbb{N}$ be given and a set $X = \{x_1 \dots x_t\}$ be fixed but unknown, and suppose that S and T are generated by a random process in such a manner that $E(\delta(S_i, T_j)|i - j \in X) = \chi > \bar{\chi} = E(\delta(S_i, T_j)|i - j \notin X)$ for some constants $\chi, \bar{\chi} \in \mathbb{R}$. Find X with high probability.

The reader will note that problems 1–4 are in decreasing order of difficulty, i.e., given an algorithm for solving problem i, it is easy to construct an algorithm for solving problem $i + 1$. In spite of this, we argue that the last problem is often the most natural formulation to consider: In circumstances where the problem of matching with mismatches occurs, the extent to which two strings match is generally not significant in itself, but rather is a marker for an underlying binary property – for example, having evolved from the same original genomic sequence, or having been compiled from the same source code – and thus it is reasonable to expect alignments of the string T against S to either match *well* or *not at all*[3].

[3]This can be viewed as akin to the difference between gold mining and searching for buried treasure — in both cases, we are looking for gold, but in the latter case, there is a clear distinction between gold and non-gold.

In addition to the four problems stated above, there are a number of variations which can be treated using improved algorithms. Whereas the vector V can be computed for arbitrary δ in $\mathrm{O}(n\,|\Sigma|\log m)$ time by using the FFT [17], it is possible to compute V in $\mathrm{O}(n\sqrt{m\log m})$ time if δ is a zero-one function and $\{(\alpha,\beta)\colon \delta(\alpha,\beta)=1\}$ is an equivalence relation, by treating common symbols with the FFT and uncommon symbols with a quadratic-time algorithm [5]. Even more interestingly, the vector V can be estimated for such a zero-one function δ using a randomized algorithm; taking τ to be an integer parameter between 1 and $|\Sigma|$ as desired, in time $\mathrm{O}(n\tau\log m)$ a vector V' can be randomly computed such that $\mathrm{E}(V'_i)=V_i$ and $\mathrm{Var}(V'_i)\leq(m-V_i)^2/\tau$ [3, 4].

Finally, and as alluded to earlier, each problem may be considered in both *indexed* and *non-indexed* forms, depending upon whether one is permitted to perform some precomputation ("indexing") using the string S before the string T is made available. As we shall see in later chapters, an indexed algorithm can be very useful when attempting to solve many problems with the same S but varying strings T.

In this chapter, we shall consider the last of the problems listed above, with indexing permitted, and with δ a zero-one function representing an equivalence relation as described above.

1.2 Problem statement

In order to allow us to formally prove anything about the algorithms we shall be presenting, we must first give a precise definition of the problem we shall be solving. We define therefore the problem as follows:

> Problem space: A problem is determined by a tuple $(n,m,t,p,\epsilon,\Sigma,X)$ where $\{n,m,t\}\subset\mathbb{N}$, $\{p,\epsilon\}\subset\mathbb{R}$, $m<n$, $0<\epsilon$, $0<p$, $|\Sigma|$ is even, and $X=\{x_1,\ldots,x_t\}\subset\{0,\ldots,n-m\}$ with $x_i\leq x_{i+1}-m$ for $1\leq i<t$.
>
> Construction: Let a string T of length m be constructed by selecting m characters independently and uniformly from the alphabet Σ. Let a string $\hat{S}$ be constructed by randomly selecting n characters independently and uniformly from the alphabet Σ. Let a string S of length n be constructed by independently taking $S_i=T_{i-x_k}$ with probability p if

$\exists x_k \in \{i-m, \ldots, i-1\}$ and $S_i = \hat{S}_i$ otherwise (with probability $1-p$ or 1 depending upon whether $X \cap \{i-m, \ldots, i-1\}$ is non-empty).

Problem (indexing): Given $(n, m, t, p, \epsilon, \Sigma, S)$ generate an (optionally randomized) index $\bar{S}$.

Problem (matching): Given $(n, m, t, p, \epsilon, \Sigma, \bar{S}, T)$, find (optionally in a randomized manner) the set X with probability at least $1-\epsilon$ for each problem.

The construction above produces strings S and T which are in a sense "as random as possible"[4] subject to the requirement that characters from S and T are more likely to match at a set of "good" offsets.

The model given above is quite restrictive, and in practice we do not expect inputs to be constructed in this manner[5], but as with most theoretical models, it has the important and necessary characteristics while still being simple enough to be useful. At the end of this chapter, we shall describe some changes which will make the algorithms we provide more useful for operating on "real-world" data.

Clearly, for it to be possible to compute the set X with probability $> 1/2$, the values V_i for $i \notin X$ must be smaller than the values V_j with $j \in X$. In light of the construction above, the probability of any given pair $(i, j) \in \bar{X} \times X$ not satisfying $V_i < V_j$ is asymptotically given by $\exp(-\mathrm{O}(m))$ with an implicit constant depending upon p and $|\Sigma|$ alone[6], and thus the set X can only be reconstructed with probability $1-\epsilon$ by *any* algorithm if $m = \Omega(\log(n/\epsilon))$, where the implicit constant depends upon p and $|\Sigma|$.

We will give a $\mathrm{O}(n)$ algorithm for constructing $\bar{S}$, and a

$$\mathrm{O}\left(\frac{n \log(nt/\epsilon) \log m}{mp^2}\right)$$

[4]Formally, whereby the entropy is maximized.

[5]In particular, most inputs do not have characters equidistributed across the alphabet: In natural language, certain letters are far more common than others, while computer data formats tend to include large numbers of 0 bytes.

[6]The values V_i and V_j can be seen as the sum of m Bernoulli trials; taking the (asymptotically-correct) normal approximation, we have two normal distributions with means $\mathrm{O}(m)$ apart and standard deviations of $\mathrm{O}(\sqrt{m})$.

randomized algorithm for finding X. To our knowledge, this is the first algorithm for any problem in the field of approximate string matching which is sublinear in n for constant error rate $k/m \approx 1 - p$ and $m = \Theta(n^\beta)$ for some $\beta > 0$.

1.3 Projective searching

It is well known that integers can be reconstructed from their images modulo relatively prime values — in other words, their projections onto groups $\mathbb{Z}_m$ — through use of the Chinese remainder theorem [54]. Reconstructing sets of integers is more difficult, since it is not clear which element of one image corresponds to a given element of another image[7,8]. Nevertheless, at the expense of randomization, and for a small number of images, this is possible.

First, we present a lemma concerning the distribution of prime numbers.

Lemma 1.1. *Let $\pi(x)$ denote the number of primes less than or equal to x. Then*

$$\pi\left(x\left(1 + 2/\log x\right)\right) - \pi(x) \geq \frac{x}{\left(\log x\right)^2}$$

for all $x \geq 5$.

Proof. Based on computations concerning the location of non-trivial zeroes of the Riemann zeta function [9], it has been shown that

$$\frac{x}{\log x}\left(1 + \frac{0.992}{\log x}\right) \leq \pi(x) \leq \frac{x}{\log x}\left(1 + \frac{1.2762}{\log x}\right)$$

for all $x \geq 599$ [15].

Using these bounds and making the substitutions $y = \log x$, $a = 0.992$, $b = 1.2762$ for the benefit of space, we find that for $x \geq 599$,

$$\begin{aligned}
\pi\left(x\left(1 + 2/\log x\right)\right) &\geq \frac{x\left(1 + 2/y\right)}{\log\left(x\left(1 + 2/y\right)\right)} \cdot \left(1 + \frac{a}{\log\left(x\left(1 + 2/y\right)\right)}\right) \\
&\geq \frac{x\left(1 + 2/y\right)}{y + 2/y} \cdot \left(1 + \frac{a}{y + 2/y}\right) \\
&= \frac{x\left(y + 2\right)\left(y^2 + ay + 2\right)}{\left(y^2 + 2\right)^2}
\end{aligned}$$

[7]Indeed, working around this problem is a challenge in a number of number-theoretic algorithms, and for a brief time a related difficulty held up the development of the Number Field Sieve.

[8]One tempting algorithm would reconstruct a set X by reconstructing and then finding the roots of $\prod_{\alpha \in X}(x - \alpha)$, but this takes time cubic in the size of X, and requires a number of images linear in $|X|$, so for our purposes it is interesting but not useful.

$$\begin{aligned}\pi\left(x\left(1+2/\log x\right)\right)-\pi\left(x\right) &\geq \frac{x\left(y+2\right)\left(y^2+ay+2\right)}{\left(y^2+2\right)^2}-\frac{x\left(y+b\right)}{y^2}\\ &= \frac{x}{y^2}\cdot\frac{(2+a-b)y^4+(2a-2)y^3+(4-4b)y^2-4y-4b}{\left(y^2+2\right)^2}\\ &\geq \frac{x}{y^2}=\frac{x}{(\log x)^2}\end{aligned}$$

where the final inequality holds for $y \geq 3.2$, i.e., $x \geq 25$. For $5 \leq x \leq 599$, the result can be verified numerically. □

The constant 2 can be reduced to $1.2842 + \epsilon$ for all x greater than an easily computable $x(\epsilon)$ by the same argument; using a sufficiently strong form of the prime number theorem [30] it can be shown that $1+\epsilon$ suffices for x greater than some bound $x_0(\epsilon)$.

With that preliminary result, we can now provide a bound on the probability that a set of random projections will have, as their intersection, the correct set, by considering the selection of "unlucky" primes which would be necessary for any value $0 \leq y < n$ to be erroneously contained in the intersection.

Theorem 1.1. *Let $n, t, k \in \mathbb{N}$,$L \geq 5$, and $X = \{x_1, \ldots, x_t\} \subseteq \{0, \ldots, n-1\}$ be fixed. Let $p_1 \ldots p_k$ be selected uniformly at random from the set of primes in the interval $[L, L(1 + 2/\log L))$, and let*

$$\bar{X} = \{0, \ldots, n-1\} \cap (X + p_1\mathbb{Z}) \cap \cdots \cap (X + p_k\mathbb{Z}).$$

Then $X \subseteq \bar{X}$, and equality holds with probability at least

$$1 - n\left(\frac{t\left(\log n\right)\left(\log L\right)}{L}\right)^k.$$

Proof. That X is a subset of the given intersection follows, by definition, from the observation that it is a subset of each element in the intersection. We turn therefore to the second part of the theorem, of deriving a lower bound on the probability that equality holds.

Suppose that equality does not hold, and let $y \in \bar{X} - X$. Then $0 \leq y \leq n-1$, and $y \in X + p_i\mathbb{Z}$ for all $i \in \{1, \ldots, k\}$. Consequently,

$$p_i \mid \prod_{j=1}^{t}\left(y - x_j\right) \qquad \forall i \in \{1, \ldots, k\}.$$

Since this product is nonzero and bounded from above by n^t, and the p_i are chosen from the interval $[L, L(1 + 2/\log L))$, we conclude that for any given $y \in \bar{X} - X$, all the p_i must lie within a set of size at most $\log_L n^t = t \log n / \log L$ "unlucky" primes. However, from Lemma 1.1, we know that there are at least $L/(\log L)^2$ primes in the interval $[L, L(1 + 2/\log L))$, so each prime p_i lies within that set with probability at most $t\,(\log n)\,(\log L)\,/L$.

Consequently, each of the n values in $\{0, \ldots, n-1\}$ lies in $\bar{X} - X$ with probability at most $(t\,(\log n)\,(\log L)\,/L)^k$, and the result follows. □

Note that this bound applies independently of the set X. This is an important distinction, since for random sets X, the probability of inaccurate reconstruction can be trivially bounded by nt^k/L^k — this result demonstrates that even for maliciously chosen sets X, we can perform the reconstruction fairly easily. This provides a useful avenue for attacking the problem of matching with mismatches: If we can compute the sets X mod P quickly for arbitrary primes P, then computing X with arbitrarily small probability of error is reduced to the problem of computing the intersection $\bar{X}$.

To see how these sets can be efficiently computed, consider once again the match count vector V, and the restrictions imposed upon it by our formulation of the problem. In most positions, V behaves essentially as Gaussian noise with mean $m\,|\Sigma|^{-1}$ and roughly the same variance; in a few positions (those corresponding to the x_i we wish to find), it "spikes" up to around pm. We have a signal which we wish to find, within a background of noise.

Now consider what happens if we take the natural projection of V from $\mathbb{R}^n$ onto the subspace $\mathbb{R}^{\mathbb{Z}_P}$. The signal remains, although its locations (i.e., the values x_i) are reduced modulo P; and the level of "background noise" increases. Providing that P is large enough — that is, providing that we don't allow the noise level to increase too much — we can find the set X mod P (subject to a given probability of error ϵ) by taking the largest values of this projection.

Algorithm 1.1. *Let S, T, Σ, n, m, p, t, and ϵ be given as specified in the problem description above, with*

$$\frac{16 \log(4n/\epsilon)}{p^2} < m < \min\left(\frac{\sqrt{32n\epsilon}}{t \log n}, \frac{8(\sqrt{n}+1)\log(4n/\epsilon)}{p^2}\right).$$

Then:

1. *Set*

$$k = \left\lceil \frac{\log(2n/\epsilon)}{\log 8n - \log\left(mt \log n\right)} \right\rceil$$

$$L = \frac{8n \log\left(2kn/\epsilon\right)}{mp^2 - 8 \log\left(2kn/\epsilon\right)}$$

$$\hat{k} = \lceil \log n / \log L \rceil$$

$$\bar{X} = \{\}.$$

2. *Compute* $P = \{x \in \mathbb{N} \colon L \leq x < L(1 + 2/\log L), x \text{ prime}\}$.

3. *For* $i = 1 \ldots k$, *pick* $p_i \in P$ *and* $\Sigma_i \subset \Sigma$ *with* $|\Sigma_i| = \frac{1}{2}|\Sigma|$ *uniformly at random, and define* $\phi_i \colon \Sigma \to \{-1, 1\}$ *by* $\phi_i(x) = (-1)^{|\Sigma_i \cap \{x\}|}$.

4. *For* $i = 1 \ldots k$, *compute the vectors* $A^{(i)}, B^{(i)} \in \mathbb{R}^{p_i}$ *where for* $0 \leq j < p_i$

$$A_j^{(i)} = \sum_{\lambda=0}^{\left\lceil \frac{n-j}{p_i} \right\rceil - 1} \phi_i(S_{j+\lambda p_i})$$

$$B_j^{(i)} = \sum_{\lambda=0}^{\left\lceil \frac{m-j}{p_i} \right\rceil - 1} \phi_i(T_{j+\lambda p_i}).$$

5. *Compute vectors* $C^{(i)} \in \mathbb{R}^{p_i}$ *as the cyclic correlations of respective* $A^{(i)}$ *and* $B^{(i)}$

$$C_j^{(i)} = \sum_{r=0}^{p_i} A_{r+j}^{(i)} B_r^{(i)}$$

using the FFT, and compute the set $X^{(i)} = \{j \in \mathbb{N} \colon 0 \leq j < p_i, C_j^{(i)} > mp/2\}$.

6. *For all* $\hat{k}$*-tuples* $(x_1, \ldots, x_{\hat{k}}) \in X^{(1)} \times \cdots \times X^{(\hat{k})}$, *compute the unique* $0 \leq x < p_1 p_2 \ldots p_{\hat{k}}$ *with the given images modulo the respective primes; if* $x < n$ *and* $x \bmod p_i \in X^{(i)}\ \forall i \in \{\hat{k} + 1 \ldots k\}$, *then set* $\bar{X} = \bar{X} \cup \{x\}$. [9]

7. *Output* $\bar{X}$.

[9] With the bounds on m as given earlier, the second part of this step simplifies somewhat since we will always have $\hat{k} = k$. We claim, without proof, that the algorithm is in fact useful for a wider range of values of m, including values for which we might have $\hat{k} < k$, and include this detail here for that reason.

Theorem 1.2. *The output of Algorithm 1.1 will consist of the elements of X with probability at least $1 - \epsilon$; further, if*

$$m \gg \frac{16 \log(4n/\epsilon)}{p^2}$$

and $n \gg 1$, then the algorithm, not including the computation of the vectors $A^{(i)}$, completes in

$$\frac{(16 + \mathrm{o}(1))Cn \log(n/\epsilon) \log \frac{n \log(n/\epsilon)}{mp^2}}{mp^2}$$

time, where C is a constant such that a length-L cyclic correlation can be computed in $CL \log L$ time using the FFT.

Proof. We first observe that the restrictions on m provide that $k = \hat{k} = 2$, and $\sqrt{n} < L < n$. Now defining

$$\sigma_{p_i}(n, m, j) = \left| \{(x, y) \in \mathbb{Z} \times \mathbb{Z} \colon 0 \le x < n, 0 \le y < m, x \equiv y + j \bmod p_i\} \right|,$$

we have

$$\begin{aligned}
\frac{1}{2}\left(C_j^{(i)} + \sigma_{p_i}(n, m, j)\right) &= \frac{1}{2}\left(\sum_{r \in Z_{p_i}} A_{r+j}^{(i)} B_r^{(i)} + \sigma_{p_i}(n, m, j)\right) \\
&= \frac{1}{2}\left(\sum_{r \in Z_{p_i}} \left(\sum_{\substack{0 \le x < n \\ x \equiv r + j}} \phi_i(S_x) \sum_{\substack{0 \le y < m \\ y \equiv r}} \phi_i(T_y)\right) + \sigma_{p_i}(n, m, j)\right) \\
&= \frac{1}{2} \sum_{\substack{0 \le x < n \\ 0 \le y < m \\ x \equiv y + j}} 1 + \phi_i(S_x)\phi_i(T_y) \\
&= \sum_{\substack{0 \le x < n \\ 0 \le y < m \\ x \equiv y + j}} \delta(\phi_i(S_x), \phi_i(T_y))
\end{aligned}$$

for all j where $\delta(\alpha, \beta) = 1$ if $\alpha = \beta$ and $\delta(\alpha, \beta) = 0$ otherwise.

Now consider an individual term $\delta(\phi_i(S_x), \phi_i(T_y))$ from the above sum. If $x - y \notin X$, then from the construction given in Section 1.2, S_x and T_y are independent and uniformly random elements of Σ; consequently $\phi_i(S_x)$ and $\phi_i(T_y)$ independently take uniformly random values from $\{-1, 1\}$. If $x - y \in X$, then the construction given

earlier defines $S_x = T_y$ with probability p, and with probability $1-p$ the values S_x and T_y are independent[10]. Consequently, the values $\frac{1}{2}\left(C_j^{(i)} + \sigma_{p_i}(n,m,j)\right)$ are independent sums of $\sigma_{p_i}(n,m,j)$ Bernoulli trials, and have expected values

$$E\left(\frac{1}{2}\left(C_j^{(i)} + \sigma_{p_i}(n,m,j)\right)\right) = \frac{1}{2}\sigma_{p_i}(n,m,j) + \frac{mp}{2}c_{i,j},$$

where $c_{i,j}$ is the number of elements of X congruent to $j \bmod p_i$.

From the Hoeffding bound [24] we note that

$$\begin{aligned} P_{S,T,\phi_i}(j \in X^{(i)} | j \notin X \bmod p_i) &\leq \exp\left(-2\left(\frac{mp}{4}\right)^2 \sigma_{p_i}(n,m,j)^{-1}\right) \\ &\leq \exp\left(\frac{-(mp)^2}{8\left(\frac{n}{p_i}+1\right)m}\right) \\ &= \exp\left(\frac{-p^2 m p_i}{8(n+p_i)}\right) \end{aligned}$$

and the same bound applies to $P_{S,T,\phi_i}(j \notin X^{(i)} | j \in X \bmod p_i)$. Consequently, using the result of Theorem 1.1,

$$\begin{aligned} P(\bar{X} = X) &\geq 1 - n\left(\frac{t(\log n)(\log L)}{L}\right)^k - P(\exists i \colon X^{(i)} \neq X \bmod p_i) \\ &\geq 1 - n\left(\frac{t(\log n)(\log n)}{8n(\log n)/m}\right)^k - \sum_{i=1\ldots k} p_i e^{\frac{-p^2 m p_i}{8(n+p_i)}} \\ &\geq 1 - n\left(\frac{mt\log n}{8n}\right)^{\frac{\log(2n/\epsilon)}{\log 8n - \log(mt\log n)}} - kLe^{\frac{-p^2 mL}{8(n+L)}} \\ &= 1 - n\left(2n/\epsilon\right)^{-1} - kne^{-\log(2kn/\epsilon)} \\ &= 1 - \epsilon/2 - \epsilon/2 = 1 - \epsilon \end{aligned}$$

as desired. To establish the time bound, we note that once the $A^{(i)}$ are precomputed the time is dominated by the computation of the vectors $C^{(i)}$; this takes $kCL\log L = 2CL\log L$ time. However, we are given that

$$m \gg \frac{16\log(4n/\epsilon)}{p^2},$$

[10]Note that $P(S_x = T_y) = p + (1-p)\left|\Sigma\right|^{-1} \neq p$ — the construction forces the characters to be equal with probability p, but they might randomly happen to be equal anyway.

and consequently

$$L = \frac{8(1+\mathrm{o}(1))n\log(n/\epsilon)}{mp^2}$$

$$kCL\log L = \frac{(16+\mathrm{o}(1))Cn\log(n/\epsilon)\log\frac{n\log(n/\epsilon)}{mp^2}}{mp^2}$$

as required. □

The vectors $C^{(i)}$ can be seen as the projections of approximations $\bar{V}$ to the match count vectors V onto $\mathbb{R}^{p_i}$.

1.4 Searching with fuzzy projections

In Algorithm 1.1, the complexity is determined almost entirely by the minimum value of L necessary for the $X^{(i)}$ to be computed. Computing $X^{(i)} = X \bmod p_i$ is, however, unnecessary; it is sufficient that one can compute some $X^{(i)} = X \bmod p_i \cup Y^{(i)}$, where the size of $Y^{(i)}$ is bounded and its elements are random, as shown by the following theorem:

Theorem 1.3. *Let $n, t, k \in \mathbb{N}$, $L \geq 5$, $\beta \in [0,1)$ and $X = \{x_1 \dots x_t\} \subseteq \{0, \dots, n-1\}$ be fixed. Let $p_1 \dots p_k$ be selected randomly from the set of primes in the interval $[L, L(1+2/\log L))$, let $Y^{(i)} = \{y_{i,1}, \dots, y_{i,\beta p_i}\} \subset \{0, \dots, n-1\} - X$ be selected randomly for each $i = 1, \dots, k$, and let*

$$\bar{X} = \{1, \dots, n\} \cap \left((X \cup Y^{(1)}) + p_1\mathbb{Z}\right) \cap \dots \cap \left((X \cup Y^{(k)}) + p_k\mathbb{Z}\right).$$

Then $X \subseteq \bar{X}$, and equality holds with probability at least

$$1 - n\left(\beta + \frac{t(\log n)(\log L)}{L}\right)^k.$$

Proof. The same argument as used in Theorem 1.1 holds, except that there is an additional probability β of y falling into each of the k sets. □

This provides an obvious direction for improving our algorithm: Rather than computing $X^{(i)} = X \bmod p_i$, which requires that the values $C_j^{(i)}$ for $j \in X \bmod p_i$ are larger than all other $C_j^{(i)}$, it is sufficient to compute $X^{(i)} \supseteq X \bmod p_i$ by taking the $t + \beta p_i$ values of j with the largest values $C_j^{(i)}$. We no longer need the spikes

in the projections of the match count vector to rise entirely above the surrounding noise; it suffices if they are close to the top.

We now present two trivial lemmas:

Lemma 1.2. *The Chernoff bound [12]*

$$P(X > (1+\delta)\mu) < \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right)^{\mu}$$

where $\mu = E(X)$, $\delta > 0$, *and* X *is the sum of independent random* 0-1 *variables is equivalent to*

$$P(X > Y) < \exp(Y - \mu + Y(\log(\mu) - \log(Y)))$$

where Y *is a constant greater than* μ *and* X *is the sum of independent random* 0-1 *variables.*

Proof. Taking $Y = (1+\delta)\mu$,

$$\begin{aligned}
P(X > Y) &= P(X > (1+\delta)\mu) \\
&< \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right)^{\mu} \\
&= \exp(\mu(\delta - (1+\delta)\log(1+\delta))) \\
&= \exp(\mu\delta - \mu(1+\delta)\log(\mu(1+\delta)/\mu)) \\
&= \exp(Y - \mu - Y\log(Y/\mu)) \\
&= \exp(Y - \mu - Y(\log Y - \log\mu)) \\
&= \exp(Y - \mu + Y(\log(\mu) - \log(Y)))
\end{aligned}$$

as required. □

Lemma 1.3. *Let* X *and* Y *be random variables and* $F, G\colon \mathbb{R} \to [0,1]$ *be monotonic differentiable functions such that*

$$\begin{aligned}
P(X \leq \gamma) &\leq F(\gamma) \\
P(Y \geq \gamma) &\leq G(\gamma) \\
F(-\infty) = G(\infty) &= 0 \\
F(\infty) = G(-\infty) &= 1
\end{aligned}$$

Then for any γ_0,

$$P(X < Y) \leq F(\gamma_0) + \int_{\gamma_0}^{\infty} G(\gamma)F'(\gamma)d\gamma$$

Proof.

$$\begin{aligned}
P(X < Y) &= E_Y(P_X(X > Y)) \\
&\leq E_Y(F(Y)) \\
&= E_Y\left(\int_{-\infty}^{Y} F'(\gamma)d\gamma\right) \\
&= \int_{-\infty}^{\infty} P(Y \leq \gamma)F'(\gamma)d\gamma \\
&\leq \int_{-\infty}^{\infty} G(\gamma)F'(\gamma)d\gamma \\
&\leq \int_{-\infty}^{\gamma_0} 1 \cdot F'(\gamma)d\gamma + \int_{\gamma_0}^{\infty} G(\gamma)F'(\gamma)d\gamma \\
&= F(\gamma_0) + \int_{\gamma_0}^{\infty} G(\gamma)F'(\gamma)d\gamma
\end{aligned}$$

□

We now provide a theorem concerning the probability that the "spikes" we seek to find come "close enough" to rising above the level of the noise.

Theorem 1.4. *Let* x_j *be independent random variables (*$0 \leq j \leq n$*), which are respectively the sums of* $\sigma(j)$ *independent Bernoulli trials (where the trials involved in* x_i *are unrelated to the trials involved in* x_j *for* $i \neq j$*). Further let the expected value of* x_j *be* $\mu\sigma(j)$ *for* $1 \leq j \leq n$ *and the expected value of* x_0 *be* $\mu\sigma(0) + \alpha$*. Then for* $\frac{16\alpha^2}{\sigma n} < \beta$, $\frac{-2\alpha^2}{\sigma} < \log\beta < \frac{-1}{2}$,

$$P(\left|\{j \colon 1 \leq j \leq n, x_j - \mu\sigma(j) > x_0 - \mu\sigma(0)\}\right| > \beta n) < 2e^{-\left(\frac{\alpha\sqrt{2}}{\sqrt{\sigma}} - \sqrt{-\log\beta}\right)^2},$$

where $\sigma \geq \max\sigma(j)$.

Proof. From the Hoeffding bounds, we note that for $1 \leq j \leq n$ and $0 \leq \delta \leq 1$,

$$\begin{aligned}
P(x_j - \mu\sigma(j) \geq \delta\alpha) &\leq e^{\frac{-2\delta^2\alpha^2}{\sigma(j)}} \leq e^{\frac{-2\delta^2\alpha^2}{\sigma}} \\
P(x_0 - \mu\sigma(0) \leq \delta\alpha) &\leq e^{\frac{-2(1-\delta)^2\alpha^2}{\sigma(0)}} \leq e^{\frac{-2(1-\delta)^2\alpha^2}{\sigma}}.
\end{aligned}$$

Making the substitution $\hat{\delta} = \frac{1}{\alpha}\sqrt{-\sigma\log(\beta)/2}$ (note that $0 < \hat{\delta} < 1$ from the bounds assumed on $\log\beta$ above), applying the Chernoff bound in the form given in Lemma 1.2, and noting that e^{-cx^2} is convex for $x^2 > \frac{1}{2c}$ provides:

$$\begin{aligned} P(\big|\{j\colon 1 \le j \le n, x_j - \mu\sigma(j) \ge \delta\alpha\}\big| \ge \beta n) &\le e^{n\beta - ne^{\frac{-2\delta^2\alpha^2}{\sigma}} + n\beta\left(\frac{-2\delta^2\alpha^2}{\sigma} - \log\beta\right)} \\ &= e^{n\left(e^{\frac{-2\alpha^2\hat{\delta}^2}{\sigma}} - e^{\frac{-2\alpha^2\delta^2}{\sigma}}\right) + \frac{2n\beta\alpha^2}{\sigma}\left(\hat{\delta}^2 - \delta^2\right)} \\ &\le e^{n\left(\frac{4\alpha^2\hat{\delta}}{\sigma}\beta\left(\delta - \hat{\delta}\right)\right) + \frac{2n\beta\alpha^2}{\sigma}\left(\hat{\delta}^2 - \delta^2\right)} \\ &= e^{-\frac{2n\beta\alpha^2}{\sigma}\left(\delta - \hat{\delta}\right)^2} \end{aligned}$$

for all $\delta > \hat{\delta}$.

For clarity[11] we note that the above is a Chernoffian bound on the (βn)th largest of the values $x_1 \ldots x_n$. If we had $\sigma(1) = \ldots = \sigma(n) = \sigma$, then the "typical" value of the (βn)th largest value would be approximately $\mu\sigma + \sqrt{-\sigma\log\beta/2}$ as this is the value which individual x_i will exceed with probability roughly β; where Chernoff bounded the probability of a random variable diverging from its mean by a value exponential in the square of the divergence, we have done the same to the probability of the (βn)th largest value from a set of random variables diverging from the typical value of the (βn)th largest value (but with a different exponential constant).

Now if we define X by $X\alpha = x_0 - \mu\sigma(0)$, Y to be the largest value such that $\big|\{j\colon 1 \le j \le n, x_j - \mu\sigma(j) \ge Y\alpha\}\big| \ge \beta n$, and

$$F(\delta) = \begin{cases} \exp(-2(1-\delta)^2\alpha^2/\sigma) & \delta \le 1 \\ 1 & \delta \ge 1 \end{cases}$$

$$G(\delta) = \begin{cases} 1 & \delta \le \hat{\delta} \\ \exp(-2n\beta\alpha^2(\delta - \hat{\delta})^2/\sigma) & \delta \ge \hat{\delta} \end{cases}$$

then the conditions of Lemma 1.3 hold.

[11] ... and to use up space on a page which would otherwise be half-empty...

Taking $\gamma_0 = \hat{\delta}$, we therefore have

$$
\begin{aligned}
&P(\left|\{j\colon 1 \le j \le n, x_j - \mu\sigma(j) > x_0 - \mu\sigma(0)\}\right| > \beta n) = P(Y > X) \\
&\quad \le e^{-\frac{2\alpha^2\left(1-\hat{\delta}\right)^2}{\sigma}} + \int_{\hat{\delta}}^{1} e^{\frac{-2n\beta\alpha^2(\delta-\hat{\delta})^2}{\sigma}} \cdot \frac{d}{d\delta}\left(e^{\frac{-2(1-\delta)^2\alpha^2}{\sigma}}\right) d\delta \\
&\quad = e^{-\frac{2\alpha^2\left(1-\hat{\delta}\right)^2}{\sigma}} + \int_{\hat{\delta}}^{1} \frac{4(1-\delta)\alpha^2}{\sigma} e^{\frac{-2\alpha^2}{\sigma}\left(n\beta(\delta-\hat{\delta})^2+(1-\delta)^2\right)} d\delta \\
&\quad = e^{-\frac{2\alpha^2\left(1-\hat{\delta}\right)^2}{\sigma}}\left(1 + e^{\frac{2(1-\hat{\delta})^2\alpha^2}{\sigma(1+n\beta)}} \int_{\hat{\delta}}^{1} \frac{4(1-\delta)\alpha^2}{\sigma} e^{\frac{-2\alpha^2(1+n\beta)}{\sigma}\left(\delta-\frac{n\beta\hat{\delta}+1}{1+n\beta}\right)^2} d\delta\right) \\
&\quad < e^{-\frac{2\alpha^2\left(1-\hat{\delta}\right)^2}{\sigma}}\left(1 + e^{\frac{2(1-\hat{\delta})^2\alpha^2}{\sigma(1+n\beta)}} \frac{4(1-\hat{\delta})\alpha^2}{\sigma} \int_{\hat{\delta}}^{1} e^{\frac{-2\alpha^2(1+n\beta)}{\sigma}\left(\delta-\frac{n\beta\hat{\delta}+1}{1+n\beta}\right)^2} d\delta\right) \\
&\quad = e^{-\frac{2\alpha^2\left(1-\hat{\delta}\right)^2}{\sigma}}\left(1 + e^{\frac{2(1-\hat{\delta})^2\alpha^2}{\sigma(1+n\beta)}} \frac{4(1-\hat{\delta})\alpha^2}{\sigma} \int_{\frac{\hat{\delta}-1}{1+n\beta}}^{\frac{n\beta(1-\hat{\delta})}{1+n\beta}} e^{\frac{-2\alpha^2(1+n\beta)x^2}{\sigma}} dx\right) \\
&\quad < e^{-\frac{2\alpha^2\left(1-\hat{\delta}\right)^2}{\sigma}}\left(1 + e^{\frac{2(1-\hat{\delta})^2\alpha^2}{\sigma(1+n\beta)}} \frac{4(1-\hat{\delta})\alpha^2}{\sigma} \left(\int_{\frac{\hat{\delta}-1}{1+n\beta}}^{0} 1dx + \int_{0}^{\infty} e^{\frac{-2\alpha^2(1+n\beta)x^2}{\sigma}} dx\right)\right) \\
&\quad = e^{-\frac{2\alpha^2\left(1-\hat{\delta}\right)^2}{\sigma}}\left(1 + e^{\frac{2(1-\hat{\delta})^2\alpha^2}{\sigma(1+n\beta)}} \left(\frac{4(1-\hat{\delta})^2\alpha^2}{\sigma(1+n\beta)} + \frac{\sqrt{2\pi}\alpha(1-\hat{\delta})}{\sqrt{\sigma(1+n\beta)}}\right)\right) \\
&\quad \le e^{-\frac{2\alpha^2\left(1-\hat{\delta}\right)^2}{\sigma}}\left(1 + e^{\frac{2}{16}}\left(\frac{4}{16} + \frac{\sqrt{2\pi}}{\sqrt{16}}\right)\right) \\
&\quad < 2e^{-\frac{2\alpha^2\left(1-\hat{\delta}\right)^2}{\sigma}} \\
&\quad = 2e^{-\left(\frac{\alpha\sqrt{2}}{\sqrt{\sigma}} - \sqrt{-\log\beta}\right)^2},
\end{aligned}
$$

as desired, using the fact that $\frac{\alpha^2(1-\hat{\delta})^2}{\sigma(1+n\beta)} < \frac{\alpha^2}{\sigma n\beta} < \frac{1}{16}$ on the third line from the end. □

We now present an improved algorithm which is significantly faster than Algorithm 1.1, at the expense of increased complexity:

Algorithm 1.2. *Let S, T, Σ, n, m, p, t, and ϵ be given as specified in the problem description above. Assume further that:*

$$m < \min\left\{\frac{\sqrt[3]{n^2\epsilon/2}}{t(\log n)^2}, \frac{\sqrt{n\epsilon/2}}{8p^2}\right\}.$$

Then:

1. *Set*

$$k = \left\lceil \frac{\log(2n/\epsilon)}{\log n - \log mt(\log n)^2} \right\rceil$$

$$\beta = \frac{1}{2}\sqrt[k]{\epsilon/2n}$$

$$x = \left(\sqrt{-\log\beta} + \sqrt{\log(4kt/\epsilon)}\right)^2$$

$$L = \frac{2nx}{mp^2 - 2x}$$

$$\hat{k} = \lceil \log n / \log L \rceil$$

$$\bar{X} = \{\}.$$

2. *Compute* $P, p_i, \phi_i, A^{(i)}, B^{(i)}, C^{(i)}$ *as in Algorithm 1.1.*

3. *Let* $X^{(i)}$ *be the set of size* $\beta p_i + t$ *of values* j *for which* $C_j^{(i)}$ *takes the largest values.*

4. *For all* $\hat{k}$*-tuples* $(x_1, \ldots, x_{\hat{k}}) \in X^{(1)} \times \cdots \times X^{(\hat{k})}$*, compute the unique integer* $0 \le x < p_1 p_2 \ldots p_{\hat{k}}$ *with the given images modulo the respective primes; if* $x < n$ *and* $x \bmod p_i \in X^{(i)}\ \forall i \in \{\hat{k}+1 \ldots k\}$*, then set* $\bar{X} = \bar{X} \cup \{x\}$.

5. *Output* $\bar{X}$.

Theorem 1.5. *The output of Algorithm 1.2 will consist of the elements of* X *with probability at least* $1 - \epsilon$.

Further, if

$$\sqrt[3]{n\epsilon^2} \ll mp^2,$$

$n \gg 1$*, and the* $A^{(i)}$ *are precomputed, then the algorithm operates in*

$$\frac{(2 + \mathrm{o}(1))\left(\sqrt{\log(n/\epsilon)} + \sqrt{3\log(t/\epsilon)}\right)^2 Cn \log \frac{n\log(n/\epsilon)}{mp^2}}{mp^2}$$

time, where C *is a constant such that a length-*L *cyclic correlation can be computed in* $CL\log L$ *time using the FFT.*

Proof. We consider first the sets $X^{(i)}$. As in Theorem 1.2, $\frac{1}{2}\left(C_j^{(i)} + \sigma_{p_i}(n,m,j)\right)$ is the sum of $\sigma_{p_i}(n,m,j)$ Bernoulli trials, and has expected value $\sigma_{p_i}(n,m,j)/2$ if $j \notin X \bmod p_i$ and expected value at least $\sigma_{p_i}(n,m,j)/2 + \frac{mp}{2}$ if $j \in X \bmod p_i$.

Taking $\alpha = mp/2$, $\sigma = (n+L)m/L$, we note that

$$\frac{-2\alpha^2}{\sigma} = -x < \log\beta < \log\frac{1}{2} < \frac{-1}{2}$$

$$\frac{16\alpha^2}{\sigma p_i} < \frac{16\alpha^2}{\sigma L} = \frac{4mp^2}{n+L} < \frac{1}{2}\sqrt{\epsilon/2n} \leq \beta,$$

so Theorem 1.4 applies with $n = p_i$. Consequently, for any $j \in X \bmod p_i$ we find:

$$\begin{aligned}
P(j \notin X^{(i)}) &< 2e^{-\left(\frac{\alpha\sqrt{2}}{\sqrt{\sigma}} - \sqrt{-\log\beta}\right)^2} \\
&= 2e^{-\left(\sqrt{\frac{m^2p^2L}{2(n+L)m}} - \sqrt{-\log\beta}\right)^2} \\
&= 2e^{-\left(\sqrt{x} - \sqrt{-\log\beta}\right)^2} \\
&= 2e^{-\log 4kt/\epsilon} \\
&= \frac{\epsilon}{2kt}
\end{aligned}$$

Since $|X| \leq t$, we conclude that $X \bmod p_i \subseteq X^{(i)}$ for all $1 \leq i \leq k$ with probability at least $1 - \epsilon/2$.

We can now apply Theorem 1.3 and (including the probability that the $X^{(i)}$ are incorrect, and noting on the second line that $L > 2n/m$) find that

$$\begin{aligned}
P(\bar{X} \neq X) &< 1 - n\left(\beta + \frac{t\log n \log L}{L}\right)^k - \epsilon/2 \\
&< 1 - n\left(\beta + \frac{mt(\log n)^2}{2n}\right)^k - \epsilon/2 \\
&\leq 1 - n\left(\frac{1}{2}\sqrt[k]{\epsilon/2n} + \frac{1}{2}\sqrt[k]{\epsilon/2n}\right)^k - \epsilon/2 \\
&= 1 - \epsilon
\end{aligned}$$

as desired.

To establish the time bound, we first note that

$$m < \frac{\sqrt{n\epsilon/2}}{8p^2} < \frac{\sqrt{n}}{p^2}$$

$$L = \frac{2nx}{mp^2 - 2x} > \frac{n}{mp^2} > \sqrt{n},$$

so we have $\hat{k} = 2$. Similarly,

$$\frac{n}{mt(\log n)^2} > \frac{n}{\sqrt[3]{n^2\epsilon/2}} = \sqrt[3]{2n/\epsilon},$$

so $k \in \{2, 3\}$.

The only steps in the algorithm which take more than $\mathrm{O}(L)$ time are the computation of the $C^{(i)}$, which takes $CkL \log L$ time using the FFT (by the definition of C), and step 4, which takes $\mathrm{O}((\beta L + t)^2)$ time. However,

$$\begin{aligned}
(\beta L + t)^2 &\leq 2\beta^2 L^2 + 2t^2 \\
&\leq \frac{1}{2}\left(\frac{\epsilon}{2n}\right)^{2/k} \frac{2nx}{mp^2 - 2x} \cdot L + \frac{n\sqrt[3]{2n\epsilon^2}}{m^2(\log n)^4} \\
&\leq (1 + \mathrm{o}(1))\frac{\sqrt[3]{n\epsilon^2}x}{mp^2} \cdot L + \frac{n\sqrt[3]{2}}{m} \\
&= \mathrm{o}(1)\,(xL + L) = \mathrm{o}(kL \log L),
\end{aligned}$$

so the entire algorithm takes

$$\begin{aligned}
(1 + \mathrm{o}(1))kL \log L &= \frac{(2 + \mathrm{o}(1))Cnkx \log L}{mp^2 - 2x} \\
&= \frac{(2 + \mathrm{o}(1))k\left(\sqrt{\frac{1}{k}\log(n/\epsilon)} + \sqrt{\log(t/\epsilon)}\right)^2 Cn \log L}{mp^2} \\
&= \frac{(2 + \mathrm{o}(1))\left(\sqrt{\log(n/\epsilon)} + \sqrt{3\log(t/\epsilon)}\right)^2 Cn \log \frac{n\log(n/\epsilon)}{mp^2}}{mp^2}
\end{aligned}$$

time. □

Essentially, this algorithm trades increased complexity in the process of reconstructing X for shorter FFT lengths p_i; by performing more processing of the vectors $C^{(i)}$, it extracts more information, thereby reducing the necessary signal-to-noise ratio. As we will see in the next section, there is yet more information which can be extracted.

1.5 A Bayesian reformulation

Bayesian analysis combines a "prior distribution" (the distribution expected in the absence of observations) of a set of parameters with the probability that the observations which *were* made *would have* been made given each possible value of said parameters, to obtain a "posterior distribution", which indicates the likelihood of the parameters having any given value. Of particular recent interest, Bayesian analysis has been used to filter electronic mail, by decomposing messages into small pieces (usually words) and using past history to estimate the probability of each piece appearing in spam or non-spam messages [21].

It seems reasonable to consider this approach for filtering "good offsets" (those in X) from "bad offsets" (the rest). We have a well-defined prior distribution — we know that out of $\{0, 1, \ldots n-m\}$ there are t values in X and $n-m-t+1$ values not in X — and we are essentially making k observations about how well the strings match at each offset. Subject to the (false) assumptions that the sum of Bernoulli trials is exactly normal, and that values $j \notin X$ never fall into the same class (modulo p_i) as any values $k \in X$, we would have (noting that $(C_j^{(i)} + \sigma_{p_i}(n,m,j))/2$ is distributed as the sum of $\sigma_{p_i}(n,m,j)$ Bernoulli trials)

$$P(C_j^{(i)} = q\sigma_{p_i}(n,m,j) + x | j \notin X) \approx \frac{e^{\frac{-x^2}{2\sigma_{p_i}(n,m,j)}}}{\sqrt{\pi\sigma_{p_i}(n,m,j)/2}}dx$$

$$P(C_j^{(i)} = q\sigma_{p_i}(n,m,j) + x | j \in X) \approx \frac{e^{\frac{-(x-pm)^2}{2\sigma_{p_i}(n,m,j)}}}{\sqrt{\pi\sigma_{p_i}(n,m,j)/2}}dx$$

and thus, to a first approximation:

$$\frac{P(j \in X | C_j^{(1)} \ldots C_j^{(k)})}{P(j \notin X | C_j^{(1)} \ldots C_j^{(k)})} = \frac{t}{n-m-t+1} \prod_i \frac{e^{\frac{-\left(C_j^{(i)} - pm\right)^2}{2\sigma_{p_i}(n,m,j)}}}{e^{\frac{-\left(C_j^{(i)}\right)^2}{2\sigma_{p_i}(n,m,j)}}}$$

$$\approx \frac{t}{n-m-t} \exp\left(2mp \sum_i \left(\frac{C_j^{(i)} - \frac{mp}{2}}{\sigma_{p_i}(n,m,j)}\right)\right)$$

where j is reduced modulo p_i as appropriate. This hints at a $\mathrm{O}(kn)$ algorithm, by computing $k, p_i, C^{(i)}$ as in Algorithm 1.2, computing the sum

$$\sum_i \frac{C_j^{(i)} - mp/2}{\sigma_{p_i}(n, m, j)}$$

for all j and finding the t largest values.

As noted, however, this fails both theoretically (the distribution is close to, but not exactly, normal), and in practice: Offsets $j \notin X$ will have their "scores" pulled upwards by the proximity (in terms of being congruent modulo primes in P) of elements of X. In particular, if $X = \{r, 2r \dots (t-1)r, x_t\}$, and r is one of the p_i randomly chosen, then the sum above will have larger values at $j \in r\mathbb{Z} - X$ than it has at $j = x_t$; while this is not a problem for *random* sets X, this (and similar constructions) makes it impossible to sufficiently bound the probability of error for arbitrary X.

This problem can be avoided, however, by filtering the vectors $C_j^{(i)}$ somewhat; if we take $D_j^{(i)} = \max(C_j^{(i)}, \delta)$, for some appropriate δ, and consider the sums of $D_j^{(i)}$ instead of $C_j^{(i)}$, then we can guarantee a small probability of error regardless of X. Furthermore, it is not necessary to compute the sum for all j; rather, since we are only interested in the largest t values, we can start by considering the largest elements of each vector $D^{(i)}$ and stop once it is clear that the remaining sums will be less than all of the t largest values found so far.

Algorithm 1.3. *Let $S, T, \Sigma, n, m, p, q, t$, and ϵ be given as specified in the problem description above. Then:*

1. *Set*

$$\delta = \frac{tmp^2 \log n}{n}$$

$$k = \left\lceil \frac{\log(nt/\epsilon)}{\log(1/4\delta)} \right\rceil$$

$$L = \frac{-8n \log\left(\sqrt[2k]{\epsilon/nt} - \sqrt{\delta} \right)}{mp^2 + 8 \log\left(\sqrt[2k]{\epsilon/nt} - \sqrt{\delta} \right)}.$$

2. *Compute* $P, p_i, \phi_i, A_j^{(i)}, B_j^{(i)}$ *and* $C^{(i)}$ *as in Algorithm 1.1, and* $\sigma_{p_i}(n, m, j)$ *as defined in the proof of Theorem 1.2.*

3. *Compute the vectors* $D^{(i)}$ *such that*

$$\exp\left(-D_j^{(i)}\right) = \sqrt{\delta} + \exp\left(\frac{-mp(C_j^{(i)} - mp/2)}{2\sigma_{p_i}(n, m, j)}\right).$$

4. *Using a priority queue, enumerate and output the* t *values of* j *for which* $\sum_i D_j^{(i)}$ *takes the largest values and* $0 \le j < n$.[12]

Theorem 1.6. *Provided that* $\sqrt{n} < L < n$*, the output of Algorithm 1.3 will be the elements of* X *with probability at least* $1 - \epsilon$*. If* $n^2 > L^k$*, the algorithm operates in asymptotic order*

$$\frac{4Cn\log(nt/\epsilon)\log\frac{n\log(nt/\epsilon)}{mp^2}}{mp^2}$$

time, where C *is a constant such that a length-*L *cyclic correlation can be computed in* $CL\log L$ *time.*

Proof. We note that since step 4 finds the values $0 \le j < n$ such that $\sum_i D_j^{(i)}$ takes on the largest values, it suffices to prove that the sum attains larger values at the t values $j \in X$ than at any of the $n - t$ values $j \notin X$.

Using the fact that, for x a sum of n Bernoulli trials,

$$E\left(\exp\left(\frac{\alpha(x - E(x))}{n}\right)\right) \le \exp\left(\frac{\alpha^2}{8n}\right),$$

we note that for $j \in X$,

$$\begin{aligned}
E(\exp(-D_j^{(i)})) &= \sqrt{\delta} + E\left(\exp\left(\frac{-mp(C_j^{(i)} - mp/2)}{2\sigma_{p_i}(n, m, j)}\right)\right) \\
&\le \sqrt{\delta} + \exp\left(\frac{-m^2p^2}{4\sigma_{p_i}(n, m, j)}\right) \cdot \exp\left(\frac{(mp)^2}{8\sigma_{p_i}(n, m, j)}\right) \\
&= \sqrt{\delta} + \exp\left(\frac{-m^2p^2}{8\sigma_{p_i}(n, m, j)}\right) \\
&\le \sqrt{\delta} + \exp\left(\frac{-mp^2p_i}{8(n + p_i)}\right)
\end{aligned}$$

[12] This algorithm is slightly non-trivial but can found in many places, e.g., Knuth[31].

Similarly, for $0 \leq j' < n$, $j' \notin X$, noting that $P(j' \in X \bmod p_i) < \delta$, we have

$$\begin{aligned} E(\exp(D_{j'}^{(i)})) &< \delta \cdot E(\exp(D_{j'}^{(i)}) | x \in X \bmod p_i) + E(\exp(D_{j'}^{(i)}) | x \notin X \bmod p_i) \\ &< \delta \cdot \sqrt{\delta^{-1}} + \exp\left(\frac{-m^2p^2}{4\sigma_{p_i}(n,m,j')}\right) \cdot \exp\left(\frac{(mp)^2}{8\sigma_{p_i}(n,m,j')}\right) \\ &\leq \sqrt{\delta} + \exp\left(\frac{-mp^2p_i}{8(n+p_i)}\right) \end{aligned}$$

Consequently, observing that since since $D_j^{(i)}$ and $D_{j'}^{(i)}$ (for $j \neq j'$) derive from the sums of distinct subsets of the elements of the match count vector V, they will be independent, we have for all pairs $j \in X$, $j' \notin X$,

$$\begin{aligned} P(\sum_i D_j^{(i)} \leq \sum_i D_{j'}^{(i)}) &\leq E(\exp(\sum_i D_{j'}^{(i)} - D_j^{(i)})) \\ &= \prod_i E(\exp(D_{j'}^{(i)}))E(\exp(-D_j^{(i)})) \\ &\leq \prod_i \left(\sqrt{\delta} + \exp\left(\frac{-mp^2p_i}{8(n+p_i)}\right)\right)^2 \\ &\leq \left(\sqrt{\delta} + \exp\left(\frac{-mp^2L}{8(n+L)}\right)\right)^{2k} \\ &= \frac{\epsilon}{nt} \end{aligned}$$

and, noting that there are fewer than nt such pairs (j, j'), we obtain the required probability of error.

To establish the time bound, we note that step 4 can be performed using a heap in time equal to $\log L$ times the number of positions j which must be considered; this number is easily bounded by $L^2n^{-2/k}$ in the same manner as the error bound. □

It is important to note that this filtering relies upon knowledge of p; if the value used is too low, useful "signal" will be filtered away, while if the value used is too high, the probability of obtaining incorrect results will be higher than desired. In some cases, it may be desirable to use a somewhat less powerful but more robust approach, by computing the ranks $r_j^{(i)}$ of the elements $C_j^{(i)}$ within their respective $C^{(i)}$, and considering the sum of $\sqrt{\log p_i - \log r_j^{(i)}}$; alternatively, one may choose to entirely ignore the danger of an increased error rate for worst-case inputs, and work with the $C_j^{(i)}$ directly.

1.6 Final notes

In order to avoid over-complicating the exposition above, we have omitted some details which, while worthy of note, do not effect the main arguments. We present them here, without proof:

1. Since in the above algorithms we always have $m < L \leq \{p_i\}$, it is not necessary to perform the general correlation of two length-L vectors to compute the vectors $C^{(i)}$; rather, it is necessary to perform a length-L correlation of a length-L vector with a length-m vector. This reduces the factor of $\log L$ to a factor of $\log m$, reducing the algorithms presented to run in order

$$\frac{n \log(nt/\epsilon) \log m}{mp^2}$$

   time.

2. In some cases, it may be preferable to compute the number of matching characters directly, rather than applying mappings ϕ_i onto $\{-1, 1\}$. This increases the cost of computing the correlations by a factor of $|\Sigma|$, but reduces the necessary correlation length L by a factor of $|\Sigma|/4$; more significantly, however, it allows for general goodness-of-match functions δ to be used, which is important in some contexts.

3. As in the paper by Atallah *et al.* [3], it is possible to use mappings $\phi\colon \Sigma \to \{\omega^i\}$ instead of mapping onto $\{-1, 1\}$. This has the effect of reducing the necessary correlation length L by a factor of two; but it also makes it necessary to work in $\mathbb{C}$ instead of $\mathbb{R}$, which in most cases will result in the correlation taking (at least) twice as long.

4. We have required that, subject to the requirement of approximately matching in positions given by the set X, the two strings S, T are random and have characters taken uniformly from Σ. In some contexts, this is not reasonable: In particular, DNA sequences often have short strings which repeat a large number of times, while computer binaries often have large regions containing mostly zero

bytes. To resolve these problems, we can borrow a technique which is often used in the field of DNA sequence alignment: If a sequence is repeated several times, the repeating characters can be replaced by "ignore" symbols which ϕ maps to 0. Alternatively, and somewhat better in the context of computer binaries, commonly-occurring characters (such as the "0" byte) can be partially masked, by weighting them (or rather, the mapping ϕ) by some factor depending upon the character frequency. In practice, we find that weighting by the inverse square root of character frequency seems to produce good results[13].

5. If the values n, m, p, t, ϵ are fixed in advance, the string S can be indexed simply by precomputing (or choosing) $L, P, p_i, \phi_i, A^{(i)}$. If, however, some of these parameters are unknown, values $P, p_i, \phi_i, A^{(i)}$ may be precomputed corresponding to values $L_i = 2^{-i}n$ for various i, taking $\mathrm{O}(n \log n)$ time and producing an index of size $\mathrm{O}(n)$; once the necessary parameters are known, the "correct" value of L can be computed and the portion of the index corresponding to the smallest $L_i \geq L$ can be used (at the expense of a slight time cost due to the larger than necessary vector lengths). Indeed, if implementation details make this desirable, specific convenient p_i can be chosen for the index — with the caveat that the most convenient values 2^i are likely poor choices if the source data is computer-generated.

6. In the last algorithm above, the values $\sigma_{p_i}(n, m, j)$ were used. For the values of n, m, L which can occur in the above algorithms, however, we have $\sigma_{p_i}(n, m, j) \approx nm/p_i$, and computations involving $\sigma_{p_i}(n, m, j)$ can be replaced – at a very slight cost in accuracy – by corresponding computations involving nm/p_i.

7. The restrictions placed upon the input parameters, and the values assigned to L, have naturally erred on the side of caution, and practical implementations will

[13] Another tempting possibility is to *deflate* the input strings, and weight each individual character according to the inverse of the length of the string represented by the symbol in which it is found; by virtue of *deflate*'s effectiveness as a compression mechanism, the weights so obtained would most likely be a good estimate of the "information" transmitted by each character, and thus of the importance of it correctly matching.

generally be able to reduce the o(1) term by adjusting the choice of parameters.

8. Given that, in practice, most input strings will be non-random, it may be useful to apply several different mappings ϕ for each prime p_i and add their resulting vectors $C^{(i)}$ together, since this will reduce the probability of choosing an "unlucky" ϕ.

9. In practical applications, the critical element for determining whether these algorithms can be used is the distribution of the values $C_j^{(i)}$. For random inputs (as constructed earlier), these values will be approximately normally distributed apart from outliers corresponding to values in X (these outliers might not necessarily *appear* to lie outside of the obvious distribution); however random or non-random the process is which produces the inputs used, if the values $C_j^{(i)}$ are distributed in an approximately normal manner then the algorithms described here are likely to succeed (although it may be necessary to make L larger by a multiplicative constant in order to compensate for non-random behaviour of the inputs).

With these final notes, we have a practical algorithm for indexed matching with mismatches which operates in sublinear time for constant error rates. In the following chapters, we shall turn our attention to two applications of this algorithm, first in the delta compression of compiled programs, and second in the remote synchronization of files.

Chapter 2

Delta Compression of Executable Code

2.1 Introduction

Historically, delta compression — that is, generating "patches" encoding the difference between two files[1] — has been used most often in the context of program source code. Revision control systems typically keep the most recent version of a file along with reverse deltas, in order that older versions can be reconstructed [55], while updates are normally distributed as forward deltas — better known as patches. These source code patches have advantages beyond that of mere delta compression; they are also human-readable and (usually) commutative and reversible, which makes them especially useful for collaborative software development.

Given that executable files are typically not human-readable, and that people rarely have cause to examine an old version of an executable file, delta compression of executable code is only used for one major purpose: software updating. In the past few years, however, this purpose has become increasingly important: As software security flaws are discovered and exploited increasingly rapidly — often by autonomous worms — it becomes vitally important that end-user systems be kept up to date. In this context, delta compression of software updates has two major advantages; not only can it dramatically reduce transmission costs, but it allows software to be updated more quickly, reducing both the potential for security flaws to be exploited before the affected software *can* be updated, and increasing the likelihood that patches *will* be

[1] Or, equivalently, compressing a file by using a second "reference" file.

applied promptly: With security updates often in the range of 5–10 MB, and many users still connecting to the Internet via modems running at 33.6 kbps,[2] it can easily take half an hour for uncompressed patches to be downloaded, exceeding the patience of most users.

Delta compression of text files has usually been performed with two basic operations: copying and insertion[3]. Typically, lines common to the old and new versions of a file will be copied, while lines which only appear in the new file will be recorded as part of the patch. This works well with text files because changes tend to be localized; a few lines might change from one version to the next, but most of the lines would remain unchanged. Unfortunately, this method fails when applied to executable files: Unlike the situation with text files, when executable files change there are usually differences spread throughout; as a result, only small regions of the old and new versions can be matched together, making it necessary for a large proportion of the new file to be stored in the patch.

One solution to this problem relies upon knowledge of the structure of executable files, and the machine code, of the files' target platform. By disassembling the old and new executable files and comparing the resulting code, it is possible to generate very compact patches [6]; however, since this approach requires a detailed understanding of the target platform, it is both highly complex and completely non-portable.

Throughout this chapter, we shall refer to the "new" and "old" files; these are also sometimes known as the "current" and "reference" files.

2.2 Differences of executable code

To construct compressed deltas of executable files in a portable manner, it is important to consider how executable files change at the byte level from one version to another. We group these changes into three categories: zeroth-order changes, first-order changes, and second-order changes.

[2]Most users have modems capable of running at 56 kbps, but poor quality wiring tends to limit potential speeds.

[3]Some programs are in fact even more restricted, using only *deletion* and insertion; in the context of text files, where blocks of text normally do not change order, this is equivalent.

Zeroth-order changes are those which are innate to the compilation process — changes which take place even when the same source code is compiled twice. While in most cases there are no such changes, some UNIX programs contain human-readable time, date, and host stamps [44], and Microsoft's Portable Executable format [37] contains a timestamp in the file header. These zeroth-order changes can be problematic for some applications — they can result in files being spuriously identified as "modified", causing unnecessary patches to be distributed or setting off intrusion-detection systems — but since they are normally very small relative to the files as a whole, they have little impact on patch generation.

First-order changes are those which can be directly attributed to changes in source code. While there is generally some expansion when compared to the source code delta — when optimizing compilers reorder instructions, they will inevitably interleave modified instructions with unmodified instructions, and the precise allocation of processor registers may be modified over an even larger region — these changes will be localized and of an extent proportional to that of the source code changes. Since the bytes of executable code affected by first-order changes belong to instructions which are essentially new, they are best compressed by well-understood (non-delta) compression algorithms.

Second-order changes are those which are induced indirectly by first-order changes. Every time bytes of code are added or removed, the absolute addresses of everything thereafter, and any relative addresses which extend across the modified region, are changed. As a result, a small first-order change will result in addresses being modified throughout an executable file. This explosion of differences, where a single line of modified source code can cause up to 5–10% of the bytes of executable code in a file to be modified, is what causes conventional delta compression algorithms to perform so poorly on executable files. As a result, efficient delta compression of executable code can be largely considered to be the problem of locating and compactly encoding these second-order changes.

2.3 Matching

Since first-order and second-order changes are fundamentally different in nature, it is important to distinguish between them, so that they can be handled differently. The natural approach to this task is to attempt to match portions of the new file against the old file; differences between closely matched regions should be treated as second-order differences (i.e., as differences between regions of binary code generated from the same source code), while regions of the new file which cannot be matched to any part of the old file should be treated as first-order differences[4].

In conventional delta compression methods, this matching is required to be exact — two strings of bytes match if and only if they are equal. Two commonly used methods used for this matching are suffix trees and hashing; depending upon the application, these are sometimes limited to fixed windows, in order to minimize memory usage.

For matching in executable files, however, exact matching is inappropriate; instead, we want to find code which matches apart from certain mutable elements (such as addresses and possibly register allocations). In platform-specific tools (e.g., [6, 43]), this is easy: The code can be disassembled, the mutable elements stripped out, and the remaining instruction listings compared. This problem becomes similarly easy given access to the source code and a (possibly modified) copy of the compiler used: By having the compiler record the lines of source code responsible for each region of binary code in the two files, the problem is reduced to one of comparing the source code for the two files. If either source code or platform-specific knowledge is unavailable, however, the problem becomes one of matching with mismatches — we wish to find regions from the two files which match in their "opcode" bytes, but not necessarily in the bytes encoding memory addresses or register allocations.

We note that this problem is related to the problem of DNA sequence alignment; in both problems, two strings are given from which approximately matched substrings

[4]Providing that the encodings used for the first-order and second-order differences are reasonably "stable" (i.e., not overly influenced by the presence of a small number of bytes which do not follow the same model as the majority), it not necessary that this matching be exactly correct.

must be found. There is, however, a quantitative difference: In DNA sequence alignment, the percentage of matched base pairs is usually quite high (often 90% or more) while the length of matched regions is quite low (tens of base pairs) due to mutations which add or delete characters; when matching executable code, in contrast, the percentage of matched characters can be lower (often down to 50%) but the matched regions tend to be longer (hundreds or thousands of characters).

A trivial quadratic algorithm can be obtained by dynamic programming (or equivalently, by finding the shortest path through a directed graph); each byte within the new file can be aligned against any position in the old file, or left unmatched. Scores are computed based on the number of matched bytes (bytes in the new file which are equal to those against which they are aligned), the number of mismatched bytes (those which are not equal to the bytes against which they are matched), the number of unmatched bytes, and the number of realignments (positions where two successive bytes in the new file are not aligned against successive bytes in the old file). Unfortunately, such a quadratic algorithm is far too slow to be useful in practice[5]. This algorithm is closely related to the well-known Needleman-Wunsch algorithm for DNA sequence alignment [40] and its derivatives.

2.4 Block alignment

Recognizing that small changes in source code tend to leave large portions of the resulting binaries unchanged apart from internal addresses, we apply the algorithm for matching with mismatches from Chapter 1. Taking the old file S, we construct an index $\bar{S}$ with $k = 2$ and $L = 4\sqrt{n \log n}$ in time linear in the file size[6]. Next, we divide the new file T into blocks of length $\sqrt{n \log n}$, and using the index $\bar{S}$ locate where in the old file each such block matches best.

If we are lucky, every sufficiently large region which matches (with mismatches) between the two files will contain at least one block with the correct alignment; but

[5] In delta compression of text files, $O(n^2)$ algorithms are not uncommon; but in such cases, n is usually on the order of 10^3 *lines*, in contrast to the 10^5–10^7 *bytes* typical of binary files.

[6] The constant 4 is to some extent arbitrary; the FFT and block lengths can be adjusted depending upon the amount of time available, the importance of obtaining correct matchings, and the importance of identifying small matched blocks.

there will almost certainly be blocks which were aligned in incorrect places, and the boundaries between adjacent blocks with different alignment offsets will usually not correspond to the boundaries between the underlying regions.

Consequently, we now "tweak" this alignment: We scan, first forwards, then backwards, through the new file, considering in turn each of the boundaries between aligned blocks. Based on the contents of the new and old files, we move these boundaries in order to reduce the number of mismatches; where a range of possible boundaries would result in the same number of mismatches, we place the boundary on a multiple of the largest power-of-two possible[7]. We also remove blocks which, as a result of this process, shrink below some threshold; this is done because such blocks would be too small to have been found by our algorithm for matching with mismatches, so we have no reason to think that they are correctly aligned.

At this point, most or all large regions which match (with mismatches, but without indels) between the two files should have been found. We now make a final pass through the list of blocks, counting the number of matching characters, and remove any blocks or parts of blocks which fail to match in at least 50% of their characters, on the assumption that if the best matching we can find for a region is that poor, then it probably doesn't match at all, but is instead a completely new region (i.e., a first-order change).

Each application of our algorithm for matching with mismatches takes $\mathrm{O}(L \log L) = \mathrm{O}(\sqrt{n \log n} \log n)$ time, and we apply the algorithm $m/\sqrt{n \log n}$ times, for an operation count of $\mathrm{O}(m \log n)$. The other stages (the initial indexing, and the "tweaking") run in $\mathrm{O}(n + m)$ time, for a total operation count of $\mathrm{O}(m \log n + n)$.

In practice, this method is quite fast — especially on processors with vector[8] floating-point arithmetic[9]. The majority of the time consumed is spent performing single-precision FFTs, and since these are among the first complex operations which

[7]Due to issues involving instruction prefetching and code caches, compilers usually attempt to align blocks of code on 8-, 16-, or 32-byte boundaries.

[8]Or, as it is more popularly known these days, SIMD.

[9]This includes all of the most widely used processor architectures: Intel processors have "SSE", PowerPC processors have "Altivec", and AMD processors have "3DNow", all of which support simultaneous operations on four single-precision floating-point values.

processor (and library) vendors optimize, block alignment tends to perform even better than a theoretical operation count might suggest. Block alignment also performs quite well from the perspective of memory usage: The index, and the resulting alignment, fit within $O(n^{1/2+\epsilon})$ bytes of memory. For pairs of binaries which are quite similar, block alignment also tends to result in very small patches.

2.5 Local alignment

While block alignment is very effective when applied to two binaries with very small source code differences (e.g., security updates), it tends to perform poorly in the presence of more extensive changes. Because it starts by aligning blocks of length $O(\sqrt{n \log n})$ (typically a few thousand bytes), it is incapable of detecting and aligning smaller regions, even if those regions match extremely well.

For these, we return to the methods used for delta compression of binary data files: We search for regions from the two files which match perfectly. First we suffix sort the string $S\#T\#$, where the character '#' is usually called the "EOF" character and is sorted before any characters in Σ; this can be done in $O((n+m)\log(n+m))$ time [34], or even in $O(n+m)$ time [26, 29, 32] with a significantly larger implicit constant. We then compute the longest common prefix vector (in linear time [27]), and then scan through the list of sorted suffices in both directions in order to obtain, for each position in T, the offset within S from which the largest number of matching characters can be found, and the length of this matching substring[10] (also in linear time).

Given these locally-optimal alignments, we can generate a set of alignments for the entire file as follows: Starting at the beginning of the file, we set the "current" alignment to be the offset (between the new and old files) where the string starting at byte zero matches the furthest, and allow the "next" alignment to iterate forwards through our array of locally optimal alignments. Any time that the "current" alignment, extended forward to the end of the matching block associated with

[10] An alternative, and more commonly used approach, involves hashing fixed-length blocks [35, 57], which is faster but often fails to find the longest matching substrings. Suffix trees are also commonly used, but in light of recent algorithmic improvements have no advantages over suffix sorting.

the "next" alignment, contains enough mismatches (typically 8 mismatches is a reasonable threshold), the "current" alignment is output and replaced by the "next" alignment. This produces a list of non-overlapping alignments; starting from these "seeds", we now extend the alignments forwards and backwards to the extent that they continue to match at least 50% of characters.

In practice, the time taken is dominated by the construction of the suffix array, and reasonably good results are produced for pairs of executables produced from significantly different source code; but for binaries with only very small differences, the performance is limited by the inability of locally optimal alignments to reflect large blocks which match with significant numbers of mismatches[11]. This local alignment is also slower than block alignment, and consumes $O(\sqrt{n})$ times as much memory.

2.6 Combining local and block alignment

In order to gain the advantages of both block alignment and local alignment, we return to dynamic programming, but avoid the $O(nm)$ running time which would result from allowing arbitrarily alignments by first pruning the graph using block and local alignment.

The block alignment works as described in section 2.4, with the exception that the final filtering stage is removed: The old file is indexed, the new file is split into blocks which are individually aligned against the old file, and the boundaries between blocks are moved (or removed entirely) in order to maximize the number of matching characters.

The local alignment proceeds up to the point of finding the longest matching substring starting from each position in the new file: The two files are suffix sorted together, the longest common prefix vector is computed, and the suffix array is scanned forwards and backwards.

We now iteratively construct a pruned graph, and find the shortest path through it, as follows: Instead of including nm vertices corresponding to the n positions in

[11] One case where this occurs is in tables of addresses, where there may be over a thousand bytes which mismatch in one or two out of every four bytes (the low order byte(s) of each 32-bit address). Since the largest perfect matches within such a block are only three bytes long, the block will remain entirely undetected by a method which relies only upon locally optimal alignments.

the old file against which each of the m bytes in the new file could be aligned, we include only $64m$. The 64 vertices associated with each position *pos* in the new file comprise: 31 computed from the offsets ([position in old file] minus [position in new file]) associated with the longest matching substrings starting at positions immediately following *pos*; 31 from offsets which have the shortest distances from the origin to position $pos - 1$; one corresponding to the matching predicted by block alignment; and one which is the "not matched" position[12].

At byte zero in the new file, all the vertices are initialized to distance zero; subsequently, the distance from (x, y) to $(x + 1, y + 1)$ is 0 if the corresponding bytes from the old and new files match, and 2 if they mismatch; the distance from the "not matched" position for $pos - 1$ to the "not matched" position for *pos* is 1; and a distance of 20 is assigned to the remaining paths from one step to the next[13].

Once the last byte of the new file is reached, we find the vertex from that final step with the minimum distance, and follow its path backwards; in so doing, we have constructed an alignment of the new file against the old file.

2.7 Delta encoding

Once the old and new files have been matched against each other, we turn to encoding their differences. The matching is easily encoded by listing, in order of the new file, the offsets and sizes of blocks which are copied from the old file and the sizes of new blocks. We encode these integers in little-endian, base-128 format, and use the most significant bit of each byte to identify the most significant byte[14]; we refer to the resulting sequence of bytes as the "control string".

Those regions of the new file which are not matched against any part of the old file (and which, therefore, are believed to be zeroth-order and first-order differences) are extracted and concatenated; this forms the "extra string".

[12]Like many other numbers in this chapter, the values 64 and 31 are chosen simply because they work well without being excessively slow.

[13]These values are obtained by experimental observation, and will not be ideal for all input data.

[14]This commonly used encoding allows for small integers to be expressed compactly, while not imposing any upper limit on the integers to be encoded.

What remains is the problem of encoding the second-order changes identified. As noted before, platform-specific information can make this task much easier; one approach [43] involving complete disassembly of the executables into assembly language removes these second-order differences completely, since upon re-assembly, the new addresses are used. Another approach uses extra data emitted by the compiler to zero all such internal addresses and store them separately [53]; this emasculated file is then delta compressed using well understood mechanisms [25, 52, 56].

In order to efficiently encode the second-order changes in a naïve manner, i.e., without platform-specific knowledge, the original source code, or a modified compiler, we recall three points: First, these changes are primarily the result of addresses (relative and absolute) being modified. Second, references to the same, or nearby, data or code targets will normally be modified in the same manner (code and data tend to be moved around in blocks). Third, data and code targets which are referenced close together have a good chance of being located together (locality of reference). Together, these points suggest that when matched blocks of code from the new and old files are compared, not only will most of the differences occur in addresses, but the numerical differences between the addresses in the new and old files will take on certain values far more commonly than others — in short, those differences will be highly compressible.

Now, without being able to disassemble the executable code into its constituent instructions, knowing that the differences represent a small number of commonly-occurring differences is immaterial in itself; but since computer binaries invariably encode addresses as binary integers, we can obtain a very compressible difference string by simply taking the bytewise differences between corresponding bytes of the new and old files.

The presence of multi-byte integers diminishes the potential compression somewhat, however, since a bytewise subtraction of `1200 - 11FC` yields `0104`, while `1210 - 120C` yields `0004`; to avoid this, we turn to multi-precision arithmetic subtraction, and write the difference in balanced notation (i.e., with digits in the range [-128 ... 127]); thus `1200 - 11FC = 1210 - 120C = (00)(04)`, while `1280 - 11C8 = 12B8 - 1200 = (01)(-48)`. Noting that this depends upon machine byte order,

we compute not one "difference" between each pair of matched regions, but instead four: First, the bytewise differences, which perform reasonably well regardless of machine byte order; second, the Lilliputian multi-precision difference; third, the Blefuscuan multi-precision difference; and fourth, a "correction map" simply containing the values from the new file in each place that the new and old files differ[15]. Each of these four constitutes a "difference string"; we will decide later which one shall be used.

Now, we note that a difference string is in fact a union of two entirely different data sets. It specifies *where* in the executable file addresses have been modified, and it also specifies *by what amount* they have been modified. Just as combining similar data before compression tends to improve compression ratios, combining dissimilar data before compression tends to reduce compression ratios[16], so we split this string into two parts: First, a "difference map", which is of the same length but contains bytes equal to zero or one, depending upon whether the corresponding byte is nonzero, and second, a "non-zero difference string", containing all the non-zero bytes without the intervening zeroes[17]. In this manner, we separate locally repetitive data (as noted before, in any region, the differences tend to take a small number of values repeatedly) from globally structured data (as a result of instruction encoding lengths and the positions within instructions where addresses are encoded, the locations where corrections must be made tend to form distinctive, and compressible, patterns).

The control string, non-zero difference string, and extra string are compressed independently, either with `zlib` [2] (a Lempel-Ziv compressor) or `libbzip2` [51] (a block sorting compressor); in general, the control and non-zero difference strings compress most efficiently with `zlib`, since they contain local repetition, while the extra string compresses best with `libbzip2`, since executable code contains significant structure, which a block sorting compressor can utilize.

[15]We have yet to find any files (not specifically constructed for this purpose) for which the "correction map" results in smaller patch sizes; but it seems a natural item to include.

[16]Birds of a feather compress better together.

[17]The "correction map" is handled slightly differently: It is transformed into a difference map marking all the positions where the two files differ, and a non-zero difference string containing the values from the new file at all these locations.

The difference map is compressed in a different manner: Recognizing that it contains entirely structural redundancies, we first take the Burrows-Wheeler transform [10] of the data, which serves to cluster related data together — in this case, causing the 1s to cluster together; next we enumerate the positions of the 1s, thus forming a strictly increasing sequence; finally, we recursively divide this sequence in half, and encode the value at the midpoint using arithmetic compression.

We now construct the patch file in five parts. The first part is a header containing a "magic" string; a byte specifying the differencing mode used (correction, bytewise, Lilliputian, or Blefuscuan) and the compression method used (none, zlib, or libbzip2) for each of the control, non-zero difference, and extra strings; the sizes of the new and old files; the MD5 hashes [50] of the new and old files[18]; the sizes of the compressed and uncompressed control, non-zero difference, and extra strings; the size of the compressed difference map; and the position of the EOF character after the Burrows-Wheeler transform has been performed. The other four parts, in order, are the compressed control string, the compressed non-zero difference string, the compressed difference map, and the compressed extra string.

We have implemented this method in a tool named 'bsdiff 6'[19]

2.8 Performance

In order to evaluate the delta compression performance of bsdiff 6 when generating patches between two different versions of an executable, we use 15 pairs of DEC UNIX Alpha binaries from the exposition of Exediff, the working of which is specific to that platform [6] (five pairs from that paper are not included here; four of these were artificial, while one pair was unavailable). In Table 2.1 we list for each of these pairs the original size of the new version of the file, both uncompressed and bzip2-

[18]It has recently been shown [58] that it is quite easy to find collisions in the MD5 hash function, but this is irrelevant to the purpose for which we are using it: We include these hashes simply as a safeguard against error, (whether human, in the form of attempting to apply a patch to the wrong file, or computer, in the form of incorrectly applying the patch).

[19]bsdiff versions 0.8 through 4.2 were previously published by the author, and ranged from the traditional copy-and-insert to a method similar to what we have described here, but using only local alignment, and using a more primitive delta encoding. Version 5 never existed except as a descriptor for experimental work which later became version 6.

Program	Original	bzip2	Xdelta	Vcdiff	zdelta	.RTPatch	Exediff	bsdiff 6
agrep: 4.0 → 4.1	262144	114338	14631	10886	7162	5910	3531	4265
glimpse: 4.0 → 4.1	524288	222548	109252	93935	64608	37951	23200	24642
glimpseindex: 4.0 → 4.1	442368	193883	98632	80325	51723	25764	18473	16240
wgconvert: 4.0 → 4.1	368640	157536	75230	60658	38544	20712	15688	12432
agrep: 3.6 → 4.0	262144	114502	80346	79962	63282	58124	41554	44327
glimpse: 3.6 → 4.0	524288	222178	177434	189926	147594	140549	106154	109680
glimpseindex: 3.6 → 4.0	442368	193892	144927	144746	115980	105510	80799	80447
netscape: 3.01 → 3.04	6250496	2396661	1100430	1013581	2519221	351759	284608	212032
gimp: 0.99.19 → 1.00.00	1646592	642725	463878	462588	345385	301879	185962	219684
iconx: 9.1 → 9.3	548864	233056	139409	119510	80017	51195	38121	31632
gcc: 2.8.0 → 2.8.1	2899968	708301	549250	422288	274652	140284	76072	88022
rcc (lcc): 4.0 → 4.1	811008	221826	889	667	373	265	303	187
apache: 1.3.0 → 1.3.1	679936	180708	111421	103611	69895	48033	42038	25927
apache: 1.2.4 → 1.3.0	671744	179369	191920	242292	200511	216867	231357	163249
rcc (lcc): 3.2 → 3.6	434176	155090	84456	76227	52324	34098	22019	22691
Average Compression	100%	36.22%	20.83%	19.66%	19.52%	10.88%	8.41%	7.67%

Table 2.1: Sizes of updates produced by bzip2, Xdelta, Vcdiff, zdelta, .RTPatch, Exediff, and bsdiff 6

compressed, along with the size of patches produced by three freely available binary patch tools (Xdelta [35], Vcdiff [33], and zdelta [56]), a widely used commercial tool (.RTPatch [45]), Exediff, and bsdiff 6[20]. In the interest of a fair comparison, and after communication with one of the authors of that tool (Robert Muth), we re-compressed Exediff's patches with bzip2 rather than gzip where this resulted in a reduction in patch size. The average compression factor shown is computed as the arithmetic mean of the individual compression factors [patch size]/[original size], weighted by the square root of the file size, in order to avoid over-weighting either the largest or the smallest files used[21]. (This can be seen as a compromise between using a constant weight and a weight linear in the file size; it might also be possible to make an argument that this weighting is appropriate by considering the sizes of patches as being produced by random walks, but no such considerations were in fact made in choosing this weighting.)

There is a very clear trend; from the largest (worst) patches to the smallest (best) patches, the order is usually bzip2, Xdelta, Vcdiff, zdelta, .RTPatch, {Exediff, bsdiff 6}, with Exediff and bsdiff 6 being optimal in 7 and 8 cases respectively; overall bsdiff 6 performs slightly better than Exediff. There are two notable exceptions to this trend: First, zdelta performs exceptionally poorly on the netscape 3.01 → 3.04 upgrade; we believe that this results from the large file size (6.5 MB) exceeding the capabilities of zdelta and causing it to effectively revert to regular (non-delta) compression[22].

The second exception is the apache 1.2.4 → 1.3.0 upgrade, where bsdiff 6 was the only delta compressor to produce a smaller "patch" than bzip2. Examining the source code for these two versions of Apache, it is not surprising that there is little similarity for the various delta compressors to grasp: The two versions share less than half of their source code. This is reflected in the patch produced by bsdiff 6: Out of

[20]It would have been useful to compare the performance of other platform-specific tools, but since those tools are both specific to a different platform, and commercially licensed, this was not possible.

[21]Different authors express the performance of compression algorithms in a variety of manners; what we refer to as a "compression factor of 10%", many authors would refer to as a "compression ratio of 10x", while yet others would refer to "90% compression" (presumably on the hypothesis that 100% compression would eliminate the file entirely).

[22]Note that zdelta uses the *deflate* algorithm, which is less effective at compressing executable code than bzip2.

the 671744 bytes in apache 1.3.0, only 263509 were delta compressed; the remaining 408235 were compressed with bzip2 and stored as "extra" bytes.

More important than the above consideration of upgrading between versions ("feature updates"), however, is security updates. These are fundamentally different, in that the source code modifications are typically extremely small — of the first 13 security advisories issued for the FreeBSD operating system in 2004, 5 were corrected by patches of less than ten lines, and only one exceeded 50 lines in length [19, 20]. We take the i386 build of FreeBSD 4.7-RELEASE and a snapshot of the RELENG_4_7 security branch as a corpus for comparison here. In Table 2.2 we show the performance of Xdelta, Vcdiff, zdelta, .RTPatch, bsdiff 6, and bsdiff 6 using block alignment only, grouped according to the file type being patched (Exediff is not considered here, because it is exclusive to a different platform). Note that the 3 'text' files (one C header file and 2 configuration files) all have very small differences; the performance of the various tools here reflects little more than the overhead in their respective patch formats.

For the 79 'binary' files, (69 executables, 7 library archives, and 3 shared object files), the tools perform in the same order as before; in order from the largest to the smallest patches: Xdelta, Vcdiff, zdelta, .RTPatch, bsdiff 6. We note, however, that the gap between best and worst performance is much larger for security patches; while the various tools produced 'upgrade' patches averaging from 7.67% to 20.83% of the original file sizes — a spread of less than a factor of three — the 'security' patches averaged from 1.27% to 9.28% — a difference of more than a factor of seven. This reflects the greater relative importance of second-order changes in the files; where the source code changes — and thus the first-order changes in the executable files — are small, the resulting patch sizes depend almost totally upon the efficiency of encoding second-order changes.

In light of the vastly reduced memory consumption, it is useful to note the delta compression performance of bsdiff 6 when operating in "block alignment only" mode: Given that the patches are on average only 8% larger, this may be a method worth considering further.

File type	# files	Total size	bzip2	Xdelta	Vcdiff	zdelta	.RTPatch	bsdiff 6	block alignment only
Executables (static)	53	15056508	6725178	1779523	1275057	773574	373788	250509	263676
Executables (dynamic)	16	8002788	3399743	1009139	664869	422074	182843	128859	133907
Library archives	7	7600906	1635094	93480	57902	38797	30661	16476	31240
Shared Object	3	1438072	599937	96651	77386	49174	20323	13338	13716
C headers	1	11402	3810	238	66	55	169	126	129
Configuration	2	10183	4407	401	81	62	280	214	313
Total	82	32119859	12368169	2979432	2075361	1283736	608064	409522	442981
Average Compression		100%	38.51%	9.28%	6.46%	4.00%	1.89%	1.27%	1.38%

Table 2.2: Sizes of security patches produced by bzip2, Xdelta, Vcdiff, zdelta, .RTPatch, bsdiff 6, and bsdiff 6 using block alignment only

2.9 Conclusions

By considering how executable files change at the byte level from one version to another, and by utilizing the algorithm for matching with mismatches from Chapter 1 and (optionally) suffix sorting, we can generate patches which are equal to or smaller than those produced by a carefully tuned platform-specific tool. In light of the great advantages of a naïve approach — greater simplicity, less potential for correctness and/or security errors, and portability across platforms — we believe this is a very useful approach to the distribution of software updates in general, and security updates in particular.

Unfortunately, in the context of "open source" software, binary patches have one critical flaw: If the "old" file is not exactly as expected, it will be impossible to correctly apply the patch, meaning the the entire "new" file must instead be transmitted[23]. We will turn to this problem in the next chapter.

[23]By using checksums or cryptographic hashes, failing to correctly apply a patch can be assumed to be detected.

Chapter 3

Universal Delta Compression

3.1 Introduction

As noted at the end of Chapter 2, while delta compression can provide considerable advantages, it also has a significant limitation: Each delta is generated from a pair of "new" and "old" files, and any attempt to apply a patch to the wrong "old" file is very unlikely to result in the correct "new" file being produced. As a consequence, normal delta compression is entirely unusable for updating files which have been modified (or in the case of 'Open Source' software, compiled) on the destination machine; it requires that copies of all published binaries be kept (either intact, or as reverse deltas) in order for future delta compression to be performed[1]; it can be impractically slow to build patches if there are a large number of old versions — in the context of security patches, even adding a couple of hours may be problematic — and finally, even once all the necessary patches have been generated, it is necessary either that each system is examined in order that the appropriate patch may be downloaded (which may be impossible for systems not connected to the Internet, or in other high security environments[2], and is often undesirable even if it is possible), or that several different patches are distributed in a single package — which would severely diminish the advantages of delta compression.

[1]This is a particular concern at present, since delta compression is only starting to become widely used; many vendors find that they would like to use delta compression, but did not keep copies of older binaries. While it would be possible to make a gradual transition towards using delta compression — using deltas whenever possible, and downloading the entire "new" files when necessary — this isn't a particularly useful approach, since the majority of security patches usually affect files which have never been updated before.

[2]Banking systems, for example.

In light of these disadvantages, we consider the problem of *universal* delta compression with respect to a distance function d:

> Problem: Let a distance function d be chosen, let a file S be fixed, and let parameters R (known as the "radius") and ϵ (the acceptable probability of failure) be given. Generate a (possibly randomized) file S', with $\left|S'\right| < |S|$, such that for any file T with $d(S,T) < R$ chosen independent of the randomization used in constructing S', S can be computed from $\{T, S'\}$ with probability at least $1 - \epsilon$.

In short, generate a single patch which can be distributed to everyone, rather than needing to generate individual patches for each old version of the file being updated. Note that the probability ϵ of error arises out of the randomized construction of S' — we are not concerned by the possibility that, given S', one can construct files T within the given radius for which the algorithm fails. Some authors refer to this as a "one-round synchronization protocol" or a "zero-feedback synchronization protocol"; given that a zero-feedback protocol is hardly a protocol at all, we opine that this falls more appropriately into the category of delta compression than that of protocols for file synchronization[3].

While there are theoretical solutions to this problem involving colouring hypergraphs [41], these require exponential computational time; as such, they are of no practical value.

Throughout this chapter, we will refer to the "sending machine" and the "receiving machine", these being the parties holding the strings S and T respectively.

3.2 Error correcting codes

If the distance function d is the Hamming distance, there is a well known algorithm for this problem [1]: Take some error correcting code C with the property that

[3]Nevertheless, there is a natural way of converting an algorithm for universal delta compression into a protocol for file synchronization: One participant computes and transmits deltas S' computed with increasing values of R, and the second participant attempts to apply these patches in turn and sends a single bit back after each patch to indicate if the most recent patch was sufficient. (We assume that a checksum can be transmitted in order that the second participant can recognize if S has been reconstructed correctly.)

$C(x) = x.P(x)$ for some parity function P, with distance at least $2R+1$, and compute and send the parity block $S' = P(S)$. At the receiving machine, on receipt of $P(S)$, form the string $T.P(S)$ and decode according to the error correcting code. Given that $|\{i\colon S_i \neq T_i\}| \leq R$ and the code C has distance at least $2R+1$, $T.P(S)$ will be decoded to the codeword $S.P(S)$, from which the string S is extracted.

In contexts where the transmission of the parity block is not subject to errors (as is the case when constructing compressed deltas — we trust patch integrity to lower level mechanisms), this can be improved slightly if we restrict ourselves to using *cyclic* codes: Rather than taking S as the data block of an error correcting code and transmitting the parity block, we can take S as a codeword with errors, and compute and transmit $S' = \text{syndrome}(S) = S(x) \bmod g(x)$, where $g(x)$ is the generator polynomial for the code. Since $S - S'$ is a codeword[4], we can compute and decode the string $T - S'$ to find that codeword, and then add S' to retrieve S. This reduces the problem of error correction from one with codewords of length $|S| + \left|S'\right|$ to one with codewords of length $|S|$.

For computer files, which are naturally interpreted as a sequence of symbols in $GF(2^8)$, an obvious choice of codes is the Reed-Solomon codes [48]: Operating on symbols from $GF(2^n)$, a Reed-Solomon code exists of length $2^n - 1$ with distance $\delta + 1$ and δ parity symbols for any $0 \leq \delta < 2^n - 1$. If the necessary block length is not one less than a power of two, these codes can easily be shortened (effectively by padding with zeroes) to provide any desired length; this construction is used quite commonly, most notably in the error correcting codes used on CD-ROM disks.

If S can be formed from T by a sequence of N_s substitutions of total length L_s bytes (e.g., if $S =$ *"ABCDEF"* and $T =$ *"ABCXYF"* then N_s and L_s can be taken to be 1 and 2 respectively) and if M bytes are grouped into each symbol, then the distance $d(S,T)$ can be bounded from above by $\min(L_s, 2N_s + (L_s - 2N_s)/M)$ by considering the number of blocks which can be corrupted by a sequence of k consecutive errored symbols. Consequently, since the Reed-Solomon codes need a length-$2d$ parity block to correct d errors, and each symbol is M bytes in length, a Reed-Solomon code can correct N_s substitutions of total length L_s bytes by transmitting

[4]Note that although $S - S'$ is a codeword, it is typically not the *closest* codeword to S.

$\min(2L_sM, 2L_s + 4N_s(M-1)) = \min(2L_s\lceil\log_{256}|S|\rceil, 2L_s + 4N_s(\lceil\log_{256}|S|\rceil - 1))$ bytes. Comparing this to the trivial asymptotic combinatorial lower bound (for $1 \ll N_s \ll L_s \ll |S|$) of

$$L_s + \log_{256}\left(\binom{|S| - L_s + 1}{N_s} \cdot \binom{L_s - 1}{N_s - 1}\right) \approx L_s + N_s \log_{256}\left(\frac{L_s(|S| - L_s)e^2}{N_s^2}\right)$$

obtained by considering the number of ways that N_s blocks of total length L_s can be located within a total length of $|S|$ and using the approximations $\log n! \approx n\log(n) - n$ and $(1 + 1/n)^n \approx e$, we note that this performs fairly well for $N_s \ll L_s$ and for small files, but transmits considerably more data than necessary when the changes occur in the form of isolated bytes in a large file.

3.3 Randomized large-codeword error correction over $GF(2^8)$

Where there are a significant number of isolated byte-substitutions, the above approach can be improved, at the expense of randomization and a small probability of error, as follows: Rather than considering the entire string S as a codeword in a Reed-Solomon code over $GF(2^k)$ (for strings of up to $(2^k - 1)\lfloor k/8 \rfloor$ bytes) and performing the error correction in a single step, we first apply an error-*reducing* code. Dividing the string into $\lceil |S|/255 \rceil$ random distinct subsequences[5] of length 255 (this can be done in a variety of ways, but the simplest approach is to construct a random permutation function ϕ — for example, by generating $|S|$ random numbers and sorting them — and taking the sequences $\phi(255i) \ldots \phi(255i + 254)$), the sending machine computes and transmits a $2R$-byte Reed-Solomon syndrome for each subsequence. Given these, the receiving machine uses Reed-Solomon decoding over $GF(2^8)$ to attempt to correct the substitutions. For each 255-byte random subsequence, if R or fewer substitutions were present, the resulting codeword will be correct; otherwise, at most R new errors

[5]Naturally, the selection of distinct subsequences would be performed in a pseudo-random manner, with the initial PRNG state either fixed or transmitted as part of the error-correcting block. The purpose of this randomization is simply to ensure that "bad" inputs, where the errors are placed in a deliberately inconvenient manner, are not likely to be encountered.

will have been introduced by the "decoding"[6]. As a result, providing that R is chosen sufficiently large, the number of errors left after this randomized decoding will be significantly reduced; a single large-field Reed-Solomon code can then correct the remaining errors.

Suppose that strings S, T of length n differ by N_s substitutions (in any positions). Then each random 255-byte subsequence of T will contain an average of $\mu = 255N_s/n$ bytes which differ from the corresponding bytes in S. Suppose further that $n > 2^{16}$ and $2\exp(-\mu(2\log(2) - 1)) < 1/2 - 1/\log_{256} n$ (for our purposes, we require that the number of errors is at least 2–5% of the file length; if there are fewer errors, then we take $R = 0$ and proceed directly to the error-correction stage, since the error-reduction would be counterproductive). Then for some appropriate $0 < \delta < 1$, let $R = \mu(1 + \delta)$ and compute syndromes as above. These will allow for individual 255-byte subsequences to be decoded with a probability of error less than the Chernoff bound of $\exp(\delta\mu)(1+\delta)^{-R}$, which for $\delta < 1$ is less than $\exp(-\mu\delta^2(2\log(2)-1))$. Consequently, the number of errors remaining after this first phase will be $2N_s \exp(\delta\mu)(1 + \delta)^{-R} + \mathrm{O}(\sqrt{n})$ (where the "implicit constant" can be effectively computed given a desired maximum probability ϵ of errors remaining), and (by our assumed bound above) the second error-correcting phase can correct these while still achieving a reduction in the number of bytes transmitted.

We note that these bounds, while algebraically complex, can be computed very efficiently numerically; in practical scenarios, one would take the values n, N_s, and compute for each $R = 0, 1, \ldots, 127$ the probability of any individual 255-byte sequence being decoded correctly, and thereby the size of the syndrome necessary in the second phase in order to obtain the desired probability of error.

To take a simple example, suppose we wish to correct a string of length $|S| = 10^6$ bytes having $L_s = N_s = 10^5$ errors, and we are permitted a probability $\epsilon = 10^{-3}$ of error. Using a single Reed-Solomon code, we would need to operate over $GF(2^{19})$ (since we can group two bytes into each symbol), and we would need a parity block of $2 \cdot 10^5$ symbols, thus we would need to transmit 475000 bytes. If we instead divide

[6]In fact, for large values of R, codewords with too many errors are most likely to simply be returned as "undecodable", since they are likely to not fall within an R-error radius of any codeword.

S into 3922 random subsequences of length 255 (padding the final subsequence with zero bytes), and take $R = 37$ (i.e., send a syndrome of 74 bytes for each 255-byte subsequence), then each individual subsequence can be correctly decoded with probability 99.14%, and the number of remaining errors is less than 4359 with probability $1 - \epsilon$. Now we can send a much smaller parity block over $GF(2^{19})$, and correct these remaining errors using 20701 bytes, for a total of 310929 bytes transmitted, less than the original requirement of 475000 (at the expense of potentially getting the wrong answer if the error-reducing stage left too many errors remaining).

3.4 Rsync revisited

The rsync algorithm, patented in 1995 [46, 47] and independently invented and popularized[7] by Andrew Tridgell [36, 57] is a two-round protocol for file synchronization[8]. For a parameter B, known as the block size and typically in the range of 300–700 bytes, the receiving machine computes and transmits checksums of bytes iB up to $(i + 1)B - 1$ of T for all $0 \leq i < |T| / B$. The sending machine, upon receiving these checksums, computes the checksums of bytes i up to $i + B - 1$ of S for all $0 \leq i < |S|$ and searches for collisions between the two lists. The sending machine then encodes S as a sequence of raw bytes and blocks from T and transmits this, allowing the receiving machine to reconstruct S[9]. The key observation which makes this algorithm practical is that a "rolling" checksum can be used, allowing the checksums of $|S|$ blocks of length B to be computed in $O(|S|)$ time, combined with a stronger checksum[10] which is only computed if the rolling checksum matches. This makes it difficult to construct files which "break" rsync, although it is still easy to

[7] The name of the algorithm comes from the free implementation written by Andrew Tridgell.

[8] In his PhD thesis [57], Andrew Tridgell refers to the patent [46] and describes it as "a somewhat similar algorithm"; however, I have been unable to discern how rsync differs from the algorithm described in that patent. Having no legal expertise, I naturally cannot comment on the validity of the patent, the claims it makes, *et cetera*.

[9] In essence, the receiving machine sends an index to the sending machine, which then uses this in constructing a binary patch; however, due to the large block size (typically 300–700 bytes), the resulting patches are much larger than those constructed by other methods.

[10] Here an unfortunate implementation decision was made in rsync: The "strong checksum" chosen is the MD4 [49] algorithm, which is now considered to be entirely broken [14, 58]. Although the author maintains that the cryptographic security is irrelevant [57], we believe this assessment is rather optimistic, since the existence of files which rsync is *unable* to synchronize breaks one of the widely-held assumptions about the tool.

construct files which force rsync to take $O(B\,|S|)$ computational time. While such files are unlikely to occur in practice, they pose a security risk, since anyone who can convince an rsync server to distribute such a file can thereafter execute a very easy denial-of-service attack against the server, simply by using rsync to fetch that file[11].

To see the relevance of rsync to our problem, we first convert it into a three-round protocol known as "reverse rsync", which was patented [22, 23] and independently reinvented a couple of years later [8]. Under this protocol, the sending machine first splits S into blocks of length B and computes hashes for each of them (these can be precomputed and stored for later use); these hashes are transmitted, and the receiving machine now computes and compares $|T|$ checksums against this list. Any blocks for which the receiving machine can identify matching blocks from T are copied; a list of the remaining (missing) blocks is sent back. Finally, the sending machine, having received this list of blocks, sends the appropriate bytes. While this has the disadvantage of requiring three rounds rather than two, it has several significant advantages: First, some checksums can be precomputed and stored on disk; second, the most cpu-intensive part of the protocol — comparing the checksums received to the local file — is moved to the client, dramatically increasing the number of connections which can be handled by a single server; and third, the final part of the protocol — sending the requested blocks of S — is in fact a subset of the HTTP/1.1 protocol [16], which provides for a "Content-Range" header allowing individual parts of a file to be requested. Consequently, the "reverse rsync" protocol can be implemented at the sending side by generating a file containing the block checksums, and thereafter using any of the many well-tested, efficient, and secure HTTP servers available.

This three-round protocol can be converted into an algorithm for universal delta compression as follows: The checksums are sent as before, but rather than waiting for the receiving machine to provide a list of missing blocks, the sending machine also computes and sends the parity block for S for some convenient cyclic error correcting code. Upon receiving these two components, the receiving machine decodes as follows: Using the checksums as before, it finds blocks in T and places them in the appropriate

[11]It is always possible to execute a denial-of-service attack through imposing heavy load on a server by making a large number of requests; such attempts are limited, however, by the attackers' available bandwidth. This attack is far more serious because it does not require significant bandwidth.

positions; once these are in place, the parity block is appended, the missing blocks are labeled as "erasures", and the error correcting code is decoded, filling in all the gaps between the matched blocks. Since the block size B is much larger than $\log |S|$, this error correcting can be done with a shortened Reed-Solomon code over $GF(256^B)$ in a number of bytes equal to the number of bytes not matched from T. For reasons of efficiency, each B-byte block would best be split into smaller blocks — decoding a length $|S|/B$ Reed-Solomon codeword over $GF(256^B)$ is far slower than decoding B/k independent length $|S|/B$ Reed-Solomon codewords over $GF(256^k)$ — but this does not affect the size of deltas produced[12].

For typical files, there is however a difference between the bandwidth used by this one-way rsync and the traditional algorithm: When transmitting the unmatched blocks, rsync normally applies *deflate* compression, which often reduces the bandwidth used by 50–60%, while the error correcting parity block (or syndrome) will in almost all cases not be compressible in such a manner. This effect could be somewhat mitigated by splitting S not into blocks of constant length B, but rather into blocks which can be individually deflated to the same size. The error correcting code would then be applied to these deflated blocks, and S would be reconstructed by re-inflating. While an interesting possibility, this would probably be too complex to be of practical value.

Suppose that S can be formed from T by a sequence of N_s substitutions of total length L_s, N_i insertions of total length L_i, and N_d deletions (of any length). Then out of the $|S|/B$ length-B blocks in S, at most $\min(L_s, 2N_s + (L_s - 2N_s)/B)$ are not in T as a result of substitutions; at most $\min(L_i, 2N_i + (L_i - 2N_i)/B)$ are unmatchable due to insertions; and at most N_d are unmatchable due to deletions. For $N_s, N_i, N_d \ll |S|$, the expected number of mismatching blocks is approximately $N_s + N_i + N_d + (L_s + L_i)/B$. Consequently, the transmitted parity block should be of length at least $(N_s + N_i + N_d)B + L_s + L_i$ bytes in order to accommodate a typical placement of insertions and substitutions, and if worst-case placement of the

[12]Theoretically, operations in $GF(2^k)$ can be performed in $\mathrm{O}(k^{1+\epsilon})$ time using the FFT; but for k less than several hundred the best implementations are either $\mathrm{O}(1)$ after computation of a lookup table of size 2^k (for k up to around 20) or $\mathrm{O}(k^2)$ time (for k from around 20 up to several hundred).

insertions and substitutions is to be accommodated, the parity block should be of length at least $(2N_s + 2N_i + N_d)B + L_s + L_i$ bytes.

The checksum block needs to contain checksums for $|S|/B$ blocks. In order to avoid excessive computation, the rolling checksum should be of length at least $\log_2 |T|$ bits; this will keep the computation of the strong checksum to a reasonable time limit for random files. The strong checksum should contain an additional $\log_2(|S|/(B\epsilon))$ bits, where ϵ is the accepted probability of failure (due to accidental hash collision); together, these two hashes consume $\log_2(|S|\,|T|/(B\epsilon))$ bits, for a total of

$$\frac{|S|}{B} \log_{256} \left(\frac{|T|\,|S|}{B\epsilon} \right)$$

bytes in the checksum block. We note that the danger of malicious inputs causing an increase in the required computational time can be reduced somewhat by using a keyed checksum function (and including the key in the patch file); more significantly, however, the computational burden is entirely at the receiving side, so there is no potential for denial of service.

Making the substitution $N = N_s + N_i + N_d$ for the benefit of clarity, if we take

$$B = \sqrt{\frac{|S|}{N} \log_{256} \left(\frac{|T| \sqrt{|S|\,N}}{\epsilon} \right)}$$

then we can construct a patch of size

$$L_s + L_i + \sqrt{2\,|S|\,N \log_{256} \left(\frac{|T|^2\,|S|\,N}{\epsilon^2} \right)}$$

which will apply to *most* files T which differ from S by at most N_s substitutions of total length L_s, N_i insertions of total length L_i, and N_d deletions. With a patch of size

$$L_s + L_i + 2\sqrt{|S|\,N \log_{256} \left(\frac{|T|^2\,|S|\,N}{\epsilon^2} \right)}$$

bytes and a random checksum function, *any* T (even those with inconveniently placed edits) within that radius can be "patched" with probability at least $1-\epsilon$; the extra factor of $\sqrt{2}$ arises from the possibility that the insertions and substitutions are located in such a manner as to maximize the number of blocks modified. For random inputs,

or if a random keyed hash is used, this operates in $O(|S|^{1+\epsilon})$ time if fast algorithms are used, or $O((|S|\,N + (L_s + L_i)\sqrt{|S|\,N})^{1+\epsilon})$ time if classical algorithms are used; in both cases, the implicit constants are much larger on the receiving side than on the sending side, since for the Reed-Solomon codes (which dominate the running time) decoding is significantly more expensive than encoding. Practical implementations may find a different (and faster) choice of error correcting code more suitable.

It is interesting to contrast this against the previous section; while error correcting codes, by themselves, are quite efficient at correcting substitutions but are entirely unable to correct insertions or deletions, this "one-way rsync" approach is able to correct insertions, deletions, and substitutions equally well. For many files (e.g., text, source code, archives) this is sufficient; but some files types, such as executable code, tend to have far more substitutions than indels, and for such files neither of these two approaches will produce very good results.

3.5 Block alignment

To fill this gap — that is, to construct universal patches for files which contain some indels, but differ most often by substitutions — we turn once more to the randomized algorithm for matching with mismatches presented in Chapter 1. In addition to the asymptotically superior computational running time of the algorithm, this algorithm has another very important advantage: It operates using an index of size only $O(n \log(n)/m)$ floating-point values.

In order to take advantage of this, we proceed as follows: Taking B as fixed for the moment, we compute $L = 4\,|S| \log |S|/B$. Following the approach taken in Chapter 1, we select random primes p_1, p_2 in the interval $[L, L(1 + 2/\log L))$, and random mappings $\phi_1, \phi_2 \colon GF(256) \to \{-1, 1\}$. In "real-world" usage, the strings S, T are most likely non-random; consequently we "mask" characters which occur more often than others: We count the number of times n_x that each byte value x appears in S, and form $\phi_i'(x) = \phi_i(x)/\sqrt{n_x}$. We now apply these two mappings and projections and compute $A^{(1)} \in \mathbb{R}^{Z_{p_1}}$, $A^{(2)} \in \mathbb{R}^{Z_{p_2}}$ with $A_j^{(i)} = \sum_k \phi_i'(S_j + kp_i)$. Since each $A_j^{(i)}$ is the sum of only $|S|\,/p_i + O(1)$ terms, each coming from a single

character, we don't need to transmit these with arbitrary precision[13]; instead, we find the minimum and maximum values, and scale and round all the $A_j^{(i)}$ to integers in the range $[0, 2^b)$, where $b = \lceil \log_2(|S|/L)/2 \rceil$[14]. The sending machine now transmits $\bar{S} = (p_1, p_2, \phi_1, \phi_2, A^{(1)}, A^{(2)})$ using

$$\frac{|S| \log |S| \log_2(\frac{B}{4 \log |S|})(1 + \mathrm{o}(1))}{2B}$$

bytes. Note that, for any given block size B, this index is larger than the checksum block needed by rsync by a factor of

$$\frac{4 \log |S| \log(\frac{B}{4 \log |S|})}{\log(\frac{|T||S|}{B})}$$

— typically around a factor of ten — but has the advantage of being able to match blocks containing substitutions.

Upon receipt of this index, the receiving machine performs length-p_i forward FFTs on the $A^{(i)}$[15], splits T into blocks of length B bytes, and for each block computes the vectors $B^{(i)}$, their Fourier transforms, and thereby the vectors $C^{(i)}$. Using priority queues, the values $0 \le j < n$ are computed for which $C_j^{(1)} + C_j^{(2)}$ is maximized for each block, indicating that each particular length B block of T probably approximately matches S starting at offset j. Furthermore, the sums $C_j^{(1)} + C_j^{(2)}$ for each block indicate roughly how well the blocks match — not perfectly, but accurately enough to distinguish between a block from T which approximately matches in S and a block from T which does not exist anywhere in S (and therefore obtains an essentially random value j).

After computing where each length B block from T best matches against S, we sort these matchings according to their positions in S[16]. Where two blocks overlap, we compare the values $C_j^{(1)} + C_j^{(2)}$ which were computed for the two overlapping blocks; the block with the higher score is retained intact, while the other block is

[13]Recall that, due to the masking used, these values are no longer integers.

[14]This range ensures that the rounding errors are of the same magnitude as the contribution from a single character in S, i.e., negligible.

[15]This pre-computation is naturally faster than recomputing the forward FFT each time we wish to compute the convolution of $A^{(i)}$ with $B^{(i)}$.

[16]This allows us to take advantage of blocks which have been re-ordered; if we are concerned solely with edit distance (which does not permit re-orderings) then this step may be omitted.

shortened (to start or end where the higher scoring matching finishes) or entirely removed (in the case of low-scoring blocks which are completely covered by other blocks). As a result of the parameters chosen ($k = 2$, $L = 4\,|S| \log |S|/B$), any of the length-B blocks from $|T|$ which do not contain any indels (with respect to S) and contain at most $B/2$ substitutions will be found. As such, placing these blocks in their respective positions will construct a string which differs from S by at most $N_d + N_i + N_s$ substitutions of total length $(N_d + N_i)B + L_i + 2L_s$, since at most $N_d + N_i$ blocks are lost due to indels.

The sending machine now transmits a Reed-Solomon parity block containing $2((N_d + N_i)B + L_i + L_s)/M + 2N_s$ symbols (i.e., capable of correcting half that number of errors) over $GF(256^M)$, where $M = \lceil \log_{256} |S| \rceil$; this requires

$$2(N_d + N_i)B + 2L_i + 2L_s + 2\lceil \log_{256} |S| \rceil N_s$$

bytes. The receiving machine then uses this to correct the errors in the string it had previously constructed: The substitutions (of total length L_s) are, in the worst case, isolated and thus corrupt one symbol each, while the regions lost due to insertions (of total length L_i) and indels (which damage at most $(N_d + N_i)B$ bytes) are in large blocks and therefore corrupt entire symbols at once[17].

Taking $B = \sqrt{|S| \log 256/N} \log |S|$, we can therefore construct a patch of size

$$\left(2L_s + 2L_i + 2\log_{256} |S|(N_s + 2\sqrt{|S|\,(N_d + N_i) \log 256})\right)(1 + o(1))$$

bytes which can be applied with probability approaching 1 to any string T which differs from S by a sequence of at most N_s substitutions of total length L_s, N_i deletions of total length L_i, and N_d insertions.

3.6 Practical improvements

While the above is quite satisfying theoretically — it reduces the "cost" of a byte-substitution from $O((|S|/N)^{1/2+\epsilon})$ (where N denotes the total number of indels and substitutions) to $O(\log |S|)$ — we can make significant practical improvements.

[17]Note that the cost of reconstructing missing data (e.g., insertions) has doubled compared to rsync: In this case, we have not identified such regions as erasures.

First, rather than taking $L = 4\,|S| \log |S|/B$, we take $L = 16\,|S| \log |S|/B$. This allows us to correctly align not only blocks which are indel-free and contain up to 50% mismatches, but also to align part of most blocks containing a single indel and up to 50% mismatches[18]: Considering the indel as dividing the block into two parts, at least one of the parts will be large enough (i.e., provide enough "signal") to be found with high probability. Conversely, any indel-free region of length at least B is likely to have at least one sub-region correctly aligned — that is, providing that no two indels are located within B bytes of each other, we will have an alignment which is correct apart from the positions of the boundaries between aligned regions, which may be incorrectly placed by up to $B/2$ bytes in either direction.

To improve this alignment, we take each such boundary in turn, and adjust it in order to maximize the number of matching characters[19]. While the receiving machine does not have a copy of S and therefore cannot perform this adjustment perfectly, it has $A^{(1)}$ and $A^{(2)}$, and can act to maximize the dot product of these with the result of mapping ϕ_i onto the potentially matching characters. Since each $A_j^{(i)}$ is the sum of contributions from at most $|S|/L + \mathrm{O}(1) = B/(16 \log \bar{S}) + \mathrm{O}(1)$ characters, this corrects the boundaries from being incorrect by an average of $B/4$ bytes to being incorrect by an average of $B/(8\sqrt{2\pi} \log \bar{S})$ bytes, which is roughly a 50-fold improvement for the sizes of strings which concern us.

We[20] now have an alignment between S and T which covers S and at least approximately reflects the "correct" alignment between the two strings: Inter-indel regions of length less than B may be missing, and the boundaries between such regions (i.e., the positions of indels) will be not be placed in exactly the correct positions, but the alignment is at least fairly close. We now make one final improvement: Noting that in some areas this alignment will be entirely incorrect (either because the correct alignment has too many indels, or due to insertions), we take each block in turn and shorten it in an attempt to replace probable errors with erasures. This is performed

[18] For that matter, an indel-free block containing 75% mismatches is likely to be correctly aligned, but this isn't useful to us, since the cost of correcting those errors would exceed the cost of reconstructing an erased block.

[19] cf. section 2.4.

[20] I excuse the (grammatically questionable) use of the first person plural on the basis that I write on behalf of both myself and the computer which is carrying out the algorithm described.

by considering again the dot product of the vectors $A^{(i)}$ with the result of mapping ϕ_i onto the potentially matching characters, but this time we estimate the contribution which would be obtained from each character if the matching were correct. This cannot be done with any degree of precision, as a result of the weighting applied to each character in constructing the $A^{(i)}$, but it nevertheless is somewhat useful when there are large regions of S which do not correspond to any part of T.

After the adjustment has been performed to the alignment, the errors and erasures are corrected; rather than using a single Reed-Solomon code for this purpose, we refer back to section 3.3 and use a two-phase randomized code, where the number of parity bytes is calculated based on the desired maximum probability of error ϵ and the radius R within which we expect the string T to lie (or, equivalently, the number of errored and erased bytes we expect to find at this point).

Finally, we note that for practical purposes, the optimal value of B given above (even if it could be computed in advance) is likely unsuitable: while the method described here operates in almost-linear time at the sender (and is dominated by the cost of Reed-Solomon encoding), it takes $\mathrm{O}((|S|\,|T|\,/B^2)^{1+\epsilon}) = \mathrm{O}((|T|\,(N_i+N_d))^{1+\epsilon})$ time at the receiving side. As in Chapter 2, we instead take a fixed $B = 2\sqrt{|S|\log|S|}$, unless we know that the two files are extremely closely related[21], in which case a larger value may be preferred.

3.7 Performance

To demonstrate the performance of the methods described here, we refer back to the two corpora used in Chapter 2, the first consisting of 15 "upgrades" and the second of 82 "security patches". For each of these pairs of files, we compute:

1. The total number of bytes transmitted (in either direction) by `rsync -z`, i.e., the normal two-phase rsync algorithm using *deflate* compression on the unmatched blocks.

[21] i.e., unless we have reason to believe that $N_i + N_d < \log 4 \log |S|$.

Program	Original	bzip2	bsdiff 6	rsync -z	one-way rsync	Block matching					
						index	erasures	errors	R_1	R_2	Total
agrep: 4.0 → 4.1	262144	114388	4265	45721	82752	11848	2823	5020	23	1615	39149
glimpse: 4.0 → 4.1	524288	222548	24642	220324	447340	18299	20769	38380	69	4818	171675
glimpseindex: 4.0 → 4.1	442368	193883	16240	193090	385975	16458	19760	28319	65	3753	137678
wgconvert: 4.0 → 4.1	368640	157536	12432	153151	313492	13725	9775	19873	53	2992	97095
agrep: 3.6 → 4.0	262144	114502	44327	120856	244152	11953	145889	22201	214	3149	239245
glimpse: 3.6 → 4.0	524288	222178	109680	236643	492940	16011	371685	38708	244	4036	527505
glimpseindex: 3.6 → 4.0	442368	193892	80447	205472	412375	15452	268397	47393	238	4533	438582
netscape: 3.01 → 3.04	6250496	2396661	212032	1996455	4349706	86789	817578	439258	93	37019	2368208
gimp: 0.99.19 → 1.00.00	1646592	642725	219684	634918	1390304	31875	964476	122590	214	11781	1443340
iconx: 9.1 → 9.3	548864	233056	31632	240298	503250	18087	106942	36012	108	5358	253337
gcc: 2.8.0 → 2.8.1	2899968	708301	88022	826093	2644877	42320	392902	170194	87	18653	1080736
rcc (lcc): 4.0 → 4.1	811008	221826	187	18482	24058	20850	0	80	0	231	21399
apache: 1.3.0 → 1.3.1	679936	180708	25927	180729	581319	19677	159302	53692	127	6385	373551
apache: 1.2.4 → 1.3.0	671744	179369	163249	212517	662780	19636	555078	93948	255	0	691561
rcc (lcc): 3.2 → 3.6	434176	155090	22691	152800	362140	17628	103265	31103	124	4319	238518
Average Compression	100%	36.22%	7.67%	33.54%	78.03%						52.51%

Table 3.1: Bandwidth used by bzip2, (two-way) rsync, one-way rsync, and our block matching universal delta compressor for "upgrade" patches.

2. The minimum size of patch needed by the "one-way rsync" algorithm described in Section 3.4 in order to correctly apply the update with probability of failure $\epsilon < 0.001$[22].

3. The size of index, the number of errors and erasures, the number of parity bytes needed per 255-byte subsequence (R_1) in the error-reduction phase, and the number of parity characters needed in the error-correction phase (R_2), and finally the minimum necessary total patch size needed by our block matching delta compressor with the improvements listed in Section 3.6 (again, subject to an acceptable probability of failure of 0.001).

In the case of rsync and one-way rsync, the block size was fixed at 300 bytes; for our block matching delta compressor, where a constant block size would lead to quadratic running-time, the block size was $2\sqrt{|S| \log |S|}$.

Note that for the two universal delta compressors, the patch sizes computed are minimum sizes — patches which included more parity data would apply with a lower probability of error. The exact sizes of patches used will necessarily depend upon the method in which this is being used: If one has all of the potential files T readily available, and the purpose of using universal delta compression is simply to allow for a single file to be broadcast instead of a large number of different patches, then the computed minimum patch sizes would likely be used; if, on the other hand, one has no knowledge of the files T, one would probably send a sequence of patches computed using increasing amounts of parity data. We expect that, in practice, most uses would fall between these two extremes — that the files T would be unknown to the sending machine, but that it could generate a number of "typical" files T, and use these to estimate the amount of parity data needed.

As in Chapter 2, we compute the average compression as the the average value of [patch size]/[original size].

In Table 3.1, we show the results for the 15-element "upgrade" corpus. Two points are immediately evident: First, as we expect from the structure of executable files,

[22] A probability of failure is inevitable in any practical universal delta compression algorithm; but by sending a strong hash (e.g., SHA1 [38]) along with the patch, failures can normally be recognized and resolved by other mechanisms.

even with a block size of 300 bytes there are very few matching blocks for rsync to utilize. As shown by the size of "one-way rsync" patches, only about 25% of the blocks can be matched; the remaining 75% are encoded literally. These few matching blocks allow `rsync -z` to narrowly beat bzip2 overall, but the vast majority of the compression it achieves arises from the deflation of the literal data. Second, while our block matching universal delta compressor performs quite poorly overall (and in two cases fails to achieve any compression at all), it performs relatively well in cases where the differences are modest: On the first four pairs of files, it generates smaller patches than rsync, and in a few other cases it is fairly close to rsync in performance. Here again we note that rsync has the advantage of being able to deflate literal data — were it not for that ability, our block matching delta compressor would produce smaller patches in almost all cases. Consequently, the best approach for "universal" delta compression in the case of such significant differences is a depressing solution: Apply normal bzip2 compression, and ignore the old files entirely. Finally, as expected, neither rsync nor either of the universal delta compressors achieve performance even remotely approaching that of bsdiff 6.

In Table 3.2, we show the results for the 82-element "security patch" corpus, and the situation is quite changed. Both rsync and one-way rsync perform significantly better, matching roughly 50% of blocks. While one-way rsync still performs on average worse than bzip2, it produces smaller patches on three file types — library archives, C header files, and (textual) configuration files — which all share the property of not being affected by the "cascade of differences" introduced by linking together executable code[23]. Of greater note is the performance of our block matching universal delta compressor: On all the file types, it performs roughly 3 times better than bzip2, and with the exception of the text files (where rsync has an advantage as a result of using a much smaller index) it uses considerably less bandwidth than rsync, despite lacking the advantages of feedback and deflation of literal data. Again, however, none of the universal delta compressors approach the performance of bsdiff

[23]Address changes are still spread throughout any individual modified object file, but not into other object files contained in the same library archive.

File type	# files	Total size	bzip2	bsdiff 6	rsync -z	one-way rsync	block matching
Executables (static)	53	15056508	6725178	250509	5356778	10106284	2273899
Executables (dynamic)	16	8002788	3399743	128859	2549330	4999430	1100731
Library archives	7	7600906	1635094	16476	502267	1288456	489293
Shared Object	3	1438072	599937	13338	396485	752056	180050
C headers	1	11402	3810	126	706	1042	1581
Configuration	2	10183	4407	214	989	1619	1421
Total	82	32119859	12368169	409522	8806555	17348887	4046975
Average Compression		100%	38.51%	1.27%	27.42%	54.01%	12.60%

Table 3.2: Bandwidth used by bzip2, (two-way) rsync, one-way rsync, and our block matching universal delta compressor for "security" patches.

6; in fact (referring back to Table 2.2), even such a poor delta compressor as Xdelta produces smaller patches.

While the compression performance makes this approach at very least interesting, it is not yet clear whether the computational performance will be adequate for widespread usage. Although the computation necessary for aligning the two files is not excessive, it is vastly more expensive than that involved in rsync or the application of normal deltas; furthermore, while the error-reducing step (requiring only Reed-Solomon decoding of 255-byte codewords) is quite fast[24], the final error-correcting stage is likely to be too slow unless it is replaced by a different error-correcting code.

In the end, the practicality of this algorithm will most likely be determined more by future advances in processors and networks than by its own merits. If the present trend where networks are increasing in speed faster than processors continues, it is possible that the entire field of delta compression will become obsolete, as it becomes increasingly simple to retransmit entire files; if, on the other hand, computing capacity starts to increase at a greater rate[25], then this algorithm, regardless of its current computational requirements, may become entirely practical.

[24]Commercially available libraries cite performance of several megabytes per second on commodity processors.

[25]While recent developments seem to indicate a slowing down of advances in processor speed, the trend towards increased internal parallelism may, at least for some problems, reverse this trend.

Appendix A: Source code and data

The "upgrade" and "security" corpora used in Chapter 2 and Chapter 3 were obtained from the paper by Baker, Manber and Muth [6], via Robert Muth, and from FreeBSD Update [44] respectively, and may be obtained from the author[1].

An earlier version of the software described in Chapter 2, namely bsdiff 4.2, has been published under an open source license and is available (at least at the time of publication) from the following web site:

`http://www.daemonology.net/bsdiff/`

Under the statutes of Oxford University [42], the remaining software may be claimed by the University if it 'may reasonably be considered to possess commercial potential'. At present, it is not yet clear if the University will so elect. If the University does not claim ownership of the remaining software, it will also be released under an open source license after it has been "cleaned up" somewhat and prepared for widespread usage.

[1]We note that the binaries were originally distributed under a variety of licenses; but we believe that their inclusion in delta compression corpora, and their use for the purpose of evaluating the effectiveness of delta compression algorithms, falls well within the legal definitions of "fair dealing" or "fair use", thus exempting them from normal copyright restrictions.

Epilogue: A Rapid String Similarity Metric

In Chapter 1 we introduced the approach of projecting vectors onto subspaces of relatively prime dimensions and used it to construct an algorithm for matching with mismatches. Lest the reader be left with the impression that this approach is useful solely for this single problem, we present one final "late-breaking" algorithm.

Recall from Chapter 1 that the match count vector $V \in \mathbb{R}^n$ is defined by

$$V_i = \sum_{j=0}^{m-1} \delta(S_{i+j}, T_j)$$

for some appropriate function $\delta(x, y)$. Consider how this vector behaves when the two strings S, T differ by a small number of indels (and up to a constant proportion of substitutions). The vector V will have a few very large values, at positions corresponding to the offsets of the indel-free blocks which match between the two strings, and will otherwise have values which cluster around μm, where μ is the mean value of δ when applied to random inputs.

Now consider the variance $\sigma^2 V = \sum V_i^2/(m-1) - (\sum V_i)^2/(m(m-1))$. For pairs of strings which are similar (or rather, which share large indel-free regions), the positions in V which have unusually large values will translate directly into a large variance, whereas for strings which are dissimilar, the variance will be comparatively small.

Now project V onto a subspace $\mathbb{R}^{Z_p}$. The "noise threshold" is increased, but again for sufficiently large p, similar strings will result in a larger variance $\sigma^2(\mathbb{R}^n \to \mathbb{R}^{Z_p})V$ than dissimilar strings.

But wait! The vector V can be estimated as the cyclic correlation of two vectors $A = \phi(S)$ and $B = \phi(T)$. The cyclic correlation is computed as the inverse Fourier

transform of the pointwise product of the Fourier transformed inputs[1]. And the variance of an inverse Fourier transform is the sum of the squared norms of the non-DC components[2].

We thus have the following:

Algorithm. *Let some* $\phi\colon \Sigma \to \{-1, 1\}$, $p \in \mathbb{N}$ *be fixed. Then given a string* S*, we compute:*

1. *For* $j = 0 \ldots p-1$,
$$A_j = \sum_k \phi(S_{j+kp})$$
2. *For* $j = 0 \ldots p-1$,
$$\bar{A}_j = \sum_k A_j \exp(2ijk\pi/p)$$
3. *For* $j = 1 \ldots (p-1)/2$,
$$\bar{S}_j = \left|\bar{A}_j\right|^2 / \sqrt{\sum_{j=1}^{(p-1)/2} \left|\bar{A}_j\right|^4}$$

Then the dot product $\bar{S} \cdot \bar{T}$ *of the "digests" of two random strings* S, T *will be approximately equal to* $1/2$*, the dot product* $\bar{S} \cdot \bar{S}$ *of the digest of a string with itself is equal to* 1*, and other pairs of strings will lie in between, in accordance with their similarity.*

Proof. It suffices to take the first half of the Fourier transform (i.e., the terms $\bar{A}_1 \ldots \bar{A}_{(p-1)/2}$, since for real inputs (such as we have) the remaining terms are simply the conjugates with the same (Euclidean) norms.

That random strings result in a dot product of $1/2$ follows from consideration of the $\bar{A}$ as being random vectors of fixed norm; the remaining result follows from the argument above concerning the influence of large indel-free regions on the match count vector V. □

[1]Subject to the second string being reversed, naturally.
[2]Recall that the Fourier transform is a rotation in $\mathbb{R}^n$, i.e., it preserves 2-norm length.

Bibliography

[1] K.A.S. Abdel-Ghaffar and A. El Abbadi. An optimal strategy for comparing file copies. *IEEE Transactions on Parallel and Distributed Systems*, 5:87–93, 1994.

[2] M. Adler and J. Gailly. zlib compression library, 2003. http://www.zlib.org.

[3] M.J. Atallah, F. Chyzak, and P. Dumas. A randomized algorithm for approximate string matching. *Algorithmica*, 29(3):468–486, 2001.

[4] K. Baba, A. Shinohara, M. Takeda, S. Inenaga, and S. Arikawa. A note on randomized algorithm for string matching with mismatches. In *Proceedings of the Prague Stringology Conference '02 (PSC'02)*, pages 9–17, 2002.

[5] R.A. Baeza-Yates and C.H. Perleberg. Fast and practical approximate string matching. In *Proceedings of the 3rd Annual Symposium on Combinatorial Pattern Matching (CPM '92)*, LNCS 644, pages 185–192. Springer-Verlag, 1992.

[6] W.S. Baker, U. Manber, and R. Muth. Compressing differences of executable code. In *ACM SIGPLAN Workshop on Compiler Support for System Software (WCSS)*, pages 1–10, 1999.

[7] L.I. Bluestein. A linear filtering approach to the computation of the discrete Fourier transform. *IEEE Transactions on Audio and Electroacoustics*, AU-18(4):451–455, 1970.

[8] G. Brederlow. reverse checksumming. rsync@lists.samba.org mailing list, April 2001.

[9] R.P. Brent, J. van de Lune, H.J.J. te Riele, and D.T. Winter. On the zeros of the Riemann zeta function in the critical strip. II. *Mathematics of Computation*, 39(160):681–688, 1982.

[10] M. Burrows and D.J. Wheeler. A block-sorting lossless data compression algorithm. SRC Research Report 124, Digital Equipment Corporation, Palo Alto, California, 1994.

[11] W. Chang and T. Marr. Approximate string matching and local similarity. In *Proceedings of the 5th Annual Symposium on Combinatorial Pattern Matching (CPM '94)*, LNCS 807, pages 259–273. Springer-Verlag, 1994.

[12] H. Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *Annals of Mathematical Statistics*, 23:493–507, 1952.

[13] J.W. Cooley and O.W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19:297–301, 1965.

[14] H. Dobbertin. Alf swindles Ann. *RSA Laboratories Crypto Bytes*, 1(3), 1995.

[15] P. Dusart. The k^{th} prime is greater than $k(\ln k + \ln \ln k - 1)$ for $k \geq 2$. *Mathematics of Computation*, 68(225):411–415, 1999.

[16] R. Fielding, J. Gettys, J. Mobul, H. Nielsen, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol — HTTP/1.1. RFC 2616, 1999.

[17] M.J. Fischer and M.S. Paterson. String matching and other products. In R. Karp, editor, *Complexity of Computation — SIAM-AMS Proceedings*, volume 7, pages 113–125, 1974.

[18] FreeBSD Project. The FreeBSD operating system. http://www.freebsd.org/.

[19] FreeBSD Project. FreeBSD CVS repository, 2004. http://www.freebsd.org/cgi/cvsweb.cgi/.

[20] FreeBSD Project. FreeBSD security information, 2004. http://www.freebsd.org/security/.

[21] P. Graham. A plan for spam, 2002. http://www.paulgraham.com/spam.html.

[22] C.J. Heath and P. Hughes. Data file synchronization. Australian Patent 763524, 1999.

[23] C.J. Heath and P. Hughes. Data file synchronization. United States Patent 6,636,872, 2000.

[24] W. Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13–30, 1963.

[25] J.T. Hunt, K.-P. Vo, and W.F. Tichy. Delta algorithms: an empirical analysis. *ACM Transactions on Software Engineering and Methodology*, 7(2):192–214, 1998.

[26] J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *Proceedings of the 13th International Conference on Automata, Languages, and Programming*, LNCS 2719, pages 943–955. Springer-Verlag, 2003.

[27] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching (CPM '01)*, LNCS 2089, pages 181–192. Springer-Verlag, 2001.

[28] V.J. Katz. *A history of mathematics*. Harper Collins, 1993.

[29] D.K. Kim, J.S. Sim, H. Park, and K. Park. Linear-time construction of suffix arrays. In *Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching*, LNCS 2676, pages 186–199. Springer-Verlag, 2003.

[30] D.E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison Wesley, third edition, 1997.

[31] D.E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison Wesley, third edition, 1997.

[32] P. Ko and S. Alaru. Space efficient linear time construction of suffix arrays. In *Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching*, LNCS 2676, pages 200–210. Springer-Verlag, 2003.

[33] D.G. Korn and K.-P. Vo. Engineering a differencing and compression data format. In *Proceedings of the 2002 USENIX Annual Technical Conference*, pages 219–228, 2002.

[34] N.S. Larsson and K. Sadakane. Faster suffix sorting. Technical Report LU-CS-TR:99-214, Department of Computer Science, Lund University, 1999.

[35] J.P. MacDonald. File system support for delta compression. Master's thesis, University of California at Berkeley, 2000.

[36] P. Mackerras and A. Tridgell. The rsync algorithm. Technical Report TR-CS-96-05, Department of Computer Science, Australian National University, 1996.

[37] Microsoft Corporation. *Microsoft Portable Executable and Common Object File Format Specification, Revision 6.0*, 1999.

[38] National Institute of Standards and Technology. The secure hash algorithm (SHA-1). NIST FIPS PUB 180-1, U.S. Department of Commerce, 1995.

[39] G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.

[40] S.B. Needleman and C.D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48:443–453, 1970.

[41] Alon Orlitsky. Interactive communication: Balanced distributions, correlated files, and average-case complexity. In *IEEE Symposium on Foundations of Computer Science*, pages 228–238, 1991.

[42] Oxford University. *University statutes and congregation regulations, Statute XVI, Part B: Intellectual property*, 2003. http://www.admin.ox.ac.uk/statutes/.

[43] S. Peleg. Difference extraction between two versions of data-tables containing intra-references. United States Patent 6,546,552, 2003.

[44] C. Percival. An automated binary security update system for FreeBSD. In *Proceedings of BSDCon '03*, pages 29–34, 2003.

[45] Pocket Soft Inc. .RTPatch, 2001. http://www.pocketsoft.com.

[46] C.F. Pyne. Remote file transfer method and apparatus. United States Patent 5,446,888, 1995.

[47] C.F. Pyne. Remote file transfer method and apparatus. United States Patent 5,721,907, 1998.

[48] I.S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society of Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[49] R. Rivest. The MD4 message-digest algorithm. RFC 1320, 1992.

[50] R. Rivest. The MD5 message-digest algorithm. RFC 1321, 1992.

[51] J. Seward. bzip2, 2002. http://sources.redhat.com/bzip2/.

[52] M.V. Sliger, T.D. McGuire, and J.A. Forbes. File update performing comparison and compression as a single process. United States Patent 6,496,974[1], 2002.

[53] M.V. Sliger, T.D. McGuire, and R.M. Shupak. Preprocessing a reference data stream for patch generation and compression. United States Patent 6,466,999, 2002.

[54] Sun Tsu Suan-Ching, 4th century AD[2]

[55] W.F. Tichy. RCS – a system for version control. *Software – Practice & Experience*, 15(7):637–654, 1985.

[1] We are surprised that this patent was granted, given that it does not appear to cover any methods not previously published in [25].

[2] A more accessible description of this algorithm is presented in [30]. For more historical background, the reader is encouraged to consult [28].

[56] D. Trendafilov, N. Memon, and T. Suel. zdelta: An efficient delta compression tool. Technical Report TR-CIS-2002-02, Polytechnic University, CIS Department, 2002.

[57] A. Tridgell. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, The Australian National University, 1999.

[58] X. Wang, D. Feng, X. Lai, and H. Yu. Collisions for hash functions MD4, MD5, HAVAL-128 and RIPEMD. Cryptology ePrint Archive, Report 2004/199, 2004.