Sequence Distance Embeddings

by

Graham Cormode

Thesis

Submitted to the University of Warwick

for the degree of

Doctor of Philosophy

Computer Science

January 2003

Contents

List of Figures

Acknowledgments

Any thesis is more than just the work of one person, and I owe a large debt to many people who have helped me along the way. In particular, I must acknowledge the help of three people who have all made huge contributions to my work.

Mike Paterson has been a great supervisor throughout my years at Warwick. His willingness to help with whatever I came to him with and his meticulous attention to detail have been of immense help. Cenk Ṣahinalp introduced me to some marvellous problems, and worked on them together with me with insight and guidance. He also arranged for me to spend a year of my degree with him at Case Western Reserve University. S. Muthukrishnan (Muthu) has always responded with enthusiasm to my work, and together we have had a lot of fun writing papers. He was a superb mentor when I visited AT&T as an intern in Summer 2000, and I'm very grateful to him for arranging frequent trips back to visit since then.

After these three, I must also thank all the other people who have written papers with me since I began my studies: Uzi Vishkin, Nick Koudas, Piotr Indyk, Tekin Ozsoyoglu, Meral Ozsoyoglu, Mayar Datar and Hurkan Balkir. Throughout my studies, I have been very fortunate with my officemates, first amongst these being Jon Sharp, who has been both an officemate and a flatmate. There has also been Mary Cryan, Hesham Al-ammal, Sun Chung, Steven Kelk, and Harish Bhanderi. I also need to thank everyone with whom I've talked about my work: Amit Chakrabarti, Funda Ergun, Martin Strauss, and many others. My studies have been supported financially by a University of Warwick Graduate Award; by a Case Western Reserve University Fellowship; by an internship at AT&T research; and by travel grants from the Center for Discrete Mathematics and Theoretical Computer Science (DIMACS); Algorithms and Complexity — Future Technologies (ALCOM-FT); the University of Haifa, Israel; University of Warwick American Study and Student Exchanges Committee (ASSEC); and the Department of Computer Science, University of Warwick.

My family, and especially my parents deserve recognition for putting up with my travails, and for never knowing entirely what it is that I do[1]. My friends and flatmates, too many to mention, have also been supportive or diverting, and so made my studies easier to manage. Mark Hadley deserves much credit for making available his wthesis LaTeXstyle file. I would also like to acknowledge the Students' Union of the University of Warwick, and in particular the Elections Group, the Postgraduate Committee and the crossword of the Warwick Boar newspaper, all of which have contributed to dragging this thesis out longer than it otherwise needed to take.

[1]Except for my grandmother, who was convinced that I was working on *Knitting Sequins In Bed*

Declarations

Unlike Wilde, I have a great deal to declare. Much of the material in this thesis derives from papers that have been published in refereed conferences. These are all collaborative works, and as is the nature of such collaborations, it is often difficult to disentangle or lay claim to the individual efforts of any particular author.

Chapter 1 consists mostly of introductory and background material. Section 1.5.2 derives from an idea of S. Muthukrishnan, described in [CM02]; the remainder is new for this thesis. Chapter 2 consists mostly of a discussion of relevant prior work of other authors (attributed when referenced), which is used by later sections. In particular, the methods described in Section 2.2.4 are due to Indyk. They were outlined in [Ind00], and then expanded upon in [CIKM02] and [CDIM02]. Some of the subsections of Section 2.3.1 is my work from [CDIM02]. The remainder, constituting the bulk of the chapter, is an original presentation of related work.

In Chapter 3, Section 3.1, Section 3.2.2, Section 3.2.3, Section 3.2.4, and Section 3.3 are extended versions of results first shown in [CMṢ01]. All the embeddings and related proofs presented there are my own work, as are the hardness results and applications of the embeddings. S. Muthukrishnan suggested the application to Approximate Pattern Matching, which I described and analysed. The second example in Section 3.2.1 has been published in another form as 1 Across in [Cor02]. Sections 3.2.1, 3.2.5, 3.2.6 and 3.3.3 is my work that has not previously been published.

The embedding presented in Chapter 4 derives from that given by Cenk Sahinalp in [MṢ00], as is discussed in the text. The construction given here is a significantly simplified reformulation, about which it is possible to prove stronger results. The proofs are due to me, as is the application to edit distance with moves. The application to pattern matching and streaming computation were suggested by S. Muthukrishnan, and were again described and analysed by myself. The material in Sections 4.4.2, 4.4.3, 4.4.4, and 4.6 is my own work which has not been published elsewhere. Chapter 5 reports experiments that I designed, coded, ran and analysed myself that were first described in [CIKM02] and [CDIM02]. In Chapter 6, Section 6.2, Section 6.3, Section 6.4.1, Section 6.4.2 and Section 6.4.4 are derived from [CPṢV00]. The entirety of the results in this chapter were discovered and described by me, and the results in Section 6.5 have not previously been published. None of this work is drawn from [OBCO00], but I wanted to mention it anyway.

The following declarations are required by the University of Warwick Guide to Examinations for Higher Degrees by research. I'd skip over them if I were you. This thesis is entirely my own work apart from the exceptions listed above which are the result of published collaborative research in which I took the lead role in the research, experiments, and preparation for publication. This thesis has not been previously submitted in any form for a degree at Warwick or any other university.

Sequence Distance Embeddings

The Maze of Ways from A to B

Writing a book is rather like going on a long journey. You know where you are (at the beginning) and you know where you want to get to (the end). The big problem is, what route should you take? I'm sure you know that the shortest distance between two points is a straight line, but if you decide now to go somewhere in a straight line, after going just a few paces you will probably come to a sudden stop and say, 'Ouch!' If the pain is in your leg, you will have walked into the furniture, but if the pain is in your nose, you will have walked into the wall. And it's no good me telling you to stop being stupid and sit down — with your nose right up against the wall, you won't be able to read this book.

[Bal87]

Abstract

Sequences represent a large class of fundamental objects in Computer Science — sets, strings, vectors and permutations are considered to be sequences. Distances between sequences measure their similarity, and computations based on distances are ubiquitous: either to compute the distance, or to use distance computation as part of a more complex problem. This thesis takes a very specific approach to solving questions of sequence distance: sequences are embedded into other distance measures, so that distance in the new space approximates the original distance. This allows the solution of a variety of problems including:

- Fast computation of short 'sketches' in a variety of computing models, which allow sequences to be compared in constant time and space irrespective of the size of the original sequences.
- Approximate nearest neighbor and clustering problems, significantly faster than the naïve exact solutions.
- Algorithms to find approximate occurrences of pattern sequences in long text sequences in near linear time.
- Efficient communication schemes to approximate the distance between, and exchange, sequences in close to the optimal amount of communication.

Solutions are given for these problems for a variety of distances, including fundamental distances on sets and vectors; distances inspired by biological problems for permutations; and certain text editing distances for strings. Many of these embeddings are computable in a streaming model where the data is too large to store in memory, and instead has to be processed as and when it arrives, piece by piece. The embeddings are also shown to be practical, with a series of large scale experiments which demonstrate that given only a small space, approximate solutions to several similarity and clustering problems can be found that are as good as or better than those found with prior methods.

Abbreviations

Sets

A, B	Sets A and B
$A \cup B$	Set Union
$A \cap B$	Set Intersection
$A \Delta B$	Symmetric set difference
$\|A\|$	Number of distinct elements in set A
$\chi(A)$	The characteristic function of a set, mapping a subset of an (ordered) set onto a bit-string — Section 2.3.1

Vectors and Matrices

$\boldsymbol{a}, \boldsymbol{b}$	Vectors
$\|\boldsymbol{a}\|$	Length (dimensionality) of the vector $\boldsymbol{a}$
$\|\|\boldsymbol{a}\|\|_p$	L_p norm of the vector $\boldsymbol{a}$ — Definition 1.2.5
$\|\|\boldsymbol{a} - \boldsymbol{b}\|\|_p$	L_p distance between vectors $\boldsymbol{a}$ and $\boldsymbol{b}$
$\|\|\boldsymbol{a}\|\|_H$	The Hamming norm of $\boldsymbol{a}$, the number of locations in which it is non-zero.
$\boldsymbol{A}, \boldsymbol{B}$	Matrices
$\boldsymbol{A}_i$	The ith row of matrix $\boldsymbol{A}$, a vector.
$\mathrm{median}(\boldsymbol{a})$	The median of the multi-set of values in the vector $\boldsymbol{a}$
$sk(\boldsymbol{a})$	The sketch of a vector $\boldsymbol{a}$ — Definition 2.1.3
$i(a, b)$	The size of the intersection of two bit-strings a and b — Definition 1.2.4

Permutations

P, Q	Permutations of the integers $1 \ldots \|P\| = n$
P^{-1}	Inverse of permutation P — Section 1.3
$\phi(P, Q)$	Number of reversal breakpoints between P and Q — Definition 3.2.5
$tb(P, Q)$	Number of transposition breakpoints between P and Q — Definition 3.2.8
$LIS(P)$	Longest (strictly) Increasing Subsequence of permutation P — Section 3.2.4
$LCS(P, Q)$	Longest Common Subsequence of sequences P and Q — Section 3.2.4

Permutation Distances

$r(P, Q)$	Reversal distance between P and Q — Definition 1.3.1
$t(P, Q)$	Transposition distance between P and Q — Definition 1.3.2
$\mathrm{swap}(P, Q)$	Swap distance between P and Q — Definition 1.3.3
$d(P, Q)$	Permutation edit distance between P and Q — Definition 1.3.4
$\tau(P, Q)$	Reversal, Transposition and Edit distance between P and Q — Section 3.2.6
$\tau'(P, Q)$	Reversal, Indel, Transposition and Edit distance (RITE) — Section 3.2.6
$\tau''(P, Q)$	RITE distance between a permutation and a string — Section 3.2.6

Strings

a, b	Strings a and b drawn from a finite alphabet σ
$a[i]$	The ith character of the string a (a member of σ)
$a[l : r]$	The substring formed by the concatenation of $a[l], a[l + 1], \dots a[r]$
k	Shorthand for $\lvert\sigma\rvert - 1$ in Section 6.2
$h(a, b)$	The Hamming distance between strings a and b — Definition 1.4.2
$e(a, b)$	String edit distance between a and b — Definition 1.4.3
$\mathrm{tichy}(a, b)$	Tichy's distance between a and b — Definition 1.4.4
$lz(a, b)$	LZ-distance between a and b — Definition 1.4.5
$c(a, b)$	Compression distance between a and b — Definition 1.4.6
$d(a, b)$	String edit distance with moves — Definition 1.4.8
T, P	Text T and pattern P in approximate pattern matching — Section 1.5.2
$D[i]$	The least distance between P and prefix of $T[i : n]$ — Definition 1.5.1

Edit Sensitive Parsing

ESP	Edit Sensitive Parsing — Section 4.2
label($a[i]$)	Function applied in the alphabet reduction step of ESP — Section 4.2.2
$ET(a)$	The parse tree produced by ESP — Section 4.2.3
$T(a)$	The set of substrings induced by $ET(a)$ — Definition 4.3.1
$V(a)$	The characteristic vector of $T(a)$ — Definition 4.3.1
$ET_i(a)_j$	The jth node at level i in $ET(a)$ — Definition 4.5.1
$range(ET_i(a)_j)$	The range $[l \dots r]$ such that $ET_i(a)_j$ corresponds to $a[l : r]$ — Definition 4.5.1
$EST(a, l, r)$	The subtree of $ET(a)$ that contains nodes that intersect with $S[l : r]$ — Definition 4.5.1
$VS(a, l, r)$	The characteristic vector of $EST(a, l, r)$ — Definition 4.5.2

Other notation

xor	The exclusive-or function, $\{((0, 0), 0), ((0, 1), 1), ((1, 0), 1), ((1, 1), 0)\}$
$\bar{\boldsymbol{a}}$	The complement of the bit-string $\boldsymbol{a}$: $\bar{\boldsymbol{a}}[i] = \mathrm{xor}(\boldsymbol{a}[i], 1)$
$\chi(H)$	The chromatic number of the (hyper)graph H
$hash(a)$	A hash function mapping onto a smaller range of integers — Section 2.1.3
$\hat{x}$	An approximation of any quantity x
(ϵ, δ)-approximation	An approximation within a factor of $(1 \pm \epsilon)$ with probability δ — Section 2.1
c-approximation	An approximation that is within a factor of c — Section 2.1

Chapter 1

Starting

Miss Hepburn runs the gamut of emotions from A to B

— Dorothy Parker, 1933

1.1 Sequence Distances

Sequences are fundamental mathematical structures to be found in any Discrete Mathematics textbook. Such familiar items as Sets, Strings, Permutations and Vectors are counted here as sequences. A very natural question to ask given two such objects (of the same kind) is to ask "How similar are these objects?" Mathematically, we should like to define the similarity between them in a quantitative manner. This question is at the core of this work; throughout, we shall be concerned with posing and answering questions which ultimately rely on comparing pairs of sequences.

We therefore begin by exploring the kinds of sequence of interest, and by describing some of the distances between them. It is not the case that there is always a single natural distance between structures. Depending on the situation, there might be many sensible ways of choosing a distance measure. By way of an example, consider measuring the distance between two houses in different towns: how far apart are they? The scientist's answer is to measure the 'straight line' distance between them. A cartographer might prefer to measure the shortest road distance between them, which is unlikely to be a straight line. A traveller would perhaps measure the distance in hours, as the time it would take to go from one to another (taking into account traffic conditions and speed limits). A child might just say that the houses were either 'near' or 'far'. Each measurement of distance is appropriate to a particular circumstance, and all are worth studying.

Sequence distances are vital to many computer science problems. The simplest question is to measure the distance between two objects, or find an approximation to this quantity. Other problems fall into various categories. There are *geometric* problems — organising objects into groups, or finding similar objects, based on the distance between them. *Pattern matching* problems revolve around finding occurrences or near occurrences of a short pattern sequence within a longer sequence. This requires finding a subsequence that has a small distance from the pattern. *Communication* problems require two or more parties to communicate with each other in order to solve some problem based on the distance.

Throughout this work we will take a uniform approach to solving problems of sequence distances. We use the idea of embeddings. These are defined formally later, but the basic idea is simple: transform the sequences of interest into different sequences so that an appropriate distance between the transformations approximates the distance between the original sequences. This can sometimes seem a counter-intuitive step: how does switching sequences around do anything but add work? There are two basic reasons why this is a useful approach. First, the embedding can reduce the dimensionality of the space. In other words, the resulting sequence can be much shorter than the original sequence. This means that if the sequence needs to be compared with many others, or sent over a communication link, then the cost of this is greatly reduced. Secondly, the embedding can transform from a distance space about which little is known into one that has been well-studied. So if we want to solve a certain problem for a novel distance measure, one approach is to transform the sequences into sequences in a well-understood metric space, and then solve the problem in this space. With certain caveats, a solution to the problem in the target space translates to a solution in the original space.

This gives a general outline of this work. We will first consider a variety of sequences and distances on them. We will also describe a set of problems that can be parameterised by a distance measure. Then, for each class of objects (sets, strings, vectors and permutations) we will give embeddings for distances between these objects. Next, we show how these embeddings can be used to solve problems of interest. The first few chapters address distances and their embeddings and immediate applications. Later chapters discuss particular applications in greater detail, and give detailed experimental studies of these methods in practice. The rest of this chapter lays the groundwork for this by giving descriptions of the different object types and distance measures between them, as well as details of the kind of problems that we are interested in solving.

1.1.1 Metrics

In discussing distances, we shall rely on the standard notion of a metric to define the distance from point A to point B. Formally, let a and b be objects drawn from some universe of objects U. Let d be a function $U \times U \to \mathcal{R}^+$. Then d is a *metric* if it has the following three properties:

- Equality: $\forall a, b \in U : d(a,b) = 0 \iff a = b$
- Symmetry: $\forall a, b \in U : d(a,b) = d(b,a)$
- Triangle Inequality: $\forall a, b, c \in U : d(a,c) \leq d(a,b) + d(b,c)$

Because our objects are all discrete, rather than continuous, it will nearly always turn out that the distance between any pair is a natural number. In fact, many of the distances we shall consider will be a particular kind of metric we call 'editing distances'.

1.1.2 Editing Distances

Definition 1.1.1 *A* unit cost editing distance *is a distance defined by a set of editing operations. Formally, the editing operations are defined by a symmetric relation R on the universe U. The editing distance between two objects is the minimum number of editing operations needed to transform one into the other. That is, $d(a,b)$ is the minimum n such that $R^n(a,b)$, and 0 if $a = b$.*

Lemma 1.1.1 *Any editing distance is a metric.*

Proof.

- Equality: follows by definition, the distance is zero if and only if $a = b$.
- Symmetry: If $d(a,b) = n$, it follows that there is a sequence of $n+1$ intermediate objects, $a_0, a_1, \ldots, a_n$ where $a_0 = a$, $a_n = b$ and $(a_i, a_{i+1}) \in R$. Because R is a symmetric relation, it follows that $(a_{i+1}, a_i) \in R$ also, and so there is a sequence $a_n, a_{n-1}, \ldots a_0$. Hence if $d(a,b) = n$ then $d(b,a) \leq n$ also, and so $d(a,b) = d(b,a)$ for all a, b.
- Triangle inequality: Suppose $d(a,b) + d(b,c) < d(a,c)$. Then there is a sequence of editing operations (a_i, a_{i+1}) that goes from a to c via b in $d(a,b) + d(b,c)$ steps: perform the editing operations of $d(a,b)$ followed by the operations of $d(b,c)$. This contradicts the initial assumption, hence the triangle inequality holds.

$\square$

This means that many of the distances we consider will be metrics, since most will be constructed as editing distances.

1.1.3 Embeddings

A majority of our results will be *embeddings* of one distance into another space. Informally, the idea of an embedding is to perform a transformation on objects of one type producing a new object, such that the distance between the transformed objects approximates the distance between the original objects. The motivation for these embeddings is that we may not know much about the original space, but we do know a lot about the target space, and so we are able to manipulate the objects more easily following this transformation. To continue the example of distance between places, a map is an embedding of the surface of the earth onto a small piece of paper — see Figure 1.1. It approximates the features of the earth's surface and although much fine detail is lost, the distance between any two points can be approximated by scaling the distance between them on the map.

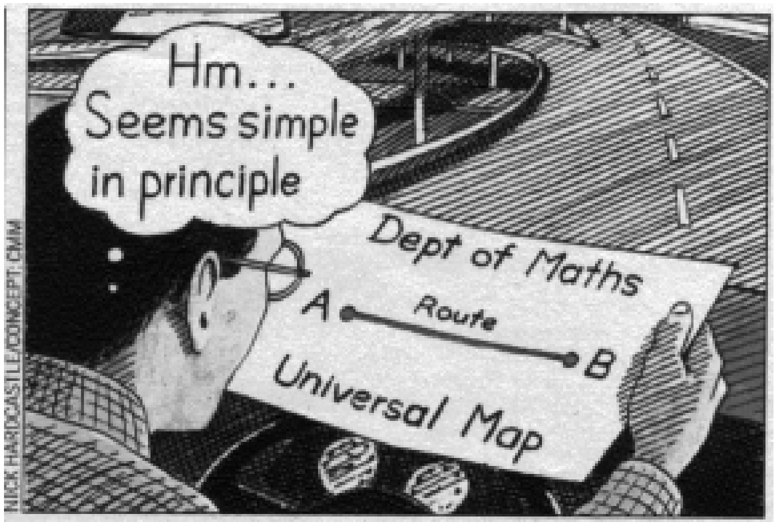


Figure 1.1: A map is an embedding of geographic data into the 2D plane with some loss of information

Definition 1.1.2 *An* embedding *from a space $\mathcal{X}$ to a space $\mathcal{Y}$ is defined by functions $f_1, f_2 : \mathcal{X} \to \mathcal{Y}$ such that for distance functions $d_{\mathcal{X}} : \mathcal{X} \times \mathcal{X} \to \mathcal{R}$ and $d_{\mathcal{Y}} : \mathcal{Y} \times \mathcal{Y} \to \mathcal{R}$ and for a* distortion factor *k,*

$$\forall x_1, x_2 \in \mathcal{X} : d_{\mathcal{X}}(x_1, x_2) \leq d_{\mathcal{Y}}(f_1(x_1), f_2(x_2)) \leq k \cdot d_{\mathcal{X}}(x_1, x_2)$$

For many settings we shall see, $f_1 = f_2$ (there is a single embedding function), although we allow for this kind of "non-symmetric" embedding for full generality. We shall also make use of *probabilistic embeddings*: these are embeddings for which there is a (small) probability that any given pair of elements will be stretched more than k. Formally,

Definition 1.1.3 *A* probabilistic embedding *is defined by functions $f_1, f_2 : \mathcal{X} \times \{0,1\}^R \to \mathcal{Y}$ so that for some constant δ and r picked uniformly from $\{0,1\}^R$:*

$$\forall x_1, x_2 \in \mathcal{X} : \Pr[d_{\mathcal{X}}(x_1, x_2) \leq d_{\mathcal{Y}}(f_1(x_1, r), f_2(x_2, r)) \leq k \cdot d_{\mathcal{X}}(x_1, x_2)] \geq 1 - \delta$$

The probability is taken over all choices of r, where r represents R bits which can be chosen uniformly at random to ensure that this embedding is not affected by adversarially chosen input.

1.2 Sets and Vectors

Sets and vectors are two of the most basic combinatorial objects, and as such there are just a few natural operations upon them. We shall begin by discussing basic set operations on sets A and B drawn from a finite universe. We will write $|A|$ for the number of elements in the set A.

1.2.1 Set Difference and Set Union

Definition 1.2.1 *The* set difference *between two sets, A and B is*

$$A \backslash B = \{x | x \in A \wedge x \notin B\}$$

Definition 1.2.2 *The* union *of two sets A and B is*

$$A \cup B = \{x | x \in A \vee x \in B\}$$

1.2.2 Symmetric Difference

Definition 1.2.3 *The* symmetric difference *between sets A and B is*

$$A\Delta B = \{x | (x \in A \wedge x \notin B) \vee (x \notin A \wedge x \in B)\}$$

A natural measure of the difference between two sets is the size of their symmetric difference. In fact, this is an example of an editing distance: to transform a set A into a set B we can remove elements from set A or add elements from the universe to the set A. This editing distance is exactly the symmetric difference $A\Delta B$, since the most efficient strategy is to remove everything in A but not in B (the set $A\backslash B$) and to insert everything in B but not in A (the set $B\backslash A$). Note that set difference is not really a primitive operation, since it can be defined in terms of our previous set operations, $A\Delta B = (A\backslash B) \cup (B\backslash A)$.

1.2.3 Intersection Size

Definition 1.2.4 *The* intersection *of sets A and B is*

$$A \cap B = \{x | x \in A \wedge x \in B\}$$

We shall consider the size of the intersection of two sets, $|A \cap B|$. This measure is quite contrary to the usual notion of distance, since $|A \cap A| = |A|$, whereas if A and B have no elements in common (so they are completely different) then $|A \cap B| = 0$. So this size is certainly not a metric.

1.2.4 Vector Norms

Throughout, we shall use $\boldsymbol{a}$ and $\boldsymbol{b}$ to denote vectors. The length of a vector (number of entries) is denoted as $|\boldsymbol{a}|$. The concatenation of two vectors, $\boldsymbol{a}||\boldsymbol{b}$ is the vector of length $|\boldsymbol{a}| + |\boldsymbol{b}|$ consisting of the entries of $\boldsymbol{a}$ followed by the entries of $\boldsymbol{b}$.

Definition 1.2.5 *The L_p* norm *on a vector $\boldsymbol{a}$ of dimension n is*

$$||\boldsymbol{a}||_p = (\sum_{i=1}^{n} |\boldsymbol{a}_i|^p)^{1/p}$$

We can use this norm to define distances between pairs of vectors:

Definition 1.2.6 *The L_p* distance *between vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ of dimension n is the L_p norm of their difference,*

$$||\boldsymbol{a} - \boldsymbol{b}||_p = (\sum_{i=1}^{n} |\boldsymbol{a}_i - \boldsymbol{b}_i|^p)^{1/p}$$

Note that this definition is symmetric, and defines a metric when p is a positive integer. Of particular interest to us will be the L_2 and L_1 distances. If we treat the vectors as defining points in n dimensional space then the L_2, or Euclidean, distance between two points gives the length of the straight line joining those points. The L_1, or Manhattan, distance gives the sum of the differences in coordinates. It gives the length of a shortest path between the points which travels only parallel to the axes. Unlike other objects we shall see, vectors are more continuous than discrete, and the L_p distance will be in $\mathcal{R}$ rather than $\mathcal{Z}$. However, if we restrict the vectors to having only integer entries, then for a vector $\boldsymbol{a}$, $||\boldsymbol{a}||_1$ and $||\boldsymbol{a}||_2^2$ (the L_2 norm squared, sometimes written L_2^2) will be in $\mathcal{N}$.

Vector Hamming Distances

Definition 1.2.7 *The* Hamming norm *of a vector* $\boldsymbol{a}$*, denoted by* $||\boldsymbol{a}||_H$ *is the number of places in which it is not zero.*

From this we can find the distance between two vectors as the norm of their difference.

Definition 1.2.8 *The* Vector Hamming distance, $||\boldsymbol{a} - \boldsymbol{b}||_H$*, between vectors* $\boldsymbol{a}$ *and* $\boldsymbol{b}$ *of length* n *is*

$$\sum_{i=1}^{n} (\boldsymbol{a}_i \neq \boldsymbol{b}_i)$$

Here, we implicitly extend $x \neq y$ as a function onto $\{0, 1\}$: it is 0 if $x = y$ and 1 otherwise. This can be cast as a special case of an L_p distance: the vector Hamming distance can be thought of as being related to the L_0 distance since $(\boldsymbol{a}_i - \boldsymbol{b}_i)^0 = (\boldsymbol{a}_i \neq \boldsymbol{b}_i)$. This is to be compared to the String Hamming distance defined later. We will also consider a variation on the Vector Hamming distance, which we call the zero-based Hamming distance.

Definition 1.2.9 *The* Zero-based Hamming distance *is defined as*

$$H_0(\boldsymbol{a}, \boldsymbol{b}) = \sum_{i=1}^{n} (\boldsymbol{a}_i = 0 \wedge \boldsymbol{b}_i \neq 0) \vee (\boldsymbol{a}_i \neq 0 \wedge \boldsymbol{b}_i = 0)$$

These measures — L_p distance, symmetric difference, Vector Hamming distance and set intersection size — define very well understood spaces. In Chapter 2 we shall explore them further and look at approximation results in these spaces. A major theme of this thesis will be to take new problems and show how they can be answered by recourse to these basic distances via embeddings.

1.3 Permutations

A permutation is an ordering of n symbols such that within a permutation each symbol is unique. We shall often represent these symbols as integers drawn from some range (usually $\{1, \ldots, n\}$), so 1 3 2 4 is a valid permutation, but 1 2 3 2 is not. We will name our permutations $P, Q, \ldots$. The i'th symbol of a permutation P will be denoted as $P[i]$, and the inverse of the permutation P^{-1} is defined so that if $P[i] = j$ then $P^{-1}[j] = i$. We can also compose one permutation with another, so $(P \circ Q)[i] = P[Q[i]]$. The "identity permutation" I is the permutation for which $I[i] = i$ for all i.

Permutations are of interest for a number of reasons. They are fundamental combinatorial objects, and so comparing permutations is a natural problem to study. They are also a specialisation of strings, as we shall see later, and so studying distances between permutations can help in the study of corresponding distances on strings. Sorting algorithms and circuits manipulate permutations with compare-and-swap since the sorted order is merely a permutation of the input.

Permutations are also often used in computational biology applications. Comparative studies of gene locations on the genetic maps of closely related species help us to understand the complex phylogenetic relationships between them. If the genetic maps of two species are modelled as permutations of (homologous) genes, the number of chromosomal rearrangements in the form of deletions, block moves, inversions and so on to transform one such permutation to another can be used as a measure of their evolutionary distance.

In computing permutation distances between pairs of permutations, previous approaches have uniformly chosen one sequence as the 'goal sequence'. One can perform a relabelling on both, so that the goal sequence is the identity permutation, and the problem reduces to sorting the other modified sequence. Hence when the operations to edit the permutations are based on reversing or transposing subsequences, these problems are often known as "sorting by reversals" and "sorting by transpositions". Clearly, relabelling both permutations consistently does not alter the distance between them, since we are just changing labels. We shall later see why this step is not always possible. In the following sections, we shall refer to several earlier works on these kinds of problems, although our focus is somewhat different to theirs. In particular, they seek to find a sequence of operations to sort a permutation whose number is an approximation to the distance. Our main concern is just to find the distance approximation, not a set of operations that achieves this bound.

We now describe the different permutation distances that we focus on. These are defined by describing the editing operations that can be performed on the permutations. There is a very large space of potential editing distances defined by arbitrary selections of editing operations. We pick out a few based on a 'meaningful' set of operations: reversals, transpositions, swaps, moves and combinations of these. A survey of metrics on permutations from a mathematical perspective is given by Deza and Huang in [DH98].

1.3.1 Reversal Distance

Definition 1.3.1 *The* Reversal Distance *between two permutations, $r(P, Q)$ is defined as the minimum number of reversals of contiguous subsequences necessary to transform one permutation into another. So if P is a permutation, $P[1] \dots P[n]$, then a reversal operation with parameters i, j $(1 \leq i \leq j \leq n)$ results in the permutation $P[1] \dots P[i-1], P[j], \dots P[i+1], P[i], P[j+1] \dots P[n]$.*

Example. The following is a reversal on the permutation 7 3 4 1 5 6 2 8 with parameters $4, 7$.

$$7 \quad 3 \quad 4 \quad \underbrace{1 \quad 5 \quad 6 \quad 2} \quad 8 \quad \longrightarrow \quad 7 \quad 3 \quad 4 \quad 2 \quad 6 \quad 5 \quad 1 \quad 8$$

Because Reversal Distance is an editing distance, it is consequentially a metric. For this distance to be well defined, we require that P and Q are permutations of each other, that is, that they have exactly the same set of symbols. Later we shall relax this requirement for a generalised notion of reversal distance.

Background and History Finding the reversal distance is frequently motivated from biological problems, since genes of a chromosome can be distinguished and given distinct labels. In certain situations, the prime mutation mechanism acts by reversing the order of a contiguous subsequence of genes [Gus97]. Hence the reversal distance between two sequences is a good indicator of their genetic similarity. The problem attracted a lot of interest in the mid-nineties, although a similar distance arose earlier in the context of "pancake flipping" [GP79]. The current interest in sorting by reversals was set off by Kececioglu and Sankoff [KS95] who gave 2-approximation (that is, an approximation that is within a factor of 2 of the correct answer) for reversal distance based on counting 'reversal breakpoints'. Later work improved the approximation to 7/4 by introducing the notion of permutation graphs, and decomposing the graph into cycles [BP93]. A modified version of the same approach improved the approximation to 3/2 [Chr98a], and subsequently improved to 11/8 [BHK01]. Reversal distance has been shown to be NP-hard to find exactly [Cap97] and subsequently to be Max-SNP hard [BK98]. However, the related problem of sorting signed permutations by reversals has been shown to be solvable in polynomial time [HP95] and in fact in linear time [BMY01].

1.3.2 Transposition Distance

Definition 1.3.2 *The* Transposition Distance *between two permutations, $t(P, Q)$, is defined as the minimum number of moves of contiguous subsequences to arbitrary new locations necessary to transform one permutation* *into the other. Given $P[1]\dots P[n]$, a transposition with parameters i, j, k $(i \le j \le k)$ gives*

$$P[1]\dots P[i-1], P[j], P[j+1]\dots P[k], P[i], P[i+1]\dots P[j-1], P[k+1]\dots P[n]$$

Example. The following is a transposition on the permutation 7 3 4 1 5 6 2 8 with parameters $2, 4, 7$.

$$7\ \ \underbrace{3\ \ 4}\ \underbrace{1\ \ 5\ \ 6\ \ 2}\ \ 8 \quad \longrightarrow \quad 7\ \ 1\ \ 5\ \ 6\ \ 2\ \ 3\ \ 4\ \ 8$$

Transposition distance is also motivated from a Biological perspective. Like Reversal distance, it is an editing distance, and it has received much attention in recent years. The complexity of transposition distance is unknown, although it is widely conjectured to be NP-Hard [Chr98b]. As with Reversal distance, constant factor approximation schemes have been proposed. By counting breakpoints in the permutations (see Chapter 3), it is easy to come up with a 3-approximation. The same measure was shown to be a $9/4$ approximation in [WDM00] and this was improved to a 2 approximation in [EEK$^+$01]. The best known approximation factor for Transposition distance is $3/2$ [BP98, Chr98b]. These cited works also use a similar idea to approximations of reversal distance, to build a graph based on a permutation and examine the decomposition of this graph into cycles.

1.3.3 Swap Distance

For permutation distances, one could easily define new distances by choosing an arbitrary set of permitted operations and taking the distance as the minimum number of these operations transforming one sequence into another, generating an editing distance. In general computing the distance under such a definition will be NP-Hard (that is, the problem is NP-Hard when the set of operations is part of the input) [EG81] and in fact NSPACE-complete [Jer85]. Instead, we choose to focus on sets of operations that are not arbitrary, but have some relevant motivation, such as reversals and transpositions as described above. The intersection of reversals and transpositions are swaps, which interchange two adjacent elements of a permutation.

Definition 1.3.3 *The* Swap Distance, $\mathrm{swap}(P, Q)$, *is the minimum number of transpositions of two adjacent characters necessary to transform one permutation into another. Given a permutation P, a swap at location i generates the new permutation $P[1]\dots P[i-1], P[i+1], P[i], P[i+2]\dots P[n]$.*

It is straightforward to show how the Swap Distance can be computed exactly by counting the number of *inversions*. We will later see how to compute this distance in other contexts, and how it relates to a similar distance defined on strings.

1.3.4 Permutation Edit Distance

Definition 1.3.4 *The permutation edit distance between two permutations, $d(P, Q)$ is the minimum number of moves required to transform P into Q. A move can take a single symbol and place it at an arbitrary new position in the permutation. Hence a move with parameters i, j $(i < j)$ turns $P[1]\dots P[n]$ into $P[1]\dots P[i-1]$, $P[i+1]\dots P[j], P[i], P[j+1]\dots P[n]$. A move with parameters i, j $(i > j)$ turns P into $P[1]\dots P[j], P[i], P[j+1]\dots P[i-1], P[i+1]\dots P[n]$*

This can be seen as only allowing transpositions where one of the blocks being moved is of length one. This distance is analogous to the Levenshtein edit distance on strings (given later in Definition 1.4.3), because it is closely related to the longest common subsequence of the two sequences. Let $LCS(P, Q)$ represent the length of the longest common subsequence of P and Q. An optimal edit sequence will have length $n - LCS(P, Q)$: every element that is not moved must form part of a common subsequence of P and Q and so an optimal edit scheme will ensure that this common subsequence is as long as possible. Hence, in an online environment, it is very easy to use dynamic programming to calculate this distance exactly in polynomial time: simply calculate the length of the longest common subsequence, and subtract this from n. This distance is sometimes also referred to as the Ulam Metric.

1.3.5 Reversals, Indels, Transpositions, Edits (RITE)

Each of the above distances can be augmented by additionally allowing insertions and deletions of a single symbol at a time. This takes care of the fact that the alphabet set in two permutations need not be identical. It will be of interest to (1) combine various operations (transposition, reversal, symbol moves) and define the respective distance between any two permutations involving the minimum number of operations, and (2) generalise the definitions so that at most one of P or Q is a string (as opposed to a permutation). If both P and Q are strings then this is an instance of string distances, covered below. Collectively, these distances are referred to as Reversals, Indels, Transpositions and Edits, or 'RITE' distances. These are described in more detail in Chapter 3.

Various approaches have been taken to these hybrid distances, including a technique described in [HP95] which combines reversals with translocations (prefix and suffix reversals) and other operations. In [GPS99] a 2-approximation is given for a distance which allows transpositions, reversals and a combined transposition-reversal (a block is moved and reversed in one operation). Closest to our concept of RITE distances is [WDM00] which gives a 3-approximation for a combination of reversals and transpositions.

1.4 Strings

Definition 1.4.1 *A string is a sequence of characters drawn from a finite alphabet σ. The length of a string a is denoted by $|a|$. A substring of a string a from position l to position r is the string formed by concatenating the $r - l + 1$ contiguous characters of a from position l. It is written $a[l : r]$, and so $a[l : r] = a[l], a[l+1], \ldots a[r-1], a[r]$.*

Strings are such fundamental structures in Computer Science, and their uses so many and varied, that it is hard to give a representative account of them. Any text is a string, so any word processing document, web page or sequence of DNA can be treated as a string, and manipulated using the techniques developed in the areas of string editing and matching. String searching has traditionally been an active area of research with many applications in Compilers, Text Editors and other systems. It has a long history in Computer Science theory tracing back to the first stored program computers. The first string problems that were studied dealt with finding exact occurrences of one or more query strings or regular expressions within single or multiple texts. As applications of string searching diversified, more involved string searching was considered during the 1980s, such as finding approximate occurrences of a string within texts; different notions of approximate occurrences such as various edit distances were isolated. New focus emerged in the last decade in areas of Computational Biology, Information Retrieval and Web Search Engines which required string searching on an unprecedented scale. New algorithmic string problems were studied including string searching in presence of dynamic changes to text, compressed matching, and subsequence searching. Now, as these application areas have matured, some of the largest data stores have come to comprise string data: collections of web documents, online technical libraries of text and music, molecular and genetic data, and many other examples. For a wider overview of strings and string distances, see [SK83] and [Gus97].

1.4.1 Hamming Distance

The first and most fundamental string distance is the Hamming distance, proposed by Hamming in the 1950's (see, for example, [Ham80]).

Definition 1.4.2 *The Hamming distance, h, between two strings a, b of the same length is the number of locations in which they differ. So*

$$h(a,b) = |\{i | a[i] \neq b[i]\}|$$

Implicitly, we assume that both strings are drawn from the same alphabet σ. We can cast Hamming distance as an editing distance: the editing operation is to change a character at a particular location. Hence Hamming distance defines a metric on strings.

Example. The Hamming distance between these two strings is $h(a,b) = 3$.

```
a = 1 0 1 1 1 0 1 0
b = 1 1 1 0 1 0 0 0
```

Hamming distance is of particular interest in coding theory. In synchronous communication channels, the major cause of error is if a one bit is mistakenly read as a zero bit, or vice-versa. The Hamming distance between the original signal and that which was received indicates the error in transmission (ideally, it should be zero). Codes are designed to correct up to a certain number of 'Hamming errors' (places where a bit has been misread), since the Hamming distance best captures the nature of the corruption that happens. Hamming codes [Ham80] can detect and correct a single error in a block of bits, whereas more sophisticated Reed-Solomon and BCH codes can correct higher numbers of errors and "burst errors" (a large number of errors happening in close proximity) [MS77, PW72].

1.4.2 Edit Distance

Almost as fundamental as Hamming distance is the String Edit distance, often also known as Levenshtein distance, after the first known reference to such a distance [Lev66].

Definition 1.4.3 *The Edit distance between two strings, $e(a,b)$ is the minimum number of character inserts, deletes or replacements required to change one string into another. Formally, these operations on a string a of length n are:*

- *Character inserts with parameters i, x: this transforms a into* $a[1] \ldots a[i-1]\ x\ a[i] \ldots a[n]$.
- *Character deletes with parameter i: this transforms a into* $a[1] \ldots a[i-1]\ a[i+1] \ldots a[n]$.
- *Character replacements with parameters i, x: this turns a into* $a[1] \ldots a[i-1]\ x\ a[i+1] \ldots a[n]$.

There are variations of the edit distance, depending on whether character replacements are allowed or not (if not, then they can be simulated by a delete followed by an insertion). In the case where replacements are not allowed, then as with permutation edit distance, the editing distance can be found by finding the longest common subsequence of the two strings. Everything in the subsequence is preserved; then characters in the first string but not in the common subsequence are deleted, and characters in the second string but not in the common subsequence are inserted. So here $e(a,b) = |a| + |b| - 2LCS(a,b)$.

Example. The edit distance between strings `algorithms` and `logarithm` allowing only insertions and deletions is found by isolating a longest common subsequence.

```
a   l   g   o   r   i   t   h   m   s
  /     |       |   |   |   |   |
l   o   g   a   r   i   t   h   m
```

The edit distance is the sum of the lengths of the two sequence, less twice the length of the longest common subsequence. Here, this is $10 + 9 - 14 = 5$. A sequence of editing operations to make the transformation will delete the first, fourth and tenth characters of the original string, then insert `o` between the first and second characters, and `a` between the second and third. Since this is a metric, the transformation can be carried out by inverting each of the operations — turning inserts into deletes, and vice-versa.

History and applications The edit distance originally arose in the scenario of errors on communication channels, where characters get inserted or deleted to the message being communicated [Lev66]. It has also arisen in the context of biological sequences which are mutated by characters being added or removed [Gus97] and from typing errors being made by humans writing documents [CR94]. For both versions (with and without character replacements) the edit distance can be calculated by standard dynamic programming techniques. This solution has been discovered independently several times over the past thirty years. The cost of this procedure is $O(mn)$ if the strings are length $|a| = n$ and $|b| = m$ respectively. An improvement, using the so-called "Four Russians" speed up can improve this to $O(nm/\log(m))$, by precalculating sub-arrays of the dynamic programming table [MP80]. The cost can be expressed in terms of the distance d itself, in which case the cost is $O(d \max(m, n))$, although in the worst case this is $O(mn)$ again [Gus97]. Many variations have been considered, such as additionally allowing swaps of adjacent characters [Wag75] and multiple sequence alignment [Gus97].

1.4.3 Block Edit Distances

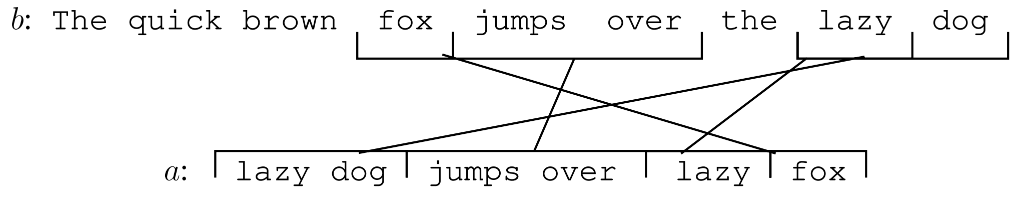
Hamming distance and (Levenshtein) edit distance are both character edit distances — that is, every operation affects only a single character at a time. These correspond to the Permutation Edit Distance and Swap Distance described above: each permitted editing operation only affects a single character, although the domain here is strings rather than permutations. Another class of string editing distances are Block Edit Distances. These manipulate arbitrarily large substrings (or blocks) at a time. These are analogous to the (permutation) reversal and transposition distances we saw above. There are many applications where substring moves are taken as a primitive: in certain computational biology settings a large subsequence being moved is just as likely as an insertion or deletion; in a text processing environment, moving a large block intact may be considered a similar level of rearrangement to inserting or deleting characters. We go on to describe a variety of edit distances which incorporate block operations: these will apply to different situations.

Tichy Distance One of the first attempts at using block operations was given in [Tic84]. Here, the problem attempted was to describe one string as a sequence of blocks that occur in another. From this we can induce a distance measure:

Definition 1.4.4 *The* Tichy distance *between two strings a and b is the minimum number of substrings of b that a can be parsed into. It is denoted* $\mathrm{tichy}(a, b)$.

If $\mathrm{tichy}(a, b) = d$ then this means that $a = b_1 b_2 \dots b_d$ where each string b_i is some substring of b, that is, $b_i = b[l_i : r_i]$.

Example. The Tichy distance from the first string to the second is 4.



b: The quick brown fox jumps over the lazy dog
a: lazy dog jumps over lazy fox

We can immediately see that the Tichy distance is not a metric. Still, it is a convenient measure of similarity, since it can be easily computed. As described in [Tic84], a greedy algorithm suffices, by starting at the left end of a and repeatedly finding the longest substring of b that matches the unparsed section of a. This parsing can be carried out in $O(|a| + |b|)$ time and space by building a suffix tree[1] of a in linear time, and repeatedly searching this from the root, taking one step for each character in a. This first block edit distance is also part of a class we loosely describe as "compression distances", since it corresponds closely to a compression algorithm: a is compressed using b as a dictionary. It means that if someone holds the string b but does not know a then a can be communicated to them efficiently by listing the parsing of a implied by finding the Tichy distance between a and b. A similar class of distance is discussed in [LT97]. However, it is shown that with only mild changes to the definitions, problems of finding block edit distances between strings are NP-Hard. This gives us the intuition that in general with a few exceptions these distances will be hard to find exactly.

LZ Distance The LZ distance is a development of the Tichy distance. It is introduced formally here and in [CPṢV00].

Definition 1.4.5 *The LZ distance between two strings a and b is the minimum number of substrings that a can be parsed into where each substring is either a substring of b or a substring that occurs earlier (to the left) in a. It is denoted $lz(a, b)$.*

This distance further develops the idea of relating distance between strings to compressibility. It has been applied to comparing texts in different languages and DNA sequences [BCL02] and in exchanging information efficiently [Evf00]. The idea is that the distance between a and b should relate to the shortest possible description of a using knowledge of b. Note that if b is empty, then the LZ distance is proportional to the size of the optimal Lempel-Ziv compressed form of the string a [ZL77]. The parsing can be computed by adapting an algorithm that performs Lempel-Ziv compression, and running it on the compound string ba (b followed by a), outputting just the parsing of a. Such an optimal parsing can be computed greedily in linear time [Sto88]. However, this measure is still not a metric, hence the introduction of the compression distance below.

Compression Distance The power of distances like the LZ distance and the Tichy distance comes from the ability to copy long blocks at unit cost. We show how to define a distance on strings that has this operation and is also a metric. This metric we introduce here and in [CPṢV00]. We call this 'compression distance' because of its relation to lossless compression, which we shall go into further later. Suppose we allow copying as a basic operation in our editing metric. Then, because the relation we define must be symmetric, we have to define the inverse of a copy operation. We refer to this as an 'uncopy': this is the removal of a substring provided that a copy of this substring remains in the string. Because we want this distance to be defined between all pairs of strings we also need to

[1]A suffix tree is a compressed trie containing all suffixes of a string. See [Wei73, McC76, Gus97] for how this can be accomplished in time linear in the length of the string

include character insert and delete operations (otherwise the distance between a string consisting of the single character x and one consisting of the character y is undefined). It is natural to think of this as a distance that measures how many "editing" operations separate two documents, where the operations are like the "cut and paste" and "copy" operations supported by word processors, as well as the basic character editing operations. This distance also has applications to computational biology settings and constructing phylogenies of texts and languages [LCL$^+$03] and other pattern matching areas [LT97]. We now give a formal definition of this distance.

Definition 1.4.6 *The* Compression Distance *between two strings a (of length n) and b, $c(a, b)$, is the minimum number of the following operations to transform a into b:*

- *Substring copies with parameters $1 \le i \le j \le n, k$*[2] *: this turns a into* $a[1] \dots a[k-1],\ a[i] \dots a[j],\ a[k] \dots a[n]$.
- *Substring uncopies with parameters $1 \le i \le j \le n$: this transforms a into* $a' = a[1] \dots a[i-1],\ a[j+1] \dots a[n]$ *provided that $a[i:j]$ is a substring of a'.*
- *Character inserts with parameters i, x: this transforms a into* $a[1] \dots a[i-1],\ x,\ a[i] \dots a[n]$.
- *Character deletes with parameter i: this transforms a into* $a[1] \dots a[i-1],\ a[i+1] \dots a[n]$.

Variations of Compression Distance It is reasonable to allow additional editing operations for variants of the Compression distance. For example, we may further allow character replacements and substring transpositions (moving a substring from one place to another in the string) as primitive operations, although these can be simulated by combinations of the core operations: a move is a copy followed by an uncopy, for example. By analogy with Permutation distances, we may also allow substrings to be reversed. Finally, we may permit arbitrary substrings to be deleted rather than the restricted deletions from the uncopy operation — note that if this operation is allowed then the induced distance is not a metric. Formally, these additional operations are defined as followed:

- Character replacements with parameters i, x turns a into $a[1] \dots a[i-1],\ x,\ a[i+1] \dots a[n]$.
- Substring moves with parameters $i \le j \le k$ transforms a into $a[1] \dots a[i-1],\ a[j] \dots a[k-1],\ a[i] \dots a[j-1],\ a[k] \dots a[n]$.
- Reversals with parameters $1 \le i \le j \le n$ turns a into $a[1] \dots a[i-1],\ a[j],\ \dots a[i],\ a[j+1] \dots a[n]$.
- Substring deletions with parameters $1 \le i \le j \le n$ turns a into $a[1] \dots a[i-1],\ a[j+1] \dots a[n]$.

Definition 1.4.7 Compression distance with unconstrained deletes *between strings a and b is defined as the minimum number of character insertions, deletions or changes, or substring copies, moves or deletions, necessary to turn a into b*

Although the question is still open, it is conjectured that the problem of computing any of these distances between a pair of strings is NP-Hard. This intuition comes from the related hardness results for similar distances on strings [LT97, SS02] and permutations [Cap97, BK98].

[2] *Note that k can lie between i and j, in which case the operation is equivalent to two copies, with parameters (k, j, k) and (i, k, k).*

String Edit Distance with Moves A final block edit distance arises when trying to answer the question, what is the simplest metric block edit distance, which is the most intuitive? The operation of "uncopying" feels somewhat unnatural, and copying is perhaps unsuited for some settings where sequences are manipulated by mutations which only rearrange [SS02]. The string edit distance with moves is introduced here and in [CM02]. It takes the string edit distance (Definition 1.4.3) and augments it with a single block edit operation, of moving a substring. It has recently been investigated further, and shown to be NP-hard [SS02].

Definition 1.4.8 *The string edit distance with moves between two strings, $d(a, b)$ is the smallest number of the following operations to turn string a into string b:*

- *Character inserts with parameters i, x: this transforms a into* $a[1] \ldots a[i-1],\ x,\ a[i] \ldots a[n]$.
- *Character deletes with parameter i: this transforms a into* $a[1] \ldots a[i-1],\ a[i+1] \ldots a[n]$.
- *Substring moves with parameters $1 \leq i \leq j \leq k \leq n$ transforms a into* $a[1] \ldots a[i-1],\ a[j] \ldots a[k-1],\ a[i] \ldots a[j-1],\ a[k] \ldots a[n]$.

This distance is by definition a metric. Note that there is no restriction on the interaction of edit operations so, for example, it is quite possible for a substring move to take a substring to a new location and then for a subsequent move to operate on a substring which overlaps the moved substring and its surrounding characters. This leads to a very powerful and flexible distance measure.

1.5 Sequence Distance Problems

Now that we have defined a variety of sequence distances, we can go on and introduce a number of problems that are based on these distances. These are problems of Efficient Communication, which asks for these embeddings to be used by distributed parties; Approximate Pattern Matching, which generalises the well-studied problems of string matching; Approximate Searching and Clustering problems which generalise problems from computational geometry.

1.5.1 Efficient Computation and Communication

The most basic problem on these distances is to compute them efficiently. We can use the embeddings to assist in this goal: our embeddings are designed to be efficiently computable (in time at most polynomial in the length of the original sequences). They give guaranteed approximations of the distances, even if there are no known efficient ways to find these distances exactly. So approximations to the distances of interest should be relatively easy to compute, by using the embeddings. A rather more challenging goal is to allow an approximation to the distance to be computed in a distributed model of computing. There are many variations of this model of computation, but we will use the most straightforward: that there are two people, A and B, who each hold a sequence (a and b). The first problem is for A and B to communicate so that they find out (an approximation to) the distance between a and b. The goal is for this communication to be as small as possible, and certainly much smaller than the cost of sending either sequence to the other person.

A further problem in this model is for A and B to communicate to allow B to discover what A's sequence is. In general, this kind of problem cannot be solved for every possible a and b without sending an amount of information linear in the size of a and b. However, we will use the intuition that if the distance between a and b is small, then a can be described to B using a much smaller amount of communication. For example, if a represents a new version of a web page that B has cached an old version of, we would hope that there was an efficient way to use this fact to save having to send the whole page again. In particular, if we measure the distance between a and b using an editing distance, then we could describe a by listing the editing operations necessary to turn b into a. We will therefore look for solutions that communicate a to B using an amount of communication that is parameterised by the distance between a and b. These may be achieved in a single communication between A and B or in a series of rounds of interactive communication.

1.5.2 Approximate Pattern Matching

String matching has a long history in computer science, dating back to the first compilers in the sixties and before. Text comparison now appears in all areas of the discipline, from compression and pattern matching to computational biology and web searching. The basic notion of string similarity used in such comparisons is the Levenshtein edit distance between pairs of strings, although other distances are also appropriate.

We can define a generalised problem of approximate pattern matching for any sequences. We consider the abstract situation of having one long sequence (the text) and wishing to find the best way of aligning a shorter sequence (the pattern) against each position in the text. The quality of an alignment can be measured using an appropriate sequence distance measure.

Definition 1.5.1 *The problem of approximate pattern matching is, given a pattern $P[1:m]$ and a long text $T[1:n]$, find $D[i]$ for each $1 \leq i \leq n$. $D[i]$ is defined as $\min_j d(P, T[i:j])$ for a given sequence distance d.*

The choice of distance d then gives different instances of this general problem. For Hamming distance, this yields the familiar "string matching with mismatches" problem (see [Gus97] for a discussion of this problem). Here the issue of alignment is not significant, since for each $i \leq n - m + 1$ then $D[i] = h(P, T[i : i + m - 1])$. This problem can be trivially solved in time $O(mn)$. Faster solutions can be achieved in time $O(|\sigma| n \log m)$ by using Fast Fourier transforms [FP74], and in time $O(n\sqrt{m \log m})$ using other combinatorial approaches [Abr87]. We shall generally be interested not in the exact version of this problem, but rather the approximate version, where for each $D[i]$ we find an approximation $\hat{D}[i]$. For the approximate version of string matching with mismatches, Karloff gives an algorithm to find a $(1 + \epsilon)$ approximation in time $O(\frac{1}{\epsilon^2} n \log^3 m)$ time [Kar93]. This has been improved by the use of embeddings to $O(\frac{1}{\epsilon^2} n \log n)$ in [Ind98, IKM00].

For the string edit distance, this problem has also been well studied. In particular, the variation where we only wish to find $D[i]$ if $D[i] < k$ for some parameter k has been addressed, with variations of the standard dynamic programming algorithm being used to solve this in time $O(kn)$ [LV86, Mye86]. However, for the general problem there are no solutions known for edit distance that beat the $O(mn)$ (quadratic) bound, even allowing approximation of the $D[i]$s.

In later chapters we shall give approximation algorithms for approximate pattern matching under a variety of distance measures, including string edit distance with moves, transposition, reversal and permutation edit distances. These will all take advantage of a "pruning lemma" given in [CM02] that allows certain editing distances to be approximated up to a constant factor by only considering a single alignment. This lemma can be applied to any editing distance that has the following property: the only way to change the length of a sequence is by unit cost symbol insertions and deletions. Given a distance metric d that has this property, we can state the following lemma:

Lemma 1.5.1 Pruning Lemma from [CM02] *Given any pattern P and text T, for all $l \leq r \leq n$,*

$$d(P, T[l : l + m - 1]) \leq 2\, d(P, T[l : r]).$$

Proof. Observe that for all r in the lemma, $d(P, T[l : r]) \geq |(r - l + 1) - m|$ since this many characters must be inserted or deleted. Using the triangle inequality since d is a metric, we have for all r, $d(P, T[l : l + m - 1])$

$$\begin{aligned} &\leq d(P, T[l : r]) + d(T[l : r],\ T[l : l + m - 1]) \\ &= d(P, T[l : r]) + |(r - l + 1)\ -\ m| \\ &\leq 2d(P, T[l : r]) \end{aligned}$$

The inequality follows by considering the longest common prefix of $T[l : r]$ and $T[l : l + m - 1]$. □

The significance of the Pruning Lemma is that it suffices to approximate only $O(n)$ distances, namely, $d(P, T[l : l + m - 1])$ for all l, in order to solve Approximate Pattern Matching problems, up to an approximation factor of 2. This follows since we now know that $D[i] \leq d(P, T[i : i+m-1]) \leq 2D[i]$.

1.5.3 Geometric Problems

There are many different questions which are collectively referred to as Geometric Problems. These include questions relating to sets of points, shapes and objects (usually in Euclidean space) and the (Euclidean) distance between them [SU98, PS85]. However, we can generalise these problems to be meaningful for any distance measure. Examples of these problems include finding points that are close to a query point, far away from a query point, dividing the points into subsets of close points, or building spanning trees of the points. We shall consider two particular problems, Approximate Neighbors and Clustering for k-Centers[3]. Both of these problems on m points of dimensionality n can be solved using $O(m)$ comparisons of points, each comparison taking time $\Omega(n)$. However, in practical situations with m equal to thousands or millions of points in high dimensional space, solutions that are linear in the size of the data are impractical. In Chapter 2 we shall describe approximate solutions for these problems with vector distances that achieve time sublinear in the size of the collection of points. In subsequent chapters we shall see how equivalent problems using different distance metrics can be solved by using these solutions as a "black box" to which we can feed transformed versions of the problem instances. We now go on to formally define these two geometric problems.

1.5.4 Approximate Neighbors

The Nearest Neighbors problem is to pre-process a given set of points P so that presented with a query point q the closest point from P can be found quickly. There are efficient solutions to this *exact* problem for low dimensional Euclidean space (say, 2 or 3 dimensions) using data structures such as k-d trees and R-trees[GG98]. However, as the dimension of the space increases these approaches are struck by the so-called "curse of dimensionality". That is, as the dimension increases, the cost of the pre-processing stage increases exponentially with the dimension. This is clearly undesirable. To overcome this, recent attention has focused on the relaxation of this problem, to finding "ϵ-approximate nearest neighbors" (ANN):

Definition 1.5.2 *The* ϵ-Approximate Nearest Neighbors Problem *is, given a set of points P in n dimensional space and a query point q, find a point $p \in P$ such that $\forall p' \in P : d(p, q) \leq (1 + \epsilon)d(p', q)$ with probability $1 - \delta$ for a parameter δ.* [4]

[3]The US spellings of "neighbours" and "centres" are adopted throughout in deference to the fact that this is the accepted technical terminology for these problems.

[4]*Different approaches exist for how this probability is realised: in [KOR98], with probability $1 - \delta$ the data structure created is good for all possible queries; whereas in [IM98] the point returned for a query is good with probability $1 - \delta$ taken over all queries selected uniformly. There the space is the Hamming space, which is discrete.*

A satisfactory solution will have pre-processing costs polynomial in $|P| = m$ and n, and query costs sublinear in m and linear or near linear in n.[5] Note that the problem statement does not specify the nature of the distance function d, and so applies to any distance measure, so we can ask for Approximate Nearest Neighbors for permutation and string distances.

The dual problem to Nearest Neighbors search is Furthest Neighbors search. By analogy with ANN, the "ϵ-approximate furthest neighbors" (AFN) problem is defined as follows.

Definition 1.5.3 *Given a set of points P in n dimensional space and a query point q, find a point p such that $\forall p' \in P : (1+\epsilon)d(p,q) \geq d(p',q)$ with probability $1-\delta$ (over all queries).*

1.5.5 Clustering for k-centers

The general problem of clustering is to divide a set of m data points into clusters, so that each cluster contains similar points. There are many formalisations of this statement. We shall consider a commonly used formalisation. Let S be the set of objects, and we are asked to divide S into k non-overlapping subsets, $cluster = \{S_1 \ldots S_k\}$, such that the union of these S_i's is S. The goal is to minimise the size of each cluster, that is, to minimise

$$\text{spread}(cluster) = \max_i \max_{j,l \in S_i} d(j,l)$$

This problem is well-defined for any distance measure d. For this version of clustering, we will define the clusters implicitly by picking k points from the space. Each of these points defines a cluster, consisting of all the data points that are closer to this point than any of the other $k-1$ centers. Ties are broken arbitrarily, by assigning a point to any of the centers it is equidistant from.

Definition 1.5.4 *A* c-approximate *solution to the clustering problem is a set of k clusters $approx$ (any partition of S into k disjoint subsets) such that $\forall clusters : \text{spread}(approx) \leq c \cdot \text{spread}(clusters)$.*

Again, this definition is general, so this problem is equally valid for any class of combinatorial objects and an appropriate distance function d.

1.6 The Shape of Things to Come

Now that we have introduced the distances that we will study and some of the problems we would like to answer, we can go on to describe the results we shall show for these distances. Our results are of two types: firstly, embeddings of these distances into different spaces with bounded distortion; and secondly, applications of these embeddings to solve problems related to the original distance. The distances that we consider are summarised in Figure 1.2 — we include the distortion factor that we will show for embedding these distances into alternative spaces. We now outline the structure of the rest of this work: Chapters 2, 3 and 4 give results on embedding sequence distances, and Chapters 5 and 6 give some applications of these embeddings. Finally, Chapter 7 concludes with some discussions on extensions, improvements and open problems.

Chapter 2: Sketching and Streaming

We begin our study of embeddings by considering metrics in Vector Spaces with the L_p distance, and show how these can be embedded in vector spaces of much smaller dimension. These embeddings

[5] In the ANN literature, it is more usual to denote the dimensionality of the space with d and the number of objects with n. We use a different convention in keeping with our choice of n for the size of objects throughout.

Set and Vector Distances	**Description**	**Reference**	**Metric**	**Factor**
Symmetric Difference	Number of elements in only one set	[AMS99]	Yes	$1 \pm \epsilon$
Intersection Size	Number of elements in both sets	[KN97]	No	ϵn
Union Size	Number of elements in either set	[FM85]	No	$1 \pm \epsilon$
L_p distances	$(\sum_i \lvert\boldsymbol{a}_i - \boldsymbol{b}_i\rvert^p)^{1/p}$	[Ind00]	Yes	$1 \pm \epsilon$
String Distance	**Description**	**Reference**	**Metric**	**Factor**
Hamming Distance	Change individual characters	[Ham80]	Yes	$1 \pm \epsilon$
Levenshtein Edit Distance	Character operations (insert, delete, change)	[Lev66]	Yes	—
LZ Distance	Create target string by copying from source or partially built string	[CPṢV00]	No	$\log n \log^* n$
Compression Distances	Unit cost substring copy, move, uncopy	[CPṢV00]	Yes	$\log n \log^* n$
Unconstrained Delete	As Compression, with unconstrained deletes	[Evf00]	No	$\log n \log^* n$
Edit Distance with moves	Substring move, insert and delete characters	[CM02]	Yes	$\log n \log^* n$
Tichy's Distance	Create target by copying from source only	[Tic84, LT97]	No	—
Permutation Distance	**Description**	**Reference**	**Metric**	**Factor**
Permutation Edit Distance	Elementary operations (insert, delete, change)	[DH98]	Yes	$\log n$
Reversal Distance	Block reversals	[Gus97]	Yes	2
Transposition Distance	Block transpositions	[Gus97]	Yes	2
Swap Distance	Adjacent character transpositions	[Jer85]	Yes	1
RITE Distances	Reversals, Indels, Transpositions and Edits	[CMṢ01]	Yes	3

A summary of the main distances that we will discuss: for each one we give a brief description and a reference. We record whether it is a metric, and give the factor of distortion of our embedding of the distance into a smaller space.

Figure 1.2: Summary of the distances of interest

can be computed in a number of computation models, including the streaming model in which a large vector is processed in an arbitrary order one entry at a time with very small working space requirements. We go on to study set measurements, and show how these are related to vector distances. We are able to show hardness results for other set problems in a number of models including the sketching model, where a short summary of a set is created to allow approximation of the set measure. Finally, we discuss further problems in vector spaces, of clustering (organising a set of vectors into subsets of close vectors) and approximate nearest neighbors (pre-processing a set of vectors to rapidly find the approximately closest to a query vector). This chapter consists mostly of a survey of prior results of other authors, which will be used in subsequent chapters to build solutions for different sequence distances.

Chapter 3: Sorting Sequences

We next study embeddings of permutations into vector spaces. Motivated by computational biology scenarios, we study problems of computing distances between permutations as well as matching permutations in sequences, and finding approximate nearest neighbors.

We adopt the general approach of embedding permutation distances into well-known vector spaces in an approximately distance-preserving manner, and solve the resulting problems on the well-known spaces. The main results are as follows: We present the approximately distance-preserving embeddings of these permutation distances into well-known spaces. Using these embeddings, we obtain several results, including (1) communication complexity protocols to estimate the permutation distances accurately; (2) efficient solutions for approximately solving nearest neighbor problems with permutations and (3) algorithms for finding permutation distances in the streaming model. We use these embeddings to allow us to solve approximate pattern matching problems for permutation distances.

Chapter 4: Strings and Substrings

We use the approximate pattern matching problem as applied to string edit distance with moves as a guiding motivation to study problems on string distances. We seek a solution to this problem that is close to linear in the size of the input instead of the existing quadratic solutions to related problems on string edit distance. Our result is a near linear time deterministic algorithm for our version of the problem. It produces an answer that is a $O(\log n \log^* n)$ approximation of the correct answer. This is the first known significantly subquadratic algorithm for a string edit distance problem in which the distance involves nontrivial alignments.

The results are obtained by embedding strings into L_1 vector space using a simplified parsing technique we call *Edit Sensitive Parsing* (ESP). This embedding is approximately distance-preserving, and we show many applications of this embedding to string proximity problems including nearest neighbors and streaming computations with strings. We also show embeddings for other variations of this distance, including the compression distances, and block edit distances with unconstrained deletes.

Chapter 5: Stables, Subtables and Streams

We study the solution of some problems on vectors using the embeddings seen in Chapter 2 to achieve sublinear time and space. We first tackle the problem of comparing data streams. In particular, we use the Hamming norm which is a well-known measure throughout data processing. When applied to a single stream, Hamming norm gives the number of distinct items that are present in the data stream, which is a statistic of great interest in databases. When applied to a difference between a pair of streams, Hamming norm gives an important measure of (dis)similarity: the number of unequal items in the two streams. This can be used in auditing data streams that are expected to be nearly identical, and can be applied to network intrusion detection.

We also examine the problem of data mining and clustering massive data tables. Detecting similarity patterns in such data sets (e.g., which geographic regions have similar cell phone usage distribution, which IP subnet traffic distributions over time intervals are similar, etc) is of great importance. Identification of such patterns poses many conceptual challenges (what is a suitable similarity distance function for two "regions") as well as technical challenges (how to perform similarity computations efficiently as massive tables get accumulated over time) that we address. We implement methods for determining similar regions in massive tabular data. Our methods are approximate, but highly accurate as we show empirically, and they are fast, running in time nearly linear in the table size.

Chapter 6: Sending and Swapping

We address the problem of minimising the communication involved in the exchange of similar documents. This turns out to be facilitated by the embeddings we have advanced in Chapters 2, 3 and 4. We consider two users, A and B, who hold documents a and b respectively. Neither of the users has any information about the other's document. They exchange messages so that B computes a; it may be required that A computes b as well. The goal is to design communication protocols with the main objective of minimising the total number of bits they exchange; other objectives are minimising the number of rounds and the complexity of internal computations. An important notion which determines the efficiency of the protocols is how we measure the distance between a and b. We consider several metrics for measuring this distance as described in Chapter 1. For each metric, we present communication-efficient protocols, which often match the corresponding lower bounds up to a constant factor. In consequence, we obtain error-correcting codes for these error models which correct up to d errors in n characters using $O(d \cdot \text{poly-log}(n))$ bits.

Chapter 2

Sketching and Streaming

The Siege of Belgrade

An Austrian army, awfully arrayed,
Boldly by battery besieged Belgrade.
Cossack commanders cannonading come,
Dealing destruction's devastating doom.
Every endeavor engineers essay,
For fame, for fortune fighting - furious fray!
Generals 'gainst generals grapple - gracious God!
How honors Heaven heroic hardihood!
Infuriate, indiscriminate in ill,
Kindred kill kinsmen, kinsmen kindred kill.
Labor low levels longest, loftiest lines;
Men march 'mid mounds, 'mid moles, 'mid murderous mines;
Now noxious, noisey numbers nothing, naught
Of outward obstacles, opposing ought;
Poor patriots, partly purchased, partly pressed,
Quite quaking, quickly "Quarter! Quarter!" quest.
Reason returns, religious right redounds,
Suwarrow stops such sanguinary sounds.
Truce to thee, Turkey! Triumph to thy train,
Unwise, unjust, unmerciful Ukraine!
Vanish vain victory! vanish, victory vain!
Why wish we warfare? Wherefore welcome were
Xerxes, Ximenes, Xanthus, Xavier?
Yield, yield, ye youths! ye yeomen, yield your yell!
Zeus', Zarpater's, Zoroaster's zeal,
Attracting all, arms against acts appeal!

— Alaric Alexander Watts, 1797 – 1864

Chapter Outline

This chapter is concerned with setting up the basic embedding results for this thesis which will be built on in later chapters. We consider just sets and integer valued vectors (for the most part, these two can be interchanged using some simple isomorphisms), and survey the known results on embeddings for distance measurements on these objects. Almost all results reported here are due to other authors, and this should be viewed as background material for the discussions and development of solutions for questions relating to the sequence distances we consider in subsequent chapters. Some detail is provided on the methods and theorems described to give an insight into how they work. Summaries of the proofs are given when this gives further insight into how they work, and also when we build on these ideas in later chapter, which requires modifying the nature of the mechanism which is used, or altering the proof. For example, we need to understand the methods used to compute sketches of vectors, clusterings and nearest neighbors because in later chapters we will want to use these algorithms on transformed sequences and be sure that corresponding results can be proved about their accuracy and running time.

We progress as follows: Section 2.1 begins by discussing different models for computing embeddings, and how to compute probabilistic equality tests in these models. Section 2.2 takes the important class of vector L_p distances, and reports on the recent progress that has been made in computing embeddings of these in various models. In Section 2.3 we consider various measurements on pairs of sets in terms of entries in Venn diagrams, and how these can be related to vector distances. Lastly, in Section 2.4 we look at a number of so-called "geometric problems" on vector spaces, which will later be applied to other spaces by way of our embeddings.

2.1 Approximations and Estimates

Throughout this thesis we shall be concerned with finding approximations of certain quantities. We therefore need to introduce the concept of an approximation, and distinguish it from the related concept of an estimate.

Definition 2.1.1 *An* estimate *of a quantity x is a random variable $\bar{x}$ so that $\bar{x} = x$ with some probability $1 - \delta$.*

In other words, an estimate allows finding the exact answer with some probability. On the other hand, an approximation allows finding a value close to the exact answer.

Definition 2.1.2 *An* approximation *of a quantity x is a random variable $\hat{x}$ so that $a\hat{x} \leq x \leq b\hat{x}$ with probability $1 - \delta$, for constants $a, b, \delta > 0$.*

From this, we see that an estimate is a special case of an approximation with $a = b = 1$. However, we shall keep these notions distinct since in many cases it will prove to be unfeasible to estimate a quantity, but highly efficient to approximate it. We shall see several different kinds of approximation: "tunable" approximations where $a = 1 - \epsilon, b = 1 + \epsilon$ where ϵ is a parameter that can be chosen to be arbitrarily small; constant factor approximations, where $a = 1$ and b is some (small) constant; and input dependent approximations, where $a = 1$ and b is some function of the size of the input. Tunable approximations, or "(ϵ, δ)-approximations" are especially desirable.

In the case where the running time of the approximation algorithm is polynomial in the size of the object, n; depends on the quality of approximation ϵ as a polynomial in $1/\epsilon$; and depends on the fidelity δ as a polynomial in $\log 1/\delta$; then this can be thought of as a 'Fully Polynomial Randomised Approximation Scheme' or FPRAS. See Section 11.2 of [MR95] for more details of this notion of approximation schemes.

2.1.1 Sketch Model

The sketch model of computation can be described informally as a model where given an object x, a shorter "sketch" of x can be made so that comparing two sketches allows a function of the original objects to be approximated. We shall focus here on where both objects are of the same type, and the function being approximated is a distance function.

Definition 2.1.3 *A distance sketch function $sk(a, r)$ with parameters ϵ, δ has the property that for a distance $d(\cdot, \cdot)$, a specified deterministic function f outputs a random variable $f(sk(a, r), sk(b, r))$ so that*

$$(1 - \epsilon)\ d(a, b) \leq f(sk(a, r), sk(b, r)) \leq (1 + \epsilon)\ d(a, b)$$

for any pair of points a and b with probability $1 - \delta$, taken over all choices of (small) seed r, chsoen uniformly at random.

For a sketch function to be of interest, it should reduce the size of the objects. Ideally, $|sk(a, r)|$ and $|r|$ should be $O(\text{poly-log}(|a|))$. There is a strong connection between sketching and the communication complexity protocols that we shall see later: the sketch of an object a can be communicated to another party who holds b to allow the computation of $f(sk(a, r), sk(b, r))$ by separated parties. The approximation of the distance between a and b can then be performed by comparing their sketches. Alternatively, two parties can each send their sketches to a third party who can find the approximate distance between a and b without ever knowing a or b.

The term 'sketch' was introduced in [Bro98] where the distance considered was the resemblance of two sets, A and B, as $\frac{|A \cap B|}{|A|}$ or alternatively $\frac{|A \cap B|}{|A \cup B|}$. We shall see details of these and other sketch algorithms later.

2.1.2 Streaming

The model of sketching assumes that the computation has complete access to one part of the input. An alternate model is that of streaming, in which the computation has limited access to the whole data. In the streaming model, the data arrives as a stream — a predetermined sequence — but there is only a limited amount of storage space in which to hold information. It is not possible to backtrack over the stream, but instead each item must be processed in turn. Thus a stream is a sequence of n data items $z = (s_1, s_2, \ldots, s_n)$ which arrive one at a time in this order. Sometimes the number of items, n, will be known in advance, otherwise the final data item s_{n+1} will be an "end of stream" marker.

Data streams are now fundamental to many data processing applications. For example, telecommunication network elements such as switches and routers periodically generate records of their traffic — telephone calls, Internet packet and flow traces — which are data streams [BSW01, Net, GKMS01a]. Atmospheric observations — weather measurements, lightning strike data and satellite imagery — also produce multiple data streams [Uni, NOA]. Emerging sensor networks produce many streams of observations, for example highway traffic conditions [Sma, MF02]. Sources of data streams — large scale transactions, web clicks, ticker tape updates of stock quotes, toll booth observations — are ubiquitous in daily life. Data streams are often generated at multiple distributed locations. Despite the exponential growth in the capacity of storage devices, it not common for such streams to be stored. Nor is it desirable or helpful to store them, since the cost of any simple processing — even just sorting the data — would be too great. Instead they must be processed "on the fly" as they are produced.

Definition 2.1.4 *A* streaming algorithm *accepts a data stream z and outputs a random variable $str(z, r)$ to approximate a function g so that*

$$(1 - \epsilon)\ g(z) \leq str(z, r) \leq (1 + \epsilon)\ g(z)$$

with probability $1 - \delta$ over all choices of the random seed r, for parameters ϵ and δ.

The most important parameter of a streaming algorithm is the amount of working space that it uses (we include in this the size of any random bits, r, used by the algorithm). For a streaming algorithm to be of interest, then the working space must be $o(n)$, and preferably $O(\text{poly-log}\, n)$. The streaming model is a good model for dealing with massive data sets that are too large to fit entirely within computer memories. We can think of the situations where data resides on tapes, or is supplied over a network link, where random access is not possible. The notion of streaming was introduced in [HRR98], where it was applied to graphs.

Recent work has subdivided the streaming model into various sub-categories [GKMS01b]. They considered two features of the stream: whether it is ordered or unordered (the data arrives conforming to some order of some attribute); and whether it is aggregated or unaggregated (whether all the information for a particular attribute arrives together or whether it is spread out arbitrarily within the stream). Since we deal exclusively with sequences in this work, it is straightforward to map these concepts onto sequences that arrive in streams. We imagine that each element in the stream will be a pair $< i, j >$ that indicates that for the sequence a, we have $a[i] = j$. Additionally, there may be information identifying which stream the entry relates to if multiple streams are being processed concurrently — we gloss over this detail. We can then consider the four possible sub-divisions of the streaming model.

1. Ordered and aggregated — pairs $<i, j>$ arrive so that i is strictly increasing.

2. Ordered and unaggregated — pairs $< i, j >$ arrive so that i is increasing, but several js may be seen consecutively for the same i. In most cases it will be trivial to aggregate this information, and so these first two cases are essentially identical from our point of view.

3. Unordered and aggregated — pairs $<i, j>$ arrive such that there is no ordering on i, but there is at most one pair for each i.

4. Unordered and unaggregated — pairs $<i, j>$ arrive with no restrictions.

The last case, of unordered unaggregated streams, is the most general and requires some further discussion. It is not immediately clear what are the semantics of seeing multiple different pairs $<i, j_k>$ throughout the stream. For a vector, we shall usually take it to mean that $a[i]$ should be the sum of all j_k that arrive in this way. This fits neatly with the idea that $a[i]$ is assumed to be zero if no pair $<i, j>$ arrives, and that the other variations of streaming are a special case of unordered unaggregated streams. For strings and permutations, it is not so clear how to interpret unaggregated streams — perhaps that we should take the most recent pair $<i, j_k>$ to define $a[i]$. Instead, our streaming algorithms for strings and permutations will mostly work in the ordered case, where this question does not arise.

We can adapt these models of streaming for distance functions: suppose that z consists of two arbitrarily interleaved sequences, a and b, and that $g(z)$ is $d(a, b)$. Then a streaming algorithm to solve this problem is approximating some distance between a and b. It is possible that an algorithm can work in both the sketch and the streaming model. Often, a stream algorithm can be construed as a sketch algorithm, if the sketch is the contents of the streaming algorithm's working store. However, a sketch algorithm is not necessarily a streaming algorithm, and a streaming algorithm is not always a sketch algorithm. We shall see examples of streaming algorithms as applied to vector distances in this chapter.

2.1.3 Equality Testing

As a first example of sketching and streaming, we consider the basic problem of determining whether two objects are identical or not. In many places, we would like to be able to test efficiently whether two vectors (equivalently, sets or strings) are equal. We can pre-compute a *hash function* of a vector $\boldsymbol{a}$, which is $hash(\boldsymbol{a})$, and we want $hash(\boldsymbol{a}) = hash(\boldsymbol{b})$ if and only if $\boldsymbol{a} = \boldsymbol{b}$. Ideally $|hash(\boldsymbol{a})| << |\boldsymbol{a}|$.

It turns out that this goal as stated is impossible, by a simple counting argument. However, if we use probabilistic methods, and accept some chance of error, then we can achieve that for any pair $\boldsymbol{a} \neq \boldsymbol{b}$ then with high probability $hash(\boldsymbol{a}) \neq hash(\boldsymbol{b})$. This relies on choosing the function $hash$ from an appropriate distribution: we define a family of functions, from which we pick $hash$ at random. Then the probability statement holds over choices of $hash$ from the family. Of course, if $\boldsymbol{a} = \boldsymbol{b}$ then $hash(\boldsymbol{a}) = hash(\boldsymbol{b})$ since $hash$ will be a deterministic function. Such a hash function is often called a fingerprint, the idea being that if two fingerprints match then it is very likely they came from the same individual.

To put this into the same setting as our discussion of distance metrics, we consider the trivial discrete metric $d_=$ on any finite set

$$\begin{aligned} d_=(a,b) = 0 &\iff a = b \\ d_=(a,b) = 1 &\iff a \neq b \end{aligned}$$

We look for a sketch function which is an (ϵ, δ) approximation for $d_=$. Hence our fingerprints will be sketch functions for $d_=$. The parameter ϵ is no longer meaningful, since d takes on only the values 0 or 1, so this can equivalently be viewed as an estimation.

Lemma 2.1.1 *There exist fingerprint functions for vectors which represent vectors of size n bits using $O(\log n)$ bits.*

Proof. We could just use the sketches for L_1 or L_2 distance that we will later develop, since these have the property that they estimate the distance to be zero only when the vectors are identical. But we can instead consider functions designed specifically for this problem. Such functions have been used in string searching [KR87] and are described in more detail in Section 7.4 of [MR95]. We assume that the vectors have only binary entries (it is easy to represent a vector where each entry is an integer that is no more than M as a binary vector of length $\log M$ times its original length). The fingerprint of such a vector $\boldsymbol{a}$ of dimension n is

$$hash(\boldsymbol{a}) = \sum_{1}^{|\boldsymbol{a}|} \boldsymbol{a}_i 2^{i-1} \mod p$$

where p is a prime chosen uniformly at random from the range $[2 \dots (n/\delta)\log(n/\delta)]$. Over a random choice of p, the probability of two different vectors having the same fingerprint is at most $O(\delta)$ (Theorem 7.5 in [MR95]). □

These fingerprints have a useful property, that the fingerprint of a vector formed by the concatenation of two other vectors can be computed by composing their fingerprints. Let $\boldsymbol{a}||\boldsymbol{b}$ be the concatenation of two vectors $\boldsymbol{a}$ and $\boldsymbol{b}$, so $|(\boldsymbol{a}||\boldsymbol{b})| = |\boldsymbol{a}| + |\boldsymbol{b}|$.

Lemma 2.1.2

$$hash(\boldsymbol{a}||\boldsymbol{b}) = (hash(\boldsymbol{a}) + 2^{|\boldsymbol{a}|} hash(\boldsymbol{b})) \mod p$$

Proof.

$$\begin{aligned} hash(\boldsymbol{a}||\boldsymbol{b}) &= \sum_{i=1}^{|\boldsymbol{a}|+|\boldsymbol{b}|} (\boldsymbol{a}||\boldsymbol{b})_i 2^{i-1} \mod p \\ &= (\sum_{i=1}^{|\boldsymbol{a}|} \boldsymbol{a}_i 2^{i-1} + \sum_{i=1}^{|\boldsymbol{b}|} \boldsymbol{b}_i 2^{|\boldsymbol{a}|+i-1}) \mod p \\ &= ((\sum_{i=1}^{|\boldsymbol{a}|} \boldsymbol{a}_i 2^{i-1} \mod p) + 2^{|\boldsymbol{a}|}(\sum_{i=1}^{|\boldsymbol{b}|} \boldsymbol{b}_i 2^{i-1}) \mod p)) \mod p \\ &= (hash(\boldsymbol{a}) + 2^{|\boldsymbol{a}|} hash(\boldsymbol{b})) \mod p \end{aligned}$$

□

Observation 2.1.1 *Fingerprints can be computed in the unordered, unaggregated streaming model when values of the vector arrive as a stream in arbitrary, interleaved order. That is, $hash(\boldsymbol{a})$ can be computed when $\boldsymbol{a}$ is presented in the form of a stream of tuples (i, c), meaning that $\boldsymbol{a}_i = c$. The space required is the same as for the sketch model, $O(\log n)$.*

This follows by observing that when we encounter the tuple (i, c) in the stream, we can modify our hash value (initially zero) by adding on $c2^{i-1}$, and computing the result modulo p. Following processing the whole stream, we have computed $\sum_i \boldsymbol{a}_i 2^{i-1} \mod p$, since modular addition is commutative.

A less obvious advantage of using fingerprints is that they are integers which can be represented in $O(\log n)$ bits. In the commonly used RAM model of computation, it is usual to assume that quantities like this can be manipulated in time $O(1)$. They can also be used for building hash tables, allowing convenient access to items without requiring sorting or complex data structure management.

2.2 Vector Distances

We shall now consider various different approaches to approximating vector L_p distances, and see how they fit into these categories of sketching and streaming. We shall mainly be interested in vectors where the entries are (positive) integers bounded by a constant, M. We shall assume this to be the case, although some of these methods extend to when the entries are rationals. An important property that sketches of vectors may possess is that of *composability*.

Definition 2.2.1 *A sketch function is said to be* composable *if, for any pair of sketches $sk(\boldsymbol{a}, r), sk(\boldsymbol{b}, r)$ we have that $sk(\boldsymbol{a} + \boldsymbol{b}, r) = sk(\boldsymbol{a}, r) + sk(\boldsymbol{b}, r)$.*

2.2.1 Johnson-Lindenstrauss lemma

The Johnson-Lindenstrauss lemma [JL84] gives a way to embed a vector in Euclidean space into a much smaller space, with only small loss in accuracy. There have been a variety of presentations of this lemma, including [DG99] and [IM98]. We shall adopt the simplified version used in [IKM00] and elsewhere.

Lemma 2.2.1 *Let $\boldsymbol{a}, \boldsymbol{b}$ be vectors of length n. Let $\boldsymbol{v}$ be a set of k different random vectors of length n. Each component $\boldsymbol{v}_{i,j}$ is picked independently from the Gaussian distribution $N(0,1)$, then each vector $\boldsymbol{v}_i$ is normalised under the L_2 norm so that the magnitude of $\boldsymbol{v}_i$ is 1. Define the sketch of $\boldsymbol{a}$ to be a vector $sk(\boldsymbol{a}, r)$ of length k so that $sk(\boldsymbol{a}, r)_i = \sum_{j=1}^{n} \boldsymbol{v}_{i,j} \boldsymbol{a}_j = \boldsymbol{v}_i \cdot \boldsymbol{a}$. Given parameters δ and ϵ, we have with probability $1 - \delta$*

$$\frac{(1-\epsilon)||\boldsymbol{a} - \boldsymbol{b}||_2^2}{n} \leq \frac{||sk(\boldsymbol{a}, r) - sk(\boldsymbol{b}, r)||_2^2}{k} \leq \frac{(1+\epsilon)||\boldsymbol{a} - \boldsymbol{b}||_2^2}{n}$$

where k is $O(1/\epsilon^2 \log 1/\delta)$.

This lemma states that we can make a sketch of much smaller ($O(1/\epsilon^2 \log 1/\delta)$) dimension by forming the convolution of each vector with a set of randomly created vectors drawn from the Normal distribution. In its general statement, the Johnson-Lindenstrauss lemma is concerned with embedding m different vectors into a reduced space. For this, we would require that δ is smaller, so that with constant probability, *all* $m^2/2$ pairwise comparisons of vectors fall within the same bounds. Thus, we arrange that $1 - \delta' = (1 - \delta)^{m^2} = 1 - m^2\delta + O(\delta^2)$. So we must ensure that $\delta = O(m^{-2})$: hence for

m vectors, the length of the sketch vector is $O(1/\epsilon^2 \log m)$. Some care is needed here: mathematically we may assume that each entry of the sketch vector is representable with infinite precision, but in a computation setting we are concerned with the number of bits we require to achieve a sufficiently accurate representation. It was shown that $O(\log n)$ bits are sufficient by Indyk [Ind00]. We shall adopt the RAM model of computation, which states that we can manipulate quantities of this size in constant time, hence this factor will not figure in time bounds, although it will be accounted for in cases when every bit is counted (such as in communication and storage bounds).

The sketching procedure is not immediately a streaming algorithm, however recent work describes how to extend this approach to a streaming environment, and for L_1 as well as L_2 distances — see Section 2.2.4. The sketches formed by this method also have a nice property, which is that they are composable, as described in Definition 2.2.1. By the linearity of the sketch construction function, we see in this case that composability follows easily, since $sk(\boldsymbol{a} + \boldsymbol{b}, r) = sk(\boldsymbol{a}, r) + sk(\boldsymbol{b}, r)$, and also $sk(\boldsymbol{a} - \boldsymbol{b}, r) = sk(\boldsymbol{a}, r) - sk(\boldsymbol{b}, r)$. This property will be made use of in later chapters.

2.2.2 Frequency Moments

Alon, Matias and Szegedy [AMS99] (originally published as [AMS96]) gave some of the first algorithms to work in the streaming model. In this case, there is an unordered unaggregated stream of n integers in the range $1 \dots M$, so $z = (s_1, s_2, \dots s_n)$ for integers s_j. From this stream define $m_i = |\{j | s_j = i\}|$, the number of occurrences of the integer i in the stream. The paper focuses on computing the frequency moments of the stream, that is, F_k, where

$$F_k = \sum_{i=1}^{M} (m_i)^k$$

F_0 is then the number of distinct elements in the sequence, F_1 is trivially the length of the sequence, n, and F_2 is referred to as the repeat rate of the sequence. It turns out to be related closely to the L_2 norm, as we shall see shortly. An algorithm to compute F_2 is given: A vector $\boldsymbol{v}$ of length M, $\boldsymbol{v} \in \{-1, +1\}^M$, is created with entries chosen at random. Each entry in the vector is either $+1$ or -1. The distribution should be 4-wise independent, that is, for any four entries chosen at random, all possibilities from $\{-1, +1\}^4$ should be equally likely. A variable Z is initialised to zero. The stream $s_1 \dots s_n$ is processed item by item: when $s_j = i$ is seen, $\boldsymbol{v}_i$ is added to Z. So after the whole stream has been seen, $Z = \sum_{i=1}^{M} \boldsymbol{v}_i m_i$. The estimate of F_2 is given as Z^2:

$$\begin{aligned} Z^2 &= \sum_{i=1}^{M} \boldsymbol{v}_i^2 m_i^2 + \sum_{i=1}^{M} \sum_{j \neq i} \boldsymbol{v}_i m_i \boldsymbol{v}_j m_j \\ &= \sum_{i=1}^{M} m_i^2 + \sum_{i=1}^{M} \sum_{j \neq i} \boldsymbol{v}_i \boldsymbol{v}_j m_i m_j \end{aligned}$$

If the entries of vector $\boldsymbol{v}$ are pairwise independent, then the expectation of the "cross-terms" $\boldsymbol{v}_i \boldsymbol{v}_j$ is zero, and the expected result is $\sum_{i=1}^{M} m_i^2 = F_2$. By repeating this procedure $O(1/\epsilon^2)$ times with a different random $\boldsymbol{v}$ each time, and taking the average, the result can be guaranteed to be a $1 \pm \epsilon$ approximation with constant probability if the random variables have four-wise independence. By finding the median of $O(\log 1/\delta)$ such averages, this constant probability can be amplified to $1 - \delta$.

Example. Suppose the stream to be processed is $(1, 2, 1, 4)$. So $m_1 = 2, m_2 = 1, m_3 = 0, m_4 = 1$ and $F_2 = 2^2 + 1^2 + 0^2 + 1^2 = 6$. Then $Z = \boldsymbol{v}_1 + \boldsymbol{v}_2 + \boldsymbol{v}_1 + \boldsymbol{v}_4 = 2\boldsymbol{v}_1 + \boldsymbol{v}_2 + \boldsymbol{v}_4$. We find $Z^2 = 4\boldsymbol{v}_1^2 + \boldsymbol{v}_2^2 + \boldsymbol{v}_4^2 + 4\boldsymbol{v}_1\boldsymbol{v}_2 + 4\boldsymbol{v}_1\boldsymbol{v}_4 + 2\boldsymbol{v}_2\boldsymbol{v}_4 = 6 +$ cross-terms. The expectation of these cross-terms is zero: if this procedure is repeated several times, then the average of Z^2 will tend to 6, which is F_2.

The important detail in [AMS99] is that we do not wish to explicitly create the vectors $\boldsymbol{v}$, since these would consume $O(M)$ space. Instead, individual entries $\boldsymbol{v}_i$ are constructed when they are needed by using a construction based on BCH codes and a randomly chosen irreducible polynomial of degree $\lceil \log M \rceil$. Thus each $\boldsymbol{v}_i$ can be made efficiently when it is needed, and based on this construction, has the 4-wise independence necessary to give the accuracy guarantees.

In [FKSV99] it was observed that this procedure to find F_2 can be adapted to find the L_2 distance between two interleaved, unaggregated streams, $\boldsymbol{a}$ and $\boldsymbol{b}$. We assume that the stream arrives as triples: $s_j = (\boldsymbol{a}_i, i, +1)$ if the element is from $\boldsymbol{a}$, and $(\boldsymbol{b}_i, i, -1)$ if the item is from stream $\boldsymbol{b}$. The goal is to find the square of the L_2 distance between $\boldsymbol{a}$ and $\boldsymbol{b}$, $\sum_i (\boldsymbol{a}_i - \boldsymbol{b}_i)^2$. Assuming the same set-up as before, we initialise Z to 0. When a triple $(\boldsymbol{a}_i, i, +1)$ arrives, we add $\boldsymbol{a}_i\boldsymbol{v}_i$ to Z. When a triple $(\boldsymbol{b}_i, i, -1)$ arrives, we subtract $\boldsymbol{b}_i\boldsymbol{v}_i$ from Z. After the whole stream has been processed, $Z = \sum(\boldsymbol{a}_i\boldsymbol{v}_i - \boldsymbol{b}_i\boldsymbol{v}_i) = \sum_i(\boldsymbol{a}_i - \boldsymbol{b}_i)\boldsymbol{v}_i$. So Z^2 is an estimator for $||\boldsymbol{a}-\boldsymbol{b}||_2^2$ for the same reasons: for each i, we find that $Z^2 = \sum_i(\boldsymbol{a}_i - \boldsymbol{b}_i)^2+$cross-terms. Again, the expectation of these cross-terms is zero, so the expectation of Z^2 is the L_2 difference of $\boldsymbol{a}$ and $\boldsymbol{b}$.

The procedure for L_2 also has a useful property that it can cope with an unaggregated stream containing multiple triples of the form $(\boldsymbol{a}_i, i, +1)$ for the same i. If k such triples occur anywhere in the stream: $(\boldsymbol{a}_{i,1}, i, +1), (\boldsymbol{a}_{i,2}, i, +1), \ldots (\boldsymbol{a}_{i,k}, i, +1)$, then because of linearity of addition, then this is treated identically as if it were a single triple $(\sum_{j=1}^{k} \boldsymbol{a}_{i,j},\ i, +1)$. This property will be useful for some of the algorithms developed in later chapters.

This streaming algorithm also translates to the sketch model: given a vector $\boldsymbol{a}$, the values of Z can be computed. The sketch of $\boldsymbol{a}$ is then these values of Z formed into a vector, $\boldsymbol{z}(\boldsymbol{a})$. So $\boldsymbol{z}(\boldsymbol{a})_i = \sum_j(\boldsymbol{a}_j - \boldsymbol{b}_j)\boldsymbol{v}_{i,j}$. This sketch vector has $O(1/\epsilon^2 \log 1/\delta)$ entries. Each entry of the vector requires $O(\log Mn)$ bits to represent it, since the maximum size of z is bounded by Mn. Two such sketches can be combined to find the sketch of the difference of the corresponding vectors since the sketches are composable in the sense of Definition 2.2.1: $\boldsymbol{z}(\boldsymbol{a} - \boldsymbol{b}) = (\boldsymbol{z}(\boldsymbol{a}) - \boldsymbol{z}(\boldsymbol{b}))$ Note that the random seed bits need to be shared, but since there are only $O(1/\epsilon^2 \log 1/\delta \log n)$ random bits, this can easily be arranged. The space used by the above streaming algorithm, and hence also the size of the sketch, is a vector of length $O(1/\epsilon^2 \log 1/\delta)$. It requires $O(\log n + \log M)$ bits to hold each entry.

Conversion to normed space

A subsequent paper [Ach01] takes essentially the same method and proves some stronger results. Specifically, the paper states that

Theorem 2.2.1 (Theorem 2 in [Ach01])

$$\frac{(1-\epsilon)||\boldsymbol{a}-\boldsymbol{b}||_2^2}{n} \leq \frac{||\boldsymbol{z}(\boldsymbol{a}) - \boldsymbol{z}(\boldsymbol{b})||_2^2}{k} \leq \frac{(1+\epsilon)||\boldsymbol{a}-\boldsymbol{b}||_2^2}{n}$$

with probability $1-\delta$, *where* $k = |\boldsymbol{z}| = O(1/\epsilon^2 \log 1/\delta)$.

In other words, the method of Alon Matias and Szegedy [AMS96] can be used as an approximate embedding into L_2 space of much smaller dimension. By construction of the sketch function $\boldsymbol{z}$, if $\boldsymbol{a}$ and $\boldsymbol{b}$ are vectors with integer entries, then so too will be their sketch $\boldsymbol{z}$. This removes the need for the averaging and median finding step, and allows these sketches to be used in existing algorithms that operate in Euclidean space.

2.2.3 L_1 Streaming Algorithm

The first algorithm to answer the problem of finding the L_1 distance in the streaming model was given by Feigenbaum, Kannan, Strauss and Viswanathan [FKSV99]. A similar approach is used to that described in Section 2.2.2. Again, it is assumed that items from vectors $\boldsymbol{a}$, $\boldsymbol{b}$ arrive in an arbitrarily interleaved order. A restriction is that it is assumed that for a given value of i, at most one triple $(\boldsymbol{a}_i, i, +1)$ and at most one triple $(\boldsymbol{b}_i, i, -1)$ occurs in the stream. So we can view this as the unordered but aggregated model.

The basic algorithm also makes use of a family of random variables, $r_{i,j}$ which take values $\{+1, -1\}$. A variable Z is initialised to 0. When the item $(\boldsymbol{a}_i, i, +1)$ arrives, add $\sum_{j=0}^{\boldsymbol{a}_i - 1} r_{i,j}$ to Z. When the item $(\boldsymbol{b}_i, i, -1)$ arrives, subtract $\sum_{j=0}^{\boldsymbol{b}_i - 1} r_{i,j}$ from Z. After the whole stream has been considered, the value of Z^2 is found. The value of Z^2 is then $\sum_{i=1}^{n} |\boldsymbol{a}_i - \boldsymbol{b}_i|$ plus cross terms of $r_{i,j}$'s. Assuming that these $r_{i,j}$'s are independent and uniform, then the expectation of the sum of these cross terms is zero. To give this method the required accuracy, this is repeated with different $r_{i,j}$'s. It is done $O(1/\epsilon^2)$ times to get an average value of Z, and then the median of $O(\log 1/\delta)$ such averages is found to get a result with the required accuracy.

The bulk of the technical results in [FKSV99] are to show how to construct families of range-summable pseudo-random variables that have 4-wise independence, using only a limited number of random bits. The 4-wise independence ensures that the expectation of the cross terms is zero, and that the variance is limited. They also need to be range-summable efficiently, so that $\sum_{j=0}^{k} r_{i,j}$ can be computed in time $O(\text{poly-log}\, k)$ rather than $O(k)$. With these results, it then follows that the L_1 distance can be computed in this streaming model using space $O(1/\epsilon^2 \log 1/\delta \log M \log n)$ and time $O(\log n \log\log n + 1/\epsilon^2 \log 1/\delta \log M)$ per item. In the same way as for L_2 distance, this streaming algorithm can be used to give a sketch algorithm by using the contents of the memory space as the sketch for the data. That is, the sketch consists of $O(1/\epsilon^2 \log 1/\delta)$ different values of z, plus a poly-logarithmic number of random bits. Again, each z is an integer requiring $\log M + \log n$ bits to represent it.

A later work of Fong and Strauss [FS00] takes a similar approach but works for any L_p distance where $p \leq 2$, by construction of a set of functions which allow the L_p difference to be found. We do not give details of this method since it shares the same limitation as [FKSV99]: it cannot be applied in the unaggregated model. We will instead make use of the methods described in Section 2.2.4 below.

2.2.4 Sketches using Stable Distributions

A limitation of the methods of [FKSV99] and [FS00] is that they insist that the stream is aggregated, so we see only one value for each $\boldsymbol{a}_i$. The approach described in [Ind00, CIKM02] is conceptually a little simpler, and works in the most general unordered, unaggregated model of streaming. It relies extensively on properties of stable distributions. There is not room for a thorough description of stable distributions here: see, for example, [Nol] for more details. We shall make reference to certain key properties of stable distributions, without proof.

Definition 2.2.2 *A* (strictly) stable distribution *is a statistical distribution, X with a parameter α in the range $(0, 2]$. It has the property that if $X_1, X_2, \ldots X_n$ are independently and identically distributed as X then $a_1X_1 + a_2X_2 + \ldots + a_nX_n$ is distributed as $||(a_1, a_2, \ldots, a_n)||_\alpha\, X$.*

Several well-known distributions are known to be stable. The Gaussian distribution is strictly stable with $\alpha = 2$; the Cauchy distribution is strictly stable with $\alpha = 1$; and the Lévy distribution is stable with $\alpha = \frac{1}{2}$. For all values of $\alpha \leq 2$, stable distributions can be simulated by using appropriate transformations from uniform distributions. Further details on stable distributions and computing values taken from stable distributions can be found in [Nol]. We focus firstly on L_1. These results were given by Indyk [Ind00].

Sketches for L_1 Distance

Given a vector $\boldsymbol{a}$, we define a sketch vector of $\boldsymbol{a}$, $sk^1(\boldsymbol{a})$. This vector is of length $k = O(1/\epsilon^2 \log 1/\delta)$. Similar to the L_2 case, we pick k random vectors $\boldsymbol{v}_1 \ldots \boldsymbol{v}_k$ where each $\boldsymbol{v}_i = (\boldsymbol{v}_{i,1}, \ldots, \boldsymbol{v}_{i,n})$ is made by drawing each component independently from a random distribution — in this case, the Cauchy distribution, given by $f(x) = \frac{1}{\pi(1+x^2)}$ for $-\infty < x < +\infty$. The L_1 sketch is defined by

$$sk^1(\boldsymbol{a})_i = \boldsymbol{a} \cdot \boldsymbol{v}_i$$

— that is, as the dot product of $\boldsymbol{a}$ with each of the k random vectors $\boldsymbol{v}_i$. For any d-length vector $\boldsymbol{a}$ let median($\boldsymbol{a}$) denote the median of the sequence $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_d$.

Theorem 2.2.2 (Theorem 1 of [Ind00]) *With probability* $1 - \delta$,

$$(1-\epsilon)||\boldsymbol{a} - \boldsymbol{b}||_1 \leq \text{median}(|sk^1(\boldsymbol{a}) - sk^1(\boldsymbol{b})|) \leq (1+\epsilon)||\boldsymbol{a} - \boldsymbol{b}||_1$$

Proof. The proof of this hinges on the fact that each element of $\boldsymbol{v}[i]$ is a real value, drawn from a stable distribution, and so $(\boldsymbol{v}_i \cdot \boldsymbol{a}) - (\boldsymbol{v}_i \cdot \boldsymbol{b}) = \boldsymbol{v}_i \cdot (\boldsymbol{a} - \boldsymbol{b})$ has the same distribution as $||\boldsymbol{a} - \boldsymbol{b}||_1 X$, where X has Cauchy distribution. Let $L = |\boldsymbol{v}_i(\boldsymbol{a} - \boldsymbol{b})|$. It was shown in [Ind00] that the median of L is equal to $||\boldsymbol{a} - \boldsymbol{b}||_1$. Thus, the repetition of the sampling ensures that the probability bound is met. By the linearity of the sketching function (it is just a dot product) it is computable in the unaggregated model. □

Sketches for L_p distances

We next turn to non-integral values of p for L_p distances. If we replace the variables from the Cauchy distribution with a strictly stable variable for which $\alpha = p$ then a similar result follows for the L_p distance between vectors. We define $sk^p(\boldsymbol{a})$ by $sk^p(\boldsymbol{a})_i = \boldsymbol{a} \cdot \boldsymbol{v}_i$, where here each $\boldsymbol{v}_i$ is made by drawing each entry independently from a stable distribution with parameter p. The sketch is still of length $O(1/\epsilon^2 \log 1/\delta)$.

Theorem 2.2.3 (Theorem 2 from [CIKM02]) *Given a sketch sk^p constructed as described above of length $O(1/\epsilon^2 \log 1/\delta)$ then for any $p \in (0, 2]$ there exists a scaling factor $B(p)$, such that for all vectors $\boldsymbol{a}, \boldsymbol{b}$, with probability $1 - \delta$ we have*

$$(1-\epsilon)||\boldsymbol{a} - \boldsymbol{b}||_p \leq B(p) \cdot \text{median}(|sk^p(\boldsymbol{a}) - sk^p(\boldsymbol{b})|) \leq (1+\epsilon)||\boldsymbol{a} - \boldsymbol{b}||_p$$

Proof. By the properties of stable distributions, if $X_0 \ldots X_n$ are distributed identically and independently as stable distributions with parameter p then $a_1 X_1 + \ldots a_n X_n$ is distributed as $||\boldsymbol{a}||_p X_0$. Then $\text{median}(|a_1 X_1 + \ldots a_n X_n|)$ is distributed as $\text{median}\,|(||\boldsymbol{a}||_p X_0)| = ||\boldsymbol{a}|_p\, \text{median}(|X_0|)$. Hence the expectation of each $|sk^p(\boldsymbol{a}) - sk^p(\boldsymbol{b})| = |sk^p(\boldsymbol{b} - \boldsymbol{b})|$, which follows from the linearity of the dot product function. This has expectation $||\boldsymbol{a} - \boldsymbol{b}||_p\, \text{median}(|X_0|)$, so this sets the scale factor $B(p)$ as $\text{median}(|X_0|)$, the median of absolute values from a stable distribution with parameter p. To get this to a $(1 \pm \epsilon)$ approximation with constant probability we require the median of $1/\epsilon^2$ independent repetitions, and to improve this to probability at least $1 - \delta$ we repeat this a further $\log 1/\delta$ times and take the median of all values. □

Observation 2.2.1 *$sk^p(\boldsymbol{a} + \boldsymbol{b}) = sk^p(\boldsymbol{a}) + sk^p(\boldsymbol{b})$, $sk^p(\boldsymbol{a} - \boldsymbol{b}) = sk^p(\boldsymbol{a}) - sk^p(\boldsymbol{b})$ and $sk^p(c\boldsymbol{a}) = c \cdot sk^p(\boldsymbol{a})$.*

These sketches are constructed in the same way as those in Section 2.2.1, as the dot product of the data with vectors of random variables, so it follows that these sketches are composable. This is a straightforward implication of the fact that sketches are constructed by a linear function. The

Reference	Metric	Vector Size	Streaming	Sketching	Composable
[JL84]	L_2	$O(1/\epsilon^2 \log 1/\delta)$	No	Yes	Yes
[AMS99]	L_2	$O(1/\epsilon^2 \log 1/\delta)$	Yes	Yes	Yes
[FKSV99]	L_1	$O(1/\epsilon^2 \log 1/\delta)$	Yes	Yes	No
[CIKM02]	$L_p, p \leq 2$	$O(1/\epsilon^2 \log 1/\delta)$	Yes	Yes	Yes

In each case, the Vector Size gives the number of entries in a sketch vector. Each entry needs $O(\log n + \log M)$ bits to represent it if M is an upper bound on the value of each coordinate . 'Streaming' indicates whether this result applies in the unordered streaming model; 'sketching' indicates whether it works in the sketch model, and 'composable' indicates whether the sketches are composable. These results apply for comparing a constant number of vectors.

Figure 2.1: Key features of the different methods described in Section 2.2.

observation of [Ind00] is that this sketching approach can be placed into the unordered streaming model if the random stable variables — the entries of the vectors $\boldsymbol{v}_i$ — can be made on the fly, so that whenever $\boldsymbol{v}_{i,j}$ is needed, the same value will be found. This can be done using standard pseudo-random generators as a function of i and j, using a limited number of truly random bits, as described in [Nis92]. Hence the necessary random variables can be created when needed, and so the sketch can be made in the streaming model.

Since $B(1) = B(2) = 1$, this means that these methods can be applied to approximating L_1 distance and L_2 distance exactly. For other values of p in the range $0 < p < 2$, the median of absolute values drawn from stable distributions can be found empirically, and used to give the scaling factor $B(p)$. In the case where $p = 2$, stable variables are taken from the Gaussian distribution, and this method is essentially the same as the Johnson-Lindenstrauss method, but using a median operation rather than an averaging operation. In general, the Johnson-Lindenstrauss method of using averaging is preferable, since it embeds into a normed space (Euclidean space), allowing algorithms to be performed directly in that space. We finally note that while the methods for L_1 in [FKSV99] require that the input stream consist of integers in a particular range, this technique can be applied to a stream of rational numbers, and the same results carry through.

2.2.5 Summary of Vector L_p Distance Algorithms

The key features of these four approaches are listed in Figure 2.1. Between them, we have efficient sketching and streaming algorithms for both L_1 and L_2 — in fact, in Section 2.2.4 we have shown that there are efficient sketching and streaming algorithms for all L_p distances, where $0 < p \leq 2$. These work in the most general, unaggregated and unordered model of streams. We observe that the space required by all algorithms depends on $1/\epsilon^2$ and $\log 1/\delta$. This is because most of the algorithms use the same techniques of finding $O(\log 1/\delta)$ averages and then taking the median of $O(1/\epsilon^2)$ averages, to ensure that the value found is a $1 \pm \epsilon$ approximation with probability $1 - \delta$. To compare m vectors with the same fixed probability of success, the size of the sketch should be multiplied by a factor of $O(\log m)$, as indicated in Section 2.2.1.

2.3 Set Spaces and Vector Distances

Having established these results for vector L_p distances, we now go on to describe a variety of distances based on sets, and show how these can be related to vector problems.

2.3.1 Symmetric Difference and Hamming Space

One of the most commonly used spaces in this thesis will be the (Vector) Hamming space, so we shall devote some attention to showing results on this space. Firstly, we shall describe several different scenarios which yield fundamentally the same distance measure.

Let A and B be sets, drawn from a universe $U = \{x_1, x_2 \dots x_n\}$, and consider the size of their symmetric difference $|A \Delta B|$, as described in Definition 1.2.3. Let a and b be binary strings of length n. As stated in Definition 1.4.2, the Hamming distance between these strings $h(a, b)$ is the number of places where they differ. It is straightforward to show how these are related: let χ be the characteristic function for the sets A and B, so χ maps a set onto a bit-string of length n. The i'th bit of $\chi(A)$ is set to 1 if x_i is a member of A, and 0 otherwise. Clearly, $h(\chi(A), \chi(B)) = |A \Delta B|$, since each x_i is included in $A \Delta B$ if and only if it occurs in exactly one of A and B. This contributes to $h(\chi(A), \chi(B))$ if and only if exactly one of $\chi(A)_i, \chi(B)_i$ is 1.

Definition 2.3.1 *We define $F(\boldsymbol{a}, \boldsymbol{b}, \odot)$ to be the function on binary strings $\boldsymbol{a}$ and $\boldsymbol{b}$ and binary functions $\odot : \{0,1\} \times \{0,1\} \to \{0,1\}$ such that:*

$$F(\boldsymbol{a}, \boldsymbol{b}, \odot) = \sum_{i=1}^{n} \boldsymbol{a}_i \odot \boldsymbol{b}_i$$

We observe that if $\odot = \mathrm{xor}$, the exclusive-or function,[1] then this function is also equivalent to the Hamming distance, $h(\boldsymbol{a}, \boldsymbol{b}) = ||\boldsymbol{a} - \boldsymbol{b}||_H = F(\boldsymbol{a}, \boldsymbol{b}, \mathrm{xor}) = F(\boldsymbol{a}, \boldsymbol{b}, \neq)$.

Approximating Hamming distance

The next observation allows us to relate the above discussion of vector distances to Hamming distance.

Observation 2.3.1 *If $\boldsymbol{a}, \boldsymbol{b}$ are bit-strings, then $||\boldsymbol{a} - \boldsymbol{b}||_H = ||\boldsymbol{a} - \boldsymbol{b}||_p^p$ for any $p > 0$.*

Proof. Since $||\boldsymbol{a} - \boldsymbol{b}||_p^p = \sum |\boldsymbol{a}_i - \boldsymbol{b}_i|^p$, we just need to consider each of the two possibilities for each $(\boldsymbol{a}_i, \boldsymbol{b}_i)$ pair. If $\boldsymbol{a}_i = \boldsymbol{b}_i$ then $|\boldsymbol{a}_i - \boldsymbol{b}_i|^p = 0$. If $\boldsymbol{a}_i \neq \boldsymbol{b}_i$ then $|\boldsymbol{a}_i - \boldsymbol{b}_i|^p = 1^p = 1$. Hence we add one to the sum for each pair of bits that differ, and only for these bits, and so this gives the Hamming distance between the bit-strings. $\square$

This observation means that any of the methods described earlier that work for approximating L_1 or L_2 distance can be used to answer equivalent problems on the Hamming distance, given an appropriate representation of the Hamming distance instance. This means that there are efficient ways of approximating the Hamming distance in the sketch and streaming models, by using the algorithms described in Section 2.2. We now state this formally:

Theorem 2.3.1 *A sketch for the Symmetric Difference (Hamming distance) between sets can be computed in the unordered, aggregated streaming model. Pairs of sketches can be used to make $1 \pm \epsilon$ approximations of the Hamming distance between their sequences, which succeed with probability $1 - \delta$. The sketch is a vector of dimension $O(\frac{1}{\epsilon^2} \log 1/\delta)$ and each entry is an integer in the range $[-n \dots n]$.*

[1]The exclusive-or function is defined as $\mathrm{xor} = \{((0,0),0), ((0,1),1), ((1,0),1), ((1,1),0)\}$

Proof. This proof draws on the results already discussed from [AMS99] and [FKSV99]. We assume each member of the sets is an integer i, so that $1 \leq i \leq n$. We begin with the sketch vector equal to zero, $\boldsymbol{z} = \boldsymbol{0}$. In the unordered, aggregated streaming model, members of the set A arrive in an arbitrary order. When a member of the set, x arrives, we can compute $\boldsymbol{v}_{i,x} \in \{-1, +1\}$ using a pseudo-random function of x as described in Section 2.2.3, by interpreting x as an item $(1, x, +1)$. We then compute $\boldsymbol{z}_i \leftarrow \boldsymbol{z}_i + \boldsymbol{v}_{i,x}$ for all i. The results of [FKSV99] as discussed in Section 2.2.3 show that the difference of two such sketches is a sketch of the difference. This allows the (ϵ, δ)-approximation of the L_1 distance between the vectors, which is exactly the Hamming distance in this case using Observation 2.3.1. The most (least) any entry of the sketch vector can be is $\pm n$, and so each entry requires $O(\log n)$ bits to represent it. The amount of random bits required is poly-logarithmic in n, as before. □

Contrarily, results from communication complexity show that *estimating* the Hamming distance is not possible in an efficient way. Pang and Gamal [PG86] showed that if two parties communicate to estimate the Hamming distance between bit-strings (where one person has one string and another has the other), then there must be $\Omega(n)$ bits of communication between them. This is enough to show that there cannot be any sketch algorithm to allow *estimation* of the Hamming distance using a sketch of size $o(n)$ bits — otherwise this would imply the existence of a communication scheme to estimate Hamming distance (one party sends the sketch of their bit-string to the other). Similarly, there cannot be a streaming algorithm to estimate Hamming distance that uses $o(n)$ space, since otherwise there would be a sketch algorithm (the sketch is the contents of the streaming data structure after one bit-string has streamed past). We go on to show that there are specific situations where approximation of Hamming distance is *not* possible.

Vector Hamming distance on unaggregated streams We will show that for different notions of Hamming distance it is not always possible to compute an approximation to the Hamming distance in sublinear space.

Lemma 2.3.1 *Let $\boldsymbol{a}$, $\boldsymbol{b}$ be vectors of length n represented as streams of tuples $(\boldsymbol{a}, i, +1)$ or $(\boldsymbol{b}, i, +1)$ in arbitrary interleaved order. The same tuple is allowed to appear multiple times in each stream. Any sketch to approximate the Zero-based Hamming distance (Definition 1.2.9) by $\hat{H}_0(\boldsymbol{a}, \boldsymbol{b})$ such that*

$$(1-\epsilon)\hat{H}_0(\boldsymbol{a}, \boldsymbol{b}) \leq H_0(\boldsymbol{a}, \boldsymbol{b}) \leq (1+\epsilon)\hat{H}_0(\boldsymbol{a}, \boldsymbol{b})$$

with constant probability, requires $\Omega(n)$ space.

Proof. We shall reduce this problem to one of communication complexity, which has known high cost. The problem known as "Index" is defined as follows: one person A holds an integer $1 \leq i \leq n$ while the other, B, holds a vector $x \in \{0,1\}^n$: the Index problem is for the first person, A to compute the i'th bit of x. In two rounds of communication, this can be computed exactly, using $\lceil \log n \rceil$ bits. However, in the one round randomized communication complexity model, $\Omega(n)$ bits are needed [KN97]. That is, any communication protocol to compute an estimate of Index requires a linear number of bits to be sent. We now show how any method in the streaming model to find the zero based Hamming distance can be transformed into a communication protocol for Index.

Let B create a representation of $\boldsymbol{a}$ in the stream by sending the pairs $(j, +1)$ for all positions j in $\boldsymbol{a}$ where $\boldsymbol{a}_j = 1$, and so the memory contents are set accordingly. Call this memory state M: we are concerned with the size of M. Note that we consider M to include any random bits used in the creation of the representation of $\boldsymbol{a}$. B then sends this memory state to A who can simulate the arrival of tuples $(\boldsymbol{a}, j, +1)$ and $(\boldsymbol{b}, j, +1)$ for all $1 \leq j \leq n$ except for $j = i$, the index of interest. If the streaming algorithm works as claimed then with probability $1 - \delta$ we have $\hat{H}_0(\boldsymbol{a}, \boldsymbol{b}) = 0 \iff \boldsymbol{a}_i = 0$ and $\hat{H}_0(\boldsymbol{a}, \boldsymbol{b}) \neq 0 \iff \boldsymbol{a}_i = 1$. So with constant probability we can solve the index problem, and so the information that was communicated, the memory contents M, must be of size at least $\Omega(n)$. □

What this lemma states is that it is not possible to have a sublinear sized sketch in the streaming model to compute the symmetric difference of sets if information is repeated (if the same index can occur more than once); on the other hand, we have already seen algorithms which use a sublinear amount of storage if no repeated information occurs. This will have implications for some of the algorithms we develop in later chapters.

Simple Embeddings into Hamming Distance

We show how methods that solve problems based on (Vector) Hamming distance can also solve problems for Hamming distance on arbitrary strings or sparse vectors, and for L_1 distance on vectors of bounded integers.

Hamming distance on non-binary alphabets A simple reduction suffices to embed the Hamming distance on constant sized non-binary alphabets into the Hamming distance on a binary alphabet. Given a string x drawn from an alphabet σ, we construct an arbitrary bijection $ord : \sigma \leftrightarrow \{1, \ldots, |\sigma|\}$. We then encode each character x_i as a bit-string $b(x_i)$ of length $|\sigma|$ where the j'th bit of the bit-string is set to one if and only if $ord(x_i) = j$. The binary encoding of x is the concatenation of these bit-strings in order from 1 to $|x|$. Because $||b(x_i) - b(y_i)||_H = 0$ if $x_i = y_i$, and is 2 if $x_i \neq y_i$, the Hamming distance of two such bit-strings is exactly twice the Hamming distance of the original strings. This approach allows solutions in the sketch and streaming models for problems utilising Hamming distance between strings from constant sized alphabets. When computing a sketch using the streaming methods outlined in Theorem 2.3.1, there is almost no overhead from this approach: we only have to encode non-zero bits (since the zero bits contribute nothing to the sketches), and so we just have to work out the contribution from the bit generated by each character.

Embedding L_1 distance into Hamming distance There are various technical solutions to embedding normed spaces into other normed spaces with certain probability guarantees. For general settings quite complicated mathematical solutions are needed, see [IM98, KOR98, Ind00]. For the kinds of L_1 distance we are interested in, we have an easier problem to solve, since our vectors $\boldsymbol{a}$ have the property that each $\boldsymbol{a}_i$ is an integer with $0 \leq \boldsymbol{a}_i \leq M$ for some known value M. This leads to a simple embedding of L_1 into Hamming distance: for each $\boldsymbol{a}_i$ create M bits $b(\boldsymbol{a}_i)$ such that $b(\boldsymbol{a}_i)_j = 1$ if $j \leq \boldsymbol{a}_i$, and 0 otherwise. This method, which has been described independently elsewhere, ensures that the Hamming distance of pairs of induced bit-strings is identical to the original L_1 distance. There is a blow-up in the dimensionality by a factor of M, however, in many cases this will not be important, since either M will be a small constant, or else this will be a preliminary step to an algorithm that does not depend on the dimensionality, so the cost will be independent of M. In fact, we will never make this embedding explicitly: the value of any bit can easily be derived when needed. By this encoding, every group of M bits can only take on $M + 1$ possible values, rather than 2^M.

Lower bound on Hamming distance

Lemma 2.3.2 *Let $\boldsymbol{a}$ and $\boldsymbol{b}$ be vectors with non-negative integer valued entries. Then $H_0(a, b) \leq ||\boldsymbol{a} - \boldsymbol{b}||_H \leq ||\boldsymbol{a} - \boldsymbol{b}||_1$*

Proof.
$H_0(a, b) \leq ||\boldsymbol{a} - \boldsymbol{b}||_H = \sum_i \boldsymbol{a}_i \neq \boldsymbol{b}_i = \sum_i (\boldsymbol{a}_i - \boldsymbol{b}_i) \neq 0 \leq \sum_i |\boldsymbol{a}_i - \boldsymbol{b}_i| = ||\boldsymbol{a} - \boldsymbol{b}||_1$. □

This lemma generalises Observation 2.3.1 for the L_1 distance.

Algorithm 2.3.1 *The Flajolet-Martin probabilistic counting algorithm*

```
initialise $bits[1,1] \dots bits[m, \log n] \leftarrow 0$
for all items $i$ do
  for $j = 1$ to $m$ do
    $bits[j, zeros(hash_j(i))] \leftarrow 1$
for $j = 1$ to $m$ do
  for $k = \log n$ downto $1$ do
    if $bits[j,k] = 0$ then
      $minzero \leftarrow k$
  $total \leftarrow total + minzero$
return($1.2928 \times 2^{total/m}$)
```

Approximating Hamming Distance for large, sparse vectors Later, we shall see examples where we construct vectors of size $O(n^2)$ and $O(2^n)$, which have only $O(n)$ non-zero entries. Directly applying the Johnson-Lindenstrauss lemma would suggest that we should construct an exponentially large vector of Gaussian variables, and convolve the two vectors. However, since the majority of the entries in the data vector are zero, these contribute nothing to the sum, and so instead we only need to focus on the non-zero entries, and their contribution to the convolution.

We consider a set representation of the vector: only the locations of the non-zero entries are listed. The size of this representation is polynomial in the size of the set. The sketch for the set can then be formed by adding appropriate 'random' variables, either by creating them on the fly and storing the values to be used again, or else by creating them based on pseudo-random functions of the value. This second option equates to the streaming approach described in Theorem 2.3.1 above. The length of these sketches is independent of the size of the vectors being represented, depending only on $\log 1/\delta$ and $1/\epsilon^2$. The number of bits required to represent each entry depends on the maximum value of any z_i, which is n, hence still only $O(\log n)$ bits are needed for each entry. Since the streaming algorithms assume any values not seen in the stream are zero, then if we pass only the locations of non-zero entries, then only an amount of work proportional to this number of entries needs to be done.

2.3.2 Set Union and Distinct Elements

As stated in Chapter 1, the union of two sets A and B, $A \cup B$ is the set $\{x | x \in A \vee x \in B\}$ (Definition 1.2.2). Approximating the size of the union of two sets can be achieved in the sketching and streaming models, by using results shown in [FM85] and developed in [AMS96]. It can also be achieved by using techniques based on the vector norm approximations that we have already discussed.

In the unaggregated streaming model, we might also want to count the number of distinct elements in the stream. This turns out to be almost identical to computing the size of the union of two sets in the same model.

Probabilistic Counting

This method to compute the number of distinct items seen in a stream of elements was first elaborated in [FM85]. Let n be the size of the universe from which the elements are being drawn. Without loss of generality, the authors assume that these are represented as integers in the range 1 to n. The aim is to compute how many distinct values are seen, that is, how many integers are represented in the stream. This is precisely the quantity F_0 of the stream as described in Section 2.2.2.

Theorem 2.3.2 *Due to Flajolet and Martin, (Theorem 2 of [FM83]).*

Using space $O(1/\epsilon^2 \log n \log 1/\delta)$, it is possible to compute an approximation of the number of distinct values in a unordered, unaggregated stream.

Proof. The procedure at the core of the estimation is as follows: for each element x which arrives in the stream for set A, compute a randomly chosen hash function of x, $hash(x)$, mapping onto $[1 \ldots n]$. In their analysis, Alon et al [AMS96] pick Linear Congruential Generators for this hash function, that is, functions of the form $hash(x) = ax + b \mod p \mod n$, with the parameters a and b picked uniformly at random from the range $[1 \ldots p]$, where p is a prime chosen in the range $n \leq p < 2n$. From this, compute the function $zeros(hash(x))$, which is the number of consecutive bits counting from the rightmost (least significant) which are zero. This has the property that, over all x, $\Pr[zeros(hash(x)) = i] = 2^{-(i+1)}$. We keep a bit vector $bits$, and every time $hash(x)$ is seen, then $bits[hash(x)] \leftarrow 1$. Following the processing of the sequence of values, we find $minzero$ as the smallest entry in the vector $bits$ that is zero. If this procedure is run on a stream of values, then $O(2^{minzero})$ is a good approximation for the number of values in the stream, using only $O(\log n)$ space to store $bits$, plus $O(\log n)$ working space. This procedure can then be made into an (ϵ, δ)-approximation by appropriate repetition and averaging for $m = O(1/\epsilon^2 \log 1/\delta)$ different hash functions (see [FM85, AMS96, BYJK$^+$02] for details), yielding a scheme with space requirements $O(1/\epsilon^2 \log n \log 1/\delta)$. Our output is two raised to the power of the average value of $minzero$, scaled by an appropriate scaling constant. This scaling factor is given in [FM85] as 1.2928. The algorithm implementing this is shown in Algorithm 2.3.1. □

An important property of this scheme is that if the same element occurs multiple times in the stream, it does not affect the result. This is because if x is seen twice, $hash(x)$ remains the same, and so seeing x again will not change $minzero$. So this procedure can be used to approximately count the number of distinct elements in an unordered unaggregated stream. This is especially important in database applications, where it is useful to maintain approximate information about database relations to allow query planning and optimisation, and even approximate query answering. We can cast this as a sketch algorithm for set union size, by using the bit vector as the sketch for the set that has been processed. It is straightforward to combine two such sketches to find the size of the union of two streams: for each $bits_A$ and $bits_B$ for streams A and B, we find $minzero$ for $(bits_A \vee bits_B)$. This can extend to multiple sets in the obvious way.

Hence, this approach can be used to compute the size of a set in the streaming model, with repeated information; and the size of the union of two (or more) sets can be approximated in the streaming and sketch models. Recent work has taken essentially the same approach, and using the same idea at the core (of a hash function where the probability of returning i is related to 2^{-i}) keeps a small *sample* of distinct elements. This is reported in [GT01] and experimental work on this in [Gib01]

Union Computation via Hamming Norms

An alternative approach to finding the union of two streams is given in [CDIM02]. We first observe that, for a single stream, the number of distinct items is given by the Hamming norm of a vector, defined as $||\boldsymbol{a}||_H = \sum_{i=1}^{n} \boldsymbol{a} \neq 0$ in Definition 1.2.7. By using a sufficiently small value of p, we can use the L_p norm in order to approximate the Hamming norm in the unaggregated streaming model.

Theorem 2.3.3 *The Hamming norm can be approximated by finding the L_p norm of a vector $\boldsymbol{a}$ for sufficiently small $p > 0$ provided we have a limit on the size of each entry in the vector.*

Proof. We provide an alternative mathematical definition for the Hamming norm. We want to find $|\{i | a_i \neq 0\}|$. Observe that $a_i^0 = 1$ if $a_i \neq 0$; we can define $a_i^0 = 0$ for $a_i = 0$. Thus, the Hamming norm of a vector $\boldsymbol{a}$ is given by $||\boldsymbol{a}||_H = \sum_i a_i^0$. This is similar to the definition of the L_p norm of a vector, which is defined as $(\sum_i |a_i|^p)^{1/p}$. Define the L_0 norm of $\boldsymbol{a}$ as $\sum_i a_i^0$. We show that L_0 (Hamming norm) of a vector can be well-approximated by $(L_p)^p$ if we take $p > 0$ small enough.

We consider $\sum_i |a_i|^p = (L_p)^p$ for a small value of p ($p > 0$). If, for all i, we have that $|a_i| \leq U$ for some upper bound U, then

$$\sum_i a_i^0 \leq \sum_i |a_i|^p \leq \sum_i U^p a_i^0 = U^p \sum_i a_i^0 \leq (1+\epsilon) \sum_i a_i^0$$

if we set $p \leq \log(1+\epsilon)/\log U \approx \epsilon/\log U$. □

Hence, if we can arrange to be able to find the quantity $\sum_i |a_i|^p$, then we can approximate the Hamming norm up to a $(1+\epsilon)$ factor.

Corollary 2.3.1 *We can make sublinear sized sketches in the unaggregated streaming model to allow the approximation of the vector Hamming distance.*

This corollary follows based on a few observations. Firstly, these sketches can be computed in the unaggregated streaming model since we can use sketches for the L_p norm which are computable in that model. Secondly, since these sketches are generated by a linear function (the dot product), it is easy to combine two sketches of vectors $\boldsymbol{a}, \boldsymbol{b}$ to get a sketch of $\boldsymbol{a} - \boldsymbol{b}$ whose Hamming norm is the vector Hamming distance of the original vectors. The space required is the same as in Theorem 2.2.3, that is, $O(1/\epsilon^2 \log 1/\delta)$. We can also find the union of two streams representing $\boldsymbol{a}$ and $\boldsymbol{b}$ by noting that the quantity we seek is $||\boldsymbol{a} + \boldsymbol{b}||_H$. This can be achieved by sketching $\boldsymbol{a}$ and $\boldsymbol{b}$ separately, generating $sk(\boldsymbol{a}, r)$ and $sk(\boldsymbol{b}, r)$, and then creating $sk(\boldsymbol{a} + \boldsymbol{b}, r) = sk(\boldsymbol{a}, r) + sk(\boldsymbol{b}, r)$. This follows, since these sketches are composable, by Observation 2.2.1.

Note that this approach to Hamming norms and related quantities is rather more general than probabilistic counting, since this still functions even if entries in $\boldsymbol{a}$ and $\boldsymbol{b}$ are negative. It is undefined and unclear what to do in probabilistic counting when entries are negative. However, here it is straightforward, and well defined. Since we are using L_p norms, if we find the norm of a vector with negative entries using a sketch, the result is known, this just counts one towards the total of non-zero entries. This behaviour is useful in many situations where these sketches are computed in a distributed fashion, or when negative values are a reasonable part of the input.

2.3.3 Set Intersection Size

If $\boldsymbol{a}, \boldsymbol{b}$ are binary strings, then their intersection size $i(\boldsymbol{a}, \boldsymbol{b}) = F(\boldsymbol{a}, \boldsymbol{b}, \times)$ gives the dot product of $\boldsymbol{a}$ and $\boldsymbol{b}$ (see Definition 2.3.1). These two statements are essentially identical, since $|A \cap B| = F(\chi(A), \chi(B), \times)$.

Hamming distance and set intersection size are closely related. Simple consideration of Venn diagrams shows that $|A| + |B| - 2|A \cap B| = |A \Delta B|$. Equivalently, for bit-strings, $|\boldsymbol{a}| + |\boldsymbol{b}| - 2i(\boldsymbol{a}, \boldsymbol{b}) = ||\boldsymbol{a} - \boldsymbol{b}||_H$, where $|\boldsymbol{a}|$ is the number of one bits, or 'weight' of the bit-string $\boldsymbol{a}$. However, this equality does not mean that we can approximate the intersection size by first approximating the Hamming distance. Informally, although we can find $|\boldsymbol{a}|$, $|\boldsymbol{b}|$ and $||\boldsymbol{a} - \boldsymbol{b}||_H$ within a factor of $1 \pm \epsilon$ each, $i(\boldsymbol{a}, \boldsymbol{b})$ may be small in comparison to these and hence directly applying the above relation would not give an (ϵ, δ) approximation to i.

There is a more direct embedding of Hamming distance into intersection space. Let $\bar{\boldsymbol{a}}$ be the bit-vector formed by taking the complement of each bit in $\boldsymbol{a}$. As noted before, the concatenation of two vectors $\boldsymbol{a}, \boldsymbol{b}$ is denoted $\boldsymbol{a}||\boldsymbol{b}$. Then $i(\boldsymbol{a}||\bar{\boldsymbol{a}}, \bar{\boldsymbol{b}}||\boldsymbol{b}) = i(\boldsymbol{a}, \bar{\boldsymbol{b}}) + i(\bar{\boldsymbol{a}}, \boldsymbol{b}) = ||\boldsymbol{a} - \boldsymbol{b}||_H$, so in this case an approximation of i would allow approximation of h (since we are *adding* these quantities). However, there is no similar reduction in the opposite direction, expressing the intersection size in terms of a summation of Hamming distance or other quantities. This is unfortunate, since although there are efficient approximations for Hamming distance, there are no known approximations for the intersection size. In fact, we go on to show that no such approximation can exist.

Hardness of Approximating Intersection Size Certainly, *estimating* the intersection size is hard. This follows immediately from the fact that estimating Hamming distance is hard. If in the sketch model we could estimate the intersection size with probability δ, then we could also estimate Hamming distance with the same probability, by using the relations above (since $|\boldsymbol{a}|$ can easily be included in a sketch using $\log n$ bits). To show that approximating the intersection size is hard, we consider the related problem of disjointness:

$$\begin{aligned} \operatorname{disj}(\boldsymbol{a},\boldsymbol{b}) = 1 \quad &\iff \quad i(\boldsymbol{a},\boldsymbol{b}) = 0 \\ \operatorname{disj}(\boldsymbol{a},\boldsymbol{b}) = 0 \quad &\iff \quad i(\boldsymbol{a},\boldsymbol{b}) \neq 0 \end{aligned}$$

It has been shown that in a communication complexity scenario, estimating the disjointness function requires $\Omega(n)$ bits of communication, in the main theorems of [KS92, Raz92].

Theorem 2.3.4 (from [Raz92], [KS92]) *Let person A hold $\boldsymbol{a}$ and person B hold $\boldsymbol{b}$. Any scheme which allows A and B to collaboratively compute $\operatorname{disj}(\boldsymbol{a},\boldsymbol{b})$ with probability $1-\delta$ requires $\Omega(n)$ bits of communication for every $\delta < \frac{1}{2}$.*

Corollary 2.3.2 *An approximation scheme for intersection size would imply a way to estimate disjointness.*

Proof. We just have to consider two cases. Firstly, when $i(\boldsymbol{a},\boldsymbol{b}) = 0$, then any approximation would have to return a result of 0, whatever the approximation parameters a, b (see Definition 2.1.2) with probability δ. Similarly, if $i(\boldsymbol{a},\boldsymbol{b}) \neq 0$ then any approximation would have to return a non-zero result with probability δ. Hence if an approximation scheme for i existed then we could derive an estimate for disj by returning 0 if $\hat{i} \neq 0$ and returning 1 if $\hat{i} = 0$. □

Therefore, by contradiction, there can be no streaming or sketching algorithm for approximating intersection size since otherwise there would exist a communication protocol for estimating disjointness. We comment that although this rules out the existence of any constant factor approximation scheme for intersection size, there is still the possibility of a scheme that returns an answer correct up to a constant factor plus an additional constant. That is, find $\hat{i}$ such that $i(\boldsymbol{a},\boldsymbol{b}) \leq \hat{i}(\boldsymbol{a},\boldsymbol{b}) \leq \epsilon i(\boldsymbol{a},\boldsymbol{b}) + c$ for constants c and ϵ. Next we pursue an alternative weaker kind of approximation.

Rough Sketches for Intersection Size Although we cannot make sketches for the intersection size that meet the specifications of Definition 2.1.3, it is still possible to find a sort of approximation that may be "good enough". Informally, we will call this a *rough sketch*. This is an additive approximation of the intersection size. That is, an approximation $\hat{i}$ such that $|i(\boldsymbol{a},\boldsymbol{b}) - \hat{i}(\boldsymbol{a},\boldsymbol{b})| \leq \epsilon n$ (rather than $\epsilon i(\boldsymbol{a},\boldsymbol{b})$). We describe a simple communication scheme that achieves such an approximation, adapted from Section 5.5 of [KN97]. Suppose person A holds a bit-string $\boldsymbol{a}$, and person B holds $\boldsymbol{b}$. Then A selects k locations from $\{1 \ldots n\}$ uniformly at random, with replacement, and sends this subset followed by the values of $\boldsymbol{a}$ at each of these positions. B then estimates i by

$$\hat{i} = \frac{n}{k} \sum_{i=1}^{k} \boldsymbol{a}_{S_i} \boldsymbol{b}_{S_i}$$

Lemma 2.3.3 *$\hat{i}$ is an ϵ-additive approximation for i with probability δ for k sufficiently large in $\Theta(1/\epsilon^2 \log 1/\delta)$.*

Proof. We pick k locations uniformly at random from the bitstrings with replacement. For any chosen bit, the probability that it is in the intersection is then $i(\boldsymbol{a},\boldsymbol{b})/n$. We compute the intersection of the two sets of chosen bits and scale by n/k, and by linearity of expectation, the expected size of this quantity is $n/k(k(i(\boldsymbol{a},\boldsymbol{b})/n)) = i(\boldsymbol{a},\boldsymbol{b}) = i$. Each test of a chosen bit is a where each outcome is independently and identically distributed so we can apply the Hoeffding inequality to the sum of k such trials, as described in Chapter 4 of [MR95]. The probability that the difference between $\hat{i}$ and i is more than ϵn is given by

$$\Pr(|\hat{i} - i| \geq \epsilon n) = \Pr(k/n|\hat{i} - i| \geq \epsilon k) \leq 2e^{-2k\epsilon^2} = \delta$$

Rearranging this shows that $k = -\frac{1}{2\epsilon^2} \ln \delta/2$ and so $k = \Theta(1/\epsilon^2 \log 1/\delta)$. □

So to allow this to take place in a communication setting, A just needs to send the k locations, plus the k bits at those locations, to B, at a cost of $O(1/\epsilon^2 \log 1/\delta \log n)$ bits of communication. We remark again on the close relation between communication protocols and sketching algorithms, since any lower bound in a communication paradigm is a lower bound on the size of a sketch — because a sketch can be viewed as a message being communicated.

Rough Sketching for Smaller Sets Work by Broder described in [Bro98] and with further technical details in [BCFM98] shows a way to make sketches for intersection size. These sketches are used in retrieving documents in the AltaVista search engine. The approach is based on choosing hash functions and (pseudo)random permutations of the universe from which sets are drawn. For a given set, a permutation is applied, and then each element is hashed to an integer. These hash values are taken modulo some integer m, and those that are congruent to 0 are selected for the sketch. The additive approximation of the intersection size of two sets is then found by taking the intersection size of their representative sketches, and scaling appropriately. The purpose of this approach is to deal with cases where the size of any individual set is very small compared to the size of the universe from which the sets are drawn. In this case the method of Lemma 2.3.3 would mainly sample bits that were 0, giving a poor approximation. Repetition of this process (using different permutations and hash functions) can improve the quality of the approximation. However, the size of this representation varies linearly with the size of the sets: the expected size of the representation of A is $O(|A|/m)$.

Variations of Intersection

We consider an alternative measure of set similarity, the Set Resemblance measure, an extension from sets to vectors, and the related set measure of Set Difference.

Set Resemblance Closely related to Set Intersection size is the Set Resemblance measure studied by Broder [Bro98]. This is also known as the Jaccard coefficient, as used in statistics and information retrieval. The resemblance of two sets A and B is $r(A,B) = \frac{|A \cap B|}{|A \cup B|}$ (it is assumed that A and B are not both the empty set). Since $r(A,B) = 0$ if and only if $A \cap B = \emptyset$, approximating this quantity is hard for the same reasons as approximating the intersection size (otherwise, we could solve the disjointness problem). Let A and B be both drawn from a universe U of size n. Suppose that P is a permutation on U (that is, it maps U bijectively to $\{1 \ldots n\}$) chosen uniformly at random from all such permutations. We can apply P to A to get $P(A) \subseteq \{1 \ldots n\}$. We can create a rough sketch for A by considering many different random permutations P_i so that the sketch of A is given by:

$$sk(A,P) = (\min(P_1(A)), \min(P_2(A)), \ldots \min(P_k(A)))$$

The approximation of $\frac{|A \cap B|}{|A \cup B|}$ is $1 - (||sk(A,P) - sk(B,P)||_H)/k$ where $||\cdot||_H$ is the Hamming norm as usual. This is because for each coordinate $\Pr(\min(P_i(A)) = \min(P_i(B))) = \frac{|A \cap B|}{|A \cup B|}$. So by summing these, the expectation of $1 - ||sk(A,P), sk(B,P)||_H/k$ is the resemblance of the sets A and B. The advantage of this approach is that, unlike that described above, it is independent of the size of the sets A and B, working equally well whether the sets are large or small. The size of this sketch is k elements, each of which can be represented using $\log n$ bits. This method of approximating the Jaccard coefficient was defined in [Bro98] and refined in [BCFM98]. Attention is given to how to choose the permutations P: storing a permutation in full would consume too much space. Instead, permutations are chosen at random from a smaller class of "Min-Wise Permutations", which can be represented in small space, although we do not discuss this further here. It is also remarked in [Bro98] that $1 - \frac{|A \cap B|}{|A \cup B|}$ is a metric, although we do not make use of this fact.

Dot products Given two vectors $\boldsymbol{a}, \boldsymbol{b}$, we have already stated that their dot product is $\sum_i \boldsymbol{a}_i \boldsymbol{b}_i$. In the case where the entries of the vectors are restricted to being 0 or 1, then as we observed, the dot product is isomorphic to the set intersection measure. It therefore follows that approximating the dot product of two vectors is at least as hard as set intersection, that is, it requires $\Omega(n)$ bits of communication in a communication complexity model. For non-negative vectors, such as those we shall be considering in later chapters, there are techniques to select from a collection of vectors those that have a large dot-product with a query vector which may improve on the direct evaluation, depending on the nature of the input data [CL99].

Set Difference We show that there can be no algorithm in the sketch model to allow the approximation of the size of the set difference, $|A \backslash B|$. This measure is quite similar to intersection, since $A \backslash B = A \cap \bar{B}$, hence we would not expect there to be an approximation scheme. Suppose that it were possible to make sketches for this measure, so that there are sketches of A and B, $sk(A, r)$ and $sk(B, r)$. Then for some function f, we would have $|f(sk(A, r), sk(B, r)) - |A \backslash B|| \leq \epsilon |A \backslash B|$ with probability $1 - \delta$. If such a sketch has been made for some set A, then this sketch and the random bits used to create it could be communicated to B, who could then compute $sk(\bar{B}, r)$ correspondingly, where $\bar{B}$ is the complement of the set B with respect to the universe, as usual. We then consider the approximation of $|A \backslash \bar{B}|$. The result is 0 with probability at least $1 - \delta$ if A does not intersect B, and non-zero with the same probability if A and B are not disjoint. In other words, we can reduce the problem to that of Disjointness, and show that the combined size of the sketch and all the information used to create it (ie the total amount of memory used) must be $\Omega(n)$. This is due to the fact that computing the disjointness function has been shown to have linear communication complexity (see Section 2.3.3).

2.3.4 Approximating Set Measures

The results of the previous section allow us to bound the space cost of approximating different parts of the Venn diagram for two sets A and B. These are reported in Figure 2.2. We consider two sets, A and B drawn from a universe of size n. The table gives space bounds in bits for the approximation (in the streaming model) of various quantities from the Venn diagram. We consider two fundamental variations of the streaming model: when each element of a set is guaranteed to be presented exactly once (equivalently, when we are in the sketch model) — this gives the results in the 'aggregated' column. In the 'unaggregated' column, we treat this as a purely streaming problem where the same element may be seen several times over within the stream. Since linear space requirements will generally be regarded as impractical or uninteresting, the divide between linear and sub-linear (poly-logarithmic space requirements) is interesting to note: intersection size and set difference size are impractical in both models, whereas set size and union size are always practical. Our most frequently used measure, symmetric difference, straddles the boundary, since it is practical in the sketch and aggregated streaming model, but impractical in the unaggregated streaming model when information can be repeated.

Quantity	Aggregated	Unaggregated
$\|A\|$	$O(\log n)$	$O(1/\epsilon^2 \log n \log 1/\delta)$
$\|A\Delta B\|$	$O(1/\epsilon^2 \log n \log 1/\delta)$	$\Omega(n)$
$\|A \cup B\|$	$O(1/\epsilon^2 \log n \log 1/\delta)$	$O(1/\epsilon^2 \log n \log 1/\delta)$
$\|A \cap B\|$	$\Omega(n)$	$\Omega(n)$
$\|A \backslash B\|$	$\Omega(n)$	$\Omega(n)$

Figure 2.2: Approximability of set quantities

2.4 Geometric Problems

We first saw these Geometric Problems in Section 1.5.3. We now describe some solutions to these problems under some of the vector distances we have described. We begin by looking at Approximate Nearest and Furthest Neighbors. The solutions we shall see here will be for the Hamming distance, but in later chapters we shall solve approximate nearest neighbors in other distance spaces. A detailed consideration of Vector Nearest Neighbors methods is given in [Tsa00]. We then go on to look at a general solution for k-centers clustering.

We first describe in brief recent methods for solving the problems of Approximate Nearest Neighbors (Definition 1.5.2). These methods actually solve a slightly simpler problem, which is given a distance r, and a query point q, to return (with high probability) some point p whose distance is at most r from q or report that there is no point whose distance is at most $(1+\epsilon)r$ (in between, either answer is acceptable). If we can solve this problem well enough for all possible values of r then we can perform a search on this structure to find an ϵ-approximate nearest or furthest neighbor of q: we begin testing whether q is in the database or not. If not, then we can do a binary chop to find the minimum value of l for which there is a distance $(1+\epsilon)^l$ with a point returned by this query. Note that we will be dealing with discrete metric spaces where the distance between two distinct points is an integer between 1 and n for some n, hence there are a limited number of distances that we need to check.

2.4.1 Locality Sensitive Hash Functions

The method we describe is originally from [IM98] and refined in [GIM99] where it is shown to work empirically. It also solves the problem of determining whether there is an Approximate Neighbor at distance r. Their method is defined to use 'Locality Sensitive Hash Functions', whose collision probability is related to the distance between objects. Related ideas are used to give alternative solutions by Kushilevitz, Ostrovsky and Rabani [KOR98].

Definition 2.4.1 (From [IM98]) *A family H of* locality sensitive hash functions *h, with parameters $r, \epsilon, p_1 > p_2$ mapping points p, q from the relevant metric space onto some discrete range satisfies the following properties for any p and q:*

If $d(p,q) \leq r$ then $\Pr_H[h(p) = h(q)] \geq p_1$.

If $d(p,q) \geq r(1+\epsilon)$ then $\Pr_H[h(p) = h(q)] \leq p_2$.

Since $p_1 > p_2$ this gives a bias towards nearest neighbors. That is, the functions h are more likely to give the same value if p and q are close, with the probability taken over all choices of h from the family H. On the other hand, if we exchange the roles of p_1 and p_2, so that if p and q have a high distance then the function is more likely to give the same value, then we can use the test to find furthest neighbors and so can be used to solve Approximate Furthest Neighbors. However, we need to amplify these probabilities to give a bounded probability of success. The method then works as follows: first, we fix l sets of k locality sensitive hash functions $h_{i,j}$, where the functions are drawn independently and uniformly at random from the family H. Then for each of the m points in the database, we repeat l times: compute k the different locality sensitive hash functions for this point p, giving $h_{i,1} \dots h_{i,k}$. We concatenate the output of these hash functions to give a single hash of the point, $h_i(p) = h_{i,1}, h_{i,2} \dots h_{i,k}$. Then use a second (linear) hash function to compress the range of this function into a hash table as described in Section 2.1.3: select a random prime $\hat{p}$ and compute $hash(h_i(p))$ using the function $hash$ defined there, setting the collision probability $\delta = 1/m$. We do this l times, giving $hash(h_1(p)) \dots hash(h_l(p))$. To look up a query point q, we perform the same transformation: hash the point under the hash function getting $hash(h_1(q)) \dots hash(h_l(q))$. We then examine the contents of the l different buckets $hash(h_i(q))$. We extract all the points found in these buckets (stopping in the unlikely case that we exceed $4l$ distinct points), extracting a set of points P such that for each $p \in P$ there is some i such that $hash(h_i(p)) = hash(h_i(q))$. Amongst this set of points P we search for the (exact) nearest or furthest neighbor of q and test whether this is within a distance of r from q, outputting it if it is, and outputting a negative response if there is no point within a distance of $r(1+\epsilon)$ from q. For the proof, we shall assume that we are dealing with Nearest Neighbors; the argument and technical details are almost identical for Furthest Neighbors.

Theorem 2.4.1 (From Theorem 4 of [IM98] and Theorem 1 of [GIM99]) *This method finds a $(1+\epsilon)$ Approximate Nearest Neighbor of a query point q with constant probability. It requires $O(n^{\frac{\log p_1}{\log p_2}})$ hash function evaluations per query.*

Proof. Suppose there is a (close) p^* such that $d(p^*, q) \leq r$. Let us set $k = \log_{1/p_2} m$. The probability for any given i that p^* hashes to the same value as q (so $hash(h_i(p^*)) = hash(h_i(q))$) is given by the probability that each of the LSH functions gives the same answer on p and q, which is

$$\Pr[hash(h_i(p^*)) = hash(h_i(q))] \leq p_1^k = p_1^{\log_{1/p_2} m} = m^{-\frac{\log 1/p_1}{\log 1/p_2}} = m^{-\rho}$$

for $\rho = \frac{\log 1/p_1}{\log 1/p_2}$. If we set $l = m^\rho$ then the probability that this works for at least one value i is $1 - (1 - m^{-\rho})^{m^\rho}$ which is at least $1 - 1/e$.

We consider the probability that some (far) point p' for which $d(p', q) \geq (1+\epsilon)r$ hashes to the same value as q under all k LSH functions or there is a collision in the bucketing hash function ($hash$): this is $p_2^k = 1/m + 1/m = 2/m$. So the expectation of the number of such p' that create hash clashes is at most 2 by linearity of expectation. Over all l, the number of buckets we expect to see filled with bad points is then at most $2l$. By the Markov inequality, the probability that this is more than $4l$ is less than $1/2$.

The probability that we do find a good point, and we do not find too many bad points, is at least $1 - ((1/2) + (1 - (1 - 1/e))) = 1/2 - 1/e$ which is a constant. □

We can increase this constant probability of success to some arbitrary probability $1 - \delta$ by repeating the procedure (with different hash functions) $O(\log 1/\delta)$ times. The query running time is that needed for $l = n^\rho$ evaluations of a hash function.

Application to Approximate Nearest Neighbors under the Hamming Metric

A LSH family for the Hamming metric is just the set of $h_i(\boldsymbol{a}) = \boldsymbol{a}_i$, that is, the functions which pick the i'th bit of a bit string. From this, we can easily find that $p_1 = 1 - r/n$ and $p_2 = 1 - r(1+\epsilon)/n$ and so $p_1 > p_2$. Hence to generate a function h_i, we choose the k locations to sample, from $[1 \dots n]$ uniformly at random (with replacement). Recall that the running time depends on m^ρ evaluations of a group of hash functions, each of which takes time at most $O(n)$. We can simplify the expression of ρ and show that $m^\rho < m^{1/(1+\epsilon)}$ (see [GIM99] for more details). Thus, the query time is $O(nm^{1/1+\epsilon})$. For nearest neighbors, there is the additional $O(\log n)$ factor to find the smallest value of r at which there is a neighbor, giving an overall ANN query time of $O(n \log n\, m^{1/(1+\epsilon)})$. In [GIM99] it is argued that for many practical applications a single test of this nature is suitable since empirically all nearest neighbors occur at about the same distance.

Application to Furthest Neighbors under Set Resemblance

Let r be the Jaccard coefficient of set resemblance = $\frac{|A\cap B|}{|A\cup B|}$. We can define a LSH family H for this measure based on the sketch method described above. The properties that we want for furthest neighbors are:

If $d(p,q) \leq r$ then $\Pr_H[h(p) = h(q)] \leq p_2$.

If $d(p,q) \geq r(1+\epsilon)$ then $\Pr_H[h(p) = h(q)] \geq p_1$.

We pick a permutation of the universe uniformly at random, and pick the minimum element of a set under this permutation, so $h_P(A) = \min P(A)$. For a fixed r, we have that if $d(p,q) \leq r$ then $\frac{|A\cap B|}{|A\cup B|} \leq r$. Then $\Pr[\text{First item picked is in both } A \text{ and } B] = \frac{|A\cap B|}{|A\cup B|}$. So $p_2 \leq r$ and $p_1 \geq r(1+\epsilon)$. Recall throughout that $0 \leq r \leq 1$.

$$\rho = \frac{\ln 1/p_1}{\ln 1/p_2} = \frac{\ln r(1+\epsilon)}{\ln r} = 1 + \frac{\ln(1+\epsilon)}{\ln r} \leq 1 - \frac{2\epsilon}{3\ln 1/r}$$

The last step assumes that $\epsilon \leq 1$. This means that we have to compute $O(m^\rho) = m^{1-\frac{2\epsilon}{3\ln 1/r}}$ different hash tables. Each look up in a hash table requires computing $\log_{1/p_2} m = \ln m/(\ln 1/r)$ hash values. To find an approximate furthest neighbor, we may try $\log n$ such tests doing a binary search on the distance. The total cost is then $O(m^{1-\frac{2\epsilon}{3\ln 1/r}} \log n \ln m/\ln 1/r)$ locality sensitive hash function evaluations, compared to the exact algorithm whose cost is $O(mn)$. Note that the cost of computing these hash functions based on permutations is not constant, and that as r approaches 1 or 0 this can become too high a cost. We shall have to address these issues when we come to applying this method later.

2.4.2 Approximate Furthest Neighbors for Euclidean Distance

A relatively simple scheme is given in [GIV01] for finding $\sqrt{2}$-approximate furthest neighbors under the Euclidean distance. At a high level, the algorithm is as follows:

Let S be the set of points of interest. Let B be a smallest hypersphere such that $\forall s \in S : s \in B$. B has radius r and is centred at some point O. We find a subset of S, R, such that every point of R is a distance of r from O and the convex hull of R contains O. R is chosen to be of size at most $\ell + 2$ points for ℓ dimensional space. Approximate furthest neighbors queries can then be answered using R: given a point query q, we compute a hyperplane that passes through O and is normal to the line joining q and O. The response to the query is any point from R that is on the other side of this hyperplane from q. A geometric argument shows that the true furthest neighbor of q cannot be more than $\sqrt{2}$ times further than the answer found this way. The time to perform this is that to examine the $O(\ell)$ elements of R, each of which is a vector of dimension ℓ.

The actual solution requires more work, to construct a good approximation to R and O in a reasonable amount of time. In [GIV01] it is shown that a set of size $O(\ell \log |S|)$ can be found to use for R. The scheme has a per-query cost of $O(\ell^2 \log |S|)$ and returns a neighbor that is at most a $\sqrt{2} + o(1)$ approximation to the further neighbor. The preprocessing cost to find R is $O(\ell^3 |S| \operatorname{poly-log} \ell |S|)$. Since this solution works in Euclidean space, we can reduce the dimensionality of the space using the Johnson-Lindenstrauss lemma (Section 2.2.1) to $O(\log |S|)$ while still guaranteeing that the expansion of any pair of distances is bounded with high probability. In this randomised setting, the algorithm gives $O(\log^3 |S|)$ query time (plus time to process the query point) and with arbitrary constant probability returns a $\sqrt{2}$-approximate furthest neighbor.

Algorithm 2.4.1 *Gonzalez's Algorithm for clustering in a metric space*

```
Initialise centers to an arbitrary point
j = 1
repeat
  Find $x$ such that $\min_{i \le j} d(x, center_i)$ is maximised
  $j = j + 1$
  Set $center_j = x$
until $j = k$
```

2.4.3 Clustering for k-centers

We return to the problem of clustering by finding k-centers as described in Section 1.5.5. A simple approximation algorithm due to Gonzalez [Gon85] allows us to find a 2-approximate solution for any distance measure that is a metric. This is a clustering $approx$ such that $\forall clusters : \mathrm{spread}(approx) \le 2\,\mathrm{spread}(clusters)$. The approximation algorithm picks out k centers from the data to define the clusters. A center is one of the data points, and the clustering induced from the k centers is to associate each other data point with the center that it is closest to. A greedy approach picks out the centers: an arbitrary data point is selected to initialise the centers. Then the data point that is furthest from all of the centers is added to the set of centers, until there are k centers. The induced clustering is then claimed to be a 2-approximation. The cost of the algorithm, given in pseudo-code as Algorithm 2.4.1, is $O(km)$ comparisons.

Theorem 2.4.2 (Theorem 2.2 of [Gon85]) *Algorithm 2.4.1 gives a 2-approximation to the optimal clustering.*

Proof. We consider the effect of running an additional iteration of the algorithm to find a $(k+1)$th center, x_{k+1}. The distance of this point from the clusters is at most its distance from the other k-centers, call this d. All of the k-centers are therefore separated by at least d, since in each iteration we pick out points that maximise their distance from the other centers. Because there are $k+1$ points where any pair is separated by at least d, it follows that the spread of an optimal clustering is at least d. Also, since the $(k+1)$th center maximises d amongst all non-centers, it follows that the distance from any point to its closest center is at most d. So, the distance between any two points in a cluster is at most $2d$, by the triangle inequality. The size of each cluster is less than $2d$, and the minimum size at least d, hence this is a 2-approximation. The running time of the algorithm is $O(kmn)$, that is there are $O(km)$ comparisons, each of which takes $O(n)$ time. □

This straightforward approximation is convenient, since it allows us to modify it to work using approximate distances rather than exact distances.

Theorem 2.4.3 *If, instead of the exact distance function, we use an approximation that is at most c times the actual value in Algorithm 2.4.1, then the algorithm gives a $2c$-approximation to the optimal clustering.*

Proof. We repeat the above proof, but adjust for the fact that the distances found are approximations of the true distance. We assume that the approximate distance, $\hat{d}$ behaves so that $d(x,y) \le \hat{d}(x,y) \le c \cdot d(x,y)$. As before, we run the algorithm, and consider the $(k+1)$th center. Then there are $k+1$ points where any pair is separated by at least $\hat{d}/c$ (by the upper bound on the approximation), so it follows that the size of an optimal clustering is at least $\hat{d}/c$ (by the lower bound on the approximation). Also, since the $(k+1)$th center maximises $\hat{d}$ amongst all non-centers, it follows that the distance from any point to its closest center is at most $\hat{d}$. So, the distance between any two points in a cluster is at most $2\hat{d}$, by the triangle inequality. The approximation factor is then $2\hat{d}/(\hat{d}/c) = 2c$. □

Corollary 2.4.1 *For the vector distance approximations in this chapter, we can find a clustering that is within a factor of $2+\epsilon'$ of the optimal with probability $1-\delta$. The running time is $O(\frac{1}{\epsilon^2}m(n+k)\log m)$.*

Proof. We use the above Theorem, but we need to attend to a couple of technical details. We assume that we are using some approximate distance oracle to approximate distances efficiently. In practice, these will be the sketches that we have described in Section 2.2.4, which are $(1\pm\epsilon)$ approximations with probability $1-\delta$. Firstly, we re-scale the approximation by a factor of $(1-\epsilon)$ so that $d(\boldsymbol{a},\boldsymbol{b}) \leq \hat{d}(\boldsymbol{a},\boldsymbol{b}) \leq \frac{1+\epsilon}{1-\epsilon}d(\boldsymbol{a},\boldsymbol{b})$. For $\epsilon \leq 1/2$ then $\frac{1+\epsilon}{1-\epsilon} \leq 1+4\epsilon$. Hence we set $\epsilon = \epsilon'/4$. Secondly, the above proof requires that the approximation is always within the stated bounds. To overcome this, we set the probability of failure, δ, so that out of all $O(m^2)$ comparisons, the probability that they all succeed is some constant, δ'. So $(1-\delta)^{m^2} = (1-\delta')$ and hence

$$(1 - m^2\delta + O(m^4\delta^2)) = (1-\delta')$$

This leads us to choose $\delta = \delta'/m^\alpha$ for some small constant α. The space needed for each distance approximation is then $O(1/\epsilon'^2(\alpha\log m + \log 1/\delta'))$. The clustering is then, with probability δ', within $2+\epsilon'$ of the optimal. The running time of this algorithm is $O(\frac{1}{\epsilon^2}mn\log m)$ to make the sketches and then $O(\frac{1}{\epsilon^2}km\log m)$ to run the algorithm. □

2.5 Discussion

We have seen a number of problems and their solutions for questions relating to vectors and (equivalently) binary strings. The main themes and concepts introduced in this chapter are:

- The idea of a sketch, which allows us to approximate the distance between objects by manipulating sketches which are significantly smaller than the objects they represent. Certain sketches can be computed in a streaming model, where the data is too large to be held in memory and instead is accessed with a single pass over it.
- Efficient ways to sketch vectors for the L_p norms for all $0 < p \leq 2$ in the most general streaming model. These sketches can be computed in a uniform way using pseudo-random variables from stable distributions in space that is effectively independent of the size of the vector being sketched.
- Applications of sketching to the Hamming distance for binary vectors and arbitrary strings. These vectors can be large and sparse without affecting the size of the sketch, and they can be presented in the streaming model.
- Proof that the intersection space is hard to approximate, but that a rough kind of sketching is possible on intersection size and set resemblance.
- The efficient approximate solution to a number of geometric problems under the Hamming distance, Euclidean Distance and Set Resemblance measure.

Chapter 3

Searching Sequences

A, B, C,
It's easy as 1, 2, 3,
As simple as Do re mi,
A, B, C, 1, 2, 3,
Baby you and me. [Fiv70]

3.1 Introduction

In this chapter, embeddings of the permutation distances discussed in Chapter 1 into vector distances are described. Unlike the embeddings of vector distances we saw in Chapter 2, these have fixed distortion factors (compared to the $1 \pm \epsilon$ distortion factors for L_p distances). On the other hand, these are non-probabilistic embeddings: the distance between any possible pair of permutations is distorted by no more than the stated distortion factor. These embeddings are constructed by using combinatorial observations about the different permutation distances. Once we have embedded permutations into vector distances, we can go on and embed these vectors into much smaller vector spaces using the embeddings in Chapter 2.

This approach yields many other interesting results. In Section 2.4 we looked at some geometric problems on vectors. We can ask the same questions for permutations, replacing the vector distances with permutation distances. By using the embeddings of permutations into vector spaces, it is often straightforward to use algorithms which solve geometric problems in those vector spaces to solve the permutation problems. Further, we can use the properties of our embeddings to solve approximate pattern matching problems under permutation distances.

3.1.1 Computational Biology Background

Following the successful sequencing of the Human Genome [V+01], the attention is shifting from raw sequence data to genetic maps. Comparative studies of gene loci among closely related species provide clues towards understanding the complex phylogenetic relationships between species and their evolutionary order. Genetic maps of two species can be thought of as permutations of homologous genes and the number of chromosomal rearrangements in the form of deletions, copies, inversions, transpositions to transform one such permutation to another can be used as a measure of their evolutionary distance. Computational methods for measuring genetic distance between species are an active area of research in computational genomics, especially in the context of comparative mapping [NT84], using reversal distance, transposition distance or other measures. In a more general setting it is of interest not only to compute the distances between two permutations but also to find the closest gene permutation to a given one in a database or to approximately find a given permutation of genes in a larger sequence. Given the representation as permutations, these can all be abstracted as permutation editing and matching problems. More complex notions of genetic distance which take into account that (1) the genome is composed of multiple chromosomes [FNS96, SN96], or (2) exact order of the genes within a genome is not necessarily known [GGP+99] have recently been proposed. Unfortunately computing the genetic distance under such extensions is NP-hard [DJK+98, GGP+99], and polynomial time algorithms only exist when a limited subset of chromosomal rearrangements are permitted [HP95, KR95]. Thus modelling genomes as permutations provides a simple but tractable means for computing genetic distance between species.

We know that permutations are ordered sequences over some alphabet with no repetitions allowed. A permutation is a string, although strings are not generally permutations since they are allowed to repeat symbols. We study problems of computing the pairwise edit distances described in Section 1.3.4 and performing similarity searching, matching and so on, motivated by Computational Biology and other scenarios. We adopt a general approach to solving all such problems on permutations: develop an embedding of permutations into vector spaces such that the distance between the resulting vectors approximates the distance between any two permutations. Thus permutation editing and matching problems reduce to natural problems on vector spaces.

Even though we have motivated permutation editing, matching and similarity searching problems from Computational Biology applications, there are other reasons for their study. Permutations form an interesting class of combinatorial objects by themselves, and therefore it is quite natural to study the complexity of computing edit distances between permutations, and to do similarity searching. In addition, they arise in many applications such as sorting networks and group theory.

3.1.2 Results

In Section 3.2 we present embeddings of permutation distances into previously described spaces such as Hamming or Set Intersection. The embeddings preserve the original distances within a small constant or logarithmic factor, and embed into a vector space that is quadratically larger than the size of the permutation being embedded. The embeddings use a technique of capturing the relative layout of pairs of symbols in the permutations by two dimensional matrices. These embeddings capture the relevant pairs that help approximate the permutation distances accurately and the resulting matrices are often sparse.

The embeddings above immediately give approximation algorithms for computing the distance between two permutations in (near) linear time. Studies have been made in Computational Biology on this problem [BP93, BP98, Chr98a, Chr98b, KS95, KR95, BHK01]. In addition, we can use the embeddings above to solve several further algorithmic problems: (1) Computing permutation distances in a distributed or Communication Complexity setting wherein the number of bits exchanged is the prime criterion for efficiency, and also in the sketch and streaming models. (2) Providing efficient approximate nearest neighbor searches and clusterings for permutation distances. These are described in Section 3.3. While there exist some approximation algorithms for computing permutation distances ([BP98, Chr98a, KR95, KS95, BHK01] and many others), they are not communication complexity protocols. In particular, they rely on consistent relabelling so one of them is an identity permutation. This is clearly not possible in the sketch model.

The problem of Approximate Pattern Matching for Permutations is, given a long text string and a pattern permutation, to find all occurrences of the pattern in the text with at most a given threshold of distance. This is an application of the general pattern matching problem of Definition 1.5.1 to the permutation distances. In Section 3.3.4 we present highly efficient, linear or near-linear time approximations for the permutation matching problems. This is intriguing since approximately solving string matching problems with corresponding string distances seems to be harder, with best known algorithms taking much longer. For example, existing algorithms for approximate string matching with edits take worst-case time $\Omega(nm/\log mn)$ for n-long text and m-long pattern where edits are transpositions, character indels and substitutions (see the survey in [Nav01]). In contrast, our algorithms take only $O(n + m)$ or $O(n \log m)$ time for permutations.

3.2 Embeddings of Permutation Distances

The best known algorithms for sorting by reversals and by transpositions all assume that that the target permutation is known in advance, and so uniformly take the first step of relabelling one of the sequences as the identity permutation, and proceed to sort the correspondingly relabelled second sequence into order. However, when trying to carry out approximate pattern matching or creating a sketch, this is not a reasonable step — in particular for sketching, at the time of creating the sketch, there is no goal sequence. Hence, we cannot follow any of the existing approaches which assume both permutations are known. Instead, we shall give approximations that are sometimes slightly weaker than the best known approximations in order to admit calculation in the sketching model and for the other problems that we consider in this chapter.

3.2.1 Swap Distance

We begin our demonstration of embeddings of permutation distances into vector spaces with a very simple example, the Swap Distance (see Definition 1.3.3), which illustrates the basic principles that we will use repeatedly. That is, we show how to define a transformation on a permutation into a binary vector, so that the vector distance between pairs of transforms gives (an approximation to) the distance of interest. Recall from Section 1.3 the definition of the inverse of a permutation, P^{-1}, such that $P^{-1}[i] = j$ if $P[j] = i$.

Definition 3.2.1 *Define an* inversion *in a permutation P relative to a permutation Q as a pair (i, j) such that $P^{-1}[i] > P^{-1}[j]$ but $Q^{-1}[i] < Q^{-1}[j]$.*

This definition of an inversion is a generalisation of the definition of an inversion given in [Knu98] (see section 5.1.1). There, Q is the identity permutation, and so an inversion is a pair $i < j$ where $P^{-1}[i] > P^{-1}[j]$. It is important to keep this notion of inversion distinct from the idea of 'inversion distance' in computational biology (a synonym for the reversal distance).

Definition 3.2.2 *Define an $n \times n$ binary matrix $S(P)$ by*

$$\begin{array}{ll} P^{-1}[i] < P^{-1}[j] \wedge i < j & \Longrightarrow S(P)[i,j] = 1 \\ P^{-1}[i] > P^{-1}[j] \vee i \geq j & \Longrightarrow S(P)[i,j] = 0 \end{array}$$

The Hamming distance between two matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ is denoted $||\boldsymbol{A} - \boldsymbol{B}||_H$. The Hamming distance between two matrices is just the Hamming distance between two vectors obtained by linearising the two matrices in any manner.

Lemma 3.2.1 $\mathrm{swap}(P, Q) = ||S(P) - S(Q)||_H$

Proof. We prove two separate facts: firstly, that $||S(P)-S(Q)||_H$ counts exactly the number of inversions between P and Q, and secondly that the number of inversions is exactly the swap distance.
(i) Consider any pair $1 \leq i < j \leq n$. Suppose (i, j) is an inversion in P relative to Q. Then either i occurs before j in P but not in Q or vice-versa. Then $S(P)[i, j] \neq S(Q)[i, j]$, and hence the pair (i, j) contributes 1 to the Hamming distance $||S(P) - S(Q)||_H$. If (i, j) is not an inversion, then i and j occur in the same relative order in both P and Q and so $S(P)[i, j] = S(Q)[i, j]$. So there is no contribution to the Hamming distance from this pair. In conclusion, every inversion, and only an inversion, contributes one to the Hamming distance; hence the Hamming distance between these matrices is exactly the number of inversions.
(ii) A pair of permutations are identical if and only if there are no inversions between them. It is certainly the case that every inversion requires a swap to rectify it: each swap can remove at most one inversion, since it affects a single pair of symbols, which can be involved in a single inversion. It remains to show that there is always a swap that reduces the number of inversions. Suppose there were no swaps that reduce the number of inversions. Then every adjacent pair of elements in the permutation is in the same order in both permutations. But if this were true, then the two permutations would be identical. Hence, if two permutations are different then there is always a swap that reduces the number of inversions by one. In conclusion, the number of inversions is identical to the swap distance, and combining this with the proof of part (i) shows the full lemma. □

Example. Consider two permutations of the letters $\{A, E, I, M, N, R, S\}$, $P = SEMINAR$ and $Q = REMAINS$. Their transforms are respectively:

S(P)	A	E	I	M	N	R	S
A	0	0	0	0	0	1	0
E	0	0	1	1	1	1	0
I	0	0	0	0	1	1	0
M	0	0	0	0	1	1	0
N	0	0	0	0	0	1	0
R	0	0	0	0	0	0	0
S	0	0	0	0	0	0	0

S(Q)	A	E	I	M	N	R	S
A	0	0	1	0	1	0	1
E	0	0	1	1	1	0	1
I	0	0	0	0	1	0	1
M	0	0	0	0	1	0	1
N	0	0	0	0	0	0	1
R	0	0	0	0	0	0	1
S	0	0	0	0	0	0	0

By comparing these tables, we see that $||S(P) - S(Q)||_H = 13$. So the swap distance should be 13. We verify this by transforming P into Q:

$SEMINAR \rightarrow_1 ESMINAR \rightarrow_2 EMSINAR \rightarrow_3 EMISNAR \rightarrow_4 EMINSAR \rightarrow_5 EMINASR \rightarrow_6 EMINARS \rightarrow_7 EMINRAS \rightarrow_8 EMIRNAS \rightarrow_9 EMRINAS \rightarrow_{10} ERMINAS \rightarrow_{11} REMINAS \rightarrow_{12} REMIANS \rightarrow_{13} REMAINS$

Each step reduces the number of inversions by one.

Extension to strings

A particularly nice feature of this approach to swap distance is that it immediately translates to the equivalent distance on strings. That is, given two strings a and b (with the same quantity of each character in the alphabet σ), the swap distance is the number of swaps of adjacent pairs of characters to transform x into y, denoted as $\mathrm{swap}'(x, y)$.

Definition 3.2.3 *We define the function* $\mathrm{permify}(a)$ *on a string a. Suppose* $a[j]$ *is* α*. Then* $\mathrm{permify}(a)[j] = \alpha_i$ *if this is the* i*'th* α *that has been seen reading left to right in a.*

Lemma 3.2.2 $\mathrm{swap}'(a, b) = \mathrm{swap}(\mathrm{permify}(a), \mathrm{permify}(b))$

Proof. This approach identifies the i'th a in x with the i'th a in y. Suppose we did not do this, so there exists some values i, j, i', j' where a_i is identified with $a_{i'}$ and a_j is identified with $a_{j'}$, and $i < j$ but $i' > j'$. Then there must be a swap which exchanges a_i with a_j in transforming x into y. But this swap is superfluous, since we can omit it and the sequence of intermediate strings is identical. Therefore, we never swap any pair of identical characters, and the ordering of the a's between x and y is unchanged. □

Example. Suppose we wish to demonstrate that $Mike\_Paterson$ is $Mistake\_Prone$. Then we apply $\mathrm{permify}$ to both strings to yield $M_1 i_1 k_1 e_1 \_{}_1 P_1 a_1 t_1 e_2 r_1 s_1 o_1 n_1$ and $M_1 i_1 s_1 t_1 a_1 k_1 e_1 \_{}_1 P_1 r_1 o_1 n_1 e_2$. Analysing the inversions of these permutations shows that

$$\mathrm{swap}'(Mike\_Paterson, Mistake\_Prone) = 20$$

Note the largest possible swap distance (referred to as the diameter) can be worked out from this transformation: the greatest value of the Hamming distance will be seen if one transformation has 1s in every possible location, and another has 0s — corresponding to the identity permutation and its reverse. The Hamming distance, and hence the swap distance, is $n(n-1)/2$. In this case, where $n = 13$, the largest possible swap distance is 78.

Although straightforward, the analysis of swap distance shows us some interesting points. Firstly, we see that initially different-seeming distances can be reconciled: the swap distance on permutations can be recast as a problem on Hamming space. Although the size of the Hamming space is quadratically larger than the permutation space, this makes little difference for most applications, since they can be made independent of the size of the space. This embedding is an isometry: the distortion factor is 1, so the distance is preserved exactly. This is a highly desirable feature of an embedding, although it will be quite rare. Secondly, it shows how study of a permutation distance can lead to a greater understanding of a corresponding string distance. We would hope that the same is true of the other permutation distances that we study. A lower bound we could prove for a permutation distance might apply to a corresponding string distance, since a permutation is just a special case of a string. This is valid when the strings are drawn from a large alphabet, one which is at least linear in the length of the sequence, so the permutations could be expressed as strings.

3.2.2 Reversal Distance

To tackle reversal distance, we should like to show a similar result to that described for swap distance: that we can make a vector from a permutation so that a vector distance gives (an approximation to) the reversal distance. We begin by considering a method that finds a 2-approximation to the reversal distance, $r(P, Q)$ (see Definition 1.3.1). This was originally given as Theorem 1 of [KS95], and was also described in Chapter 19 of [Gus97]. The goal there is to take an arbitrary permutation and apply reversals until it is sorted — that is, until it is the identity permutation. The approach is based on local features of the permutation called breakpoints, which consider adjacent pairs of symbols. For uniformity, we extend *all* permutations P by adding $P[0] = 0$ and $P[n+1] = n+1$, where n is the length of P. This allows the first and last symbols of P to be treated identically to the other symbols (as having two adjacent symbols).

Definition 3.2.4 *A* reversal breakpoint *is a location i in a permutation such that $|P[i] - P[i+1]| \neq 1$. The number of reversal breakpoints in a permutation is denoted by $\phi(P)$.*

Theorem 3.2.1 (from [KS95]) $r(P, I) \leq \phi(P) \leq 2r(P, I)$

Proof. For the upper bound, consider the identity permutation, and observe that it has $\phi(I) = 0$. Therefore, in transforming P into I the goal is to remove all breakpoints. Any single reversal can remove at most 2 breakpoints; hence the upper bound follows. For the lower bound, we must consider increasing strips (sequences of the form $j\ j+1\ j+2 \ldots$) and decreasing strips (sequences of the form $j\ j-1\ j-2 \ldots$). It can be shown that
(i) Any permutation with a decreasing strip has a reversal that removes 1 breakpoint.
(ii) If every possible reversal on a permutation with a decreasing strip leaves no decreasing strips, then there is a reversal which removes 2 breakpoints.
(iii) Any permutation with only increasing strips has a reversal that does not affect the number of breakpoints but creates decreasing strips.
With these facts, all breakpoints can be removed using at most as many reversals as there are breakpoints: greedily look for reversals which remove as many breakpoints as possible, and favour those that leave decreasing strips. Whenever there is no way to retain decreasing strips, case (ii) applies, so we have an extra 'credit' to pay for the next move which will be of case (iii). The first reversal is 'paid for' by observing that the final reversal always removes two breakpoints. □

We now show how to adapt this method to give an embedding of reversal distance into the Hamming distance. Firstly we extend our notion of a reversal breakpoint.

Definition 3.2.5 *Define a* Reversal Breakpoint of P relative to Q *as a location, i, where the symbol following $P[i]$ in P is not adjacent to $P[i]$ where it occurs in Q. Formally, this is when $|Q^{-1}[P[i]] - Q^{-1}[P[i+1]]| \neq 1$. We denote the total number of such breakpoints as $\phi(P, Q)$.*

Next, we give the embedding into the Hamming distance by defining a new vector transformation of a permutation.

Definition 3.2.6 *We define a two dimensional matrix, $R(P)$, as a binary matrix of size $(n+2) \times (n+2)$. For all $0 \le i < j \le n+1$, set $R(P)[i,j]$ to 1 if $j > i$ and i is adjacent to j in P. Otherwise, $R[i,j] = 0$. So*

$$\begin{array}{lcl} |P^{-1}[i] - P^{-1}[j]| = 1 & \Longrightarrow & R[i,j] = 1 \\ \text{otherwise} & \Longrightarrow & R[i,j] = 0 \end{array}$$

A few simple observations about the matrix $R(P)$ follow: it has $n+1$ non-zero entries if P is a permutation of n items. From $R(P)$ it is possible to reconstruct P, so P is unique for $R(P)$.

Theorem 3.2.2 $r(P,Q) \le \frac{1}{2}||R(P) - R(Q)||_H \le 2r(P,Q)$

Proof. We show that the reversal breakpoints of P relative to Q are closely related to the earlier notion of a reversal breakpoint. Clearly, if $P = Q$, then $\phi(P,Q) = 0$, and this is the only way in which the count is zero. So in transforming P into Q using reversals, our goal is to reduce ϕ to zero. Now consider relabelling Q as the identity permutation, and applying this same relabelling to P generating $Q^{-1} \circ P$. Since we have only changed the labels, this does not affect the number of reversal breakpoints, since these do not depend on the names of the labels. Hence $\phi(P,Q) = \phi(Q^{-1} \circ P, I)$. But from the definitions, $\phi(P', I) = \phi(P')$. Also, as we have already mentioned, $r(P,Q) = r(Q^{-1} \circ P, I)$ (this is tacitly assumed by any approach that relabels permutations so that the target is the identity permutation). Hence by using Theorem 3.2.1 above, $r(P,Q) \le \phi(P,Q) \le 2r(P,Q)$.

It remains to show that $||R(P) - R(Q)||_H = 2\phi(P,Q)$. Suppose that $R(P)[i,j] = 1$ and $R(Q)[i,j] = 0$. This means that i and j are adjacent in P but not in Q. If we sum the number of distinct pairs i,j which are adjacent in P but not in Q, then this finds $\phi(P,Q)$. This is because every breakpoint will generate such a pair, and such pairs can only arise from breakpoints. An identical argument follows when $R(P)[i,j] = 0$ and $R(Q)[i,j] = 1$, yielding $\phi(Q,P)$. Since $\phi(P,Q) = \phi(Q,P)$, it follows that $||R(P) - R(Q)||_H$ counts each breakpoint exactly twice. □

Example. Observe that P and Q can be parsed canonically into maximal contiguous subsequences that are common to both either forwards or backwards. For example, if $Q = 0\ 2\ 4\ 3\ 7\ 1\ 6\ 5\ 8$ and $P = 0\ 2\ 3\ 7\ 5\ 6\ 1\ 4\ 8$, then the parsing of Q is (0 2)(4)(3 7)(1 6 5)(8), and P is (0 2)(3 7)(5 6 1)(4)(8). If we choose to fix Q and relabel P accordingly, then in this example P becomes (0 1)(3 4)(7 6 5)(2)(8). That is, this parsing tells us where the reversal breakpoints are. Since both sequences are parsed using the same number of segments, this tells us that necessarily $\phi(P,Q) = \phi(Q,P)$. In this example, $\phi(P,Q) = 4$.

R(P)	0	1	2	3	4	5	6	7	8
0	0	0	1	0	0	0	0	0	0
1	0	0	0	0	1	0	1	0	0
2	0	0	0	1	0	0	0	0	0
3	0	0	0	0	0	0	0	1	0
4	0	0	0	0	0	0	0	0	1
5	0	0	0	0	0	0	1	1	0
6	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0

R(Q)	0	1	2	3	4	5	6	7	8
0	0	0	1	0	0	0	0	0	0
1	0	0	0	0	0	0	1	1	0
2	0	0	0	0	1	0	0	0	0
3	0	0	0	0	1	0	0	1	0
4	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	1	0	1
6	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0

Comparing $R(P)$ and $R(Q)$ we see that $||R(P) - R(Q)||_H = 2\phi(P,Q) = 8$. So according to Theorem 3.2.2, $r(P,Q)$ must be between 2 and 4. In fact, $r(P,Q) = 3$ as demonstrated by the following sequence of reversals[1]:

[1]The only way there could be a pair of reversals to turn P into Q would be if there were two reversals which removed two breakpoints each. There is no reversal that removes two breakpoints from P.

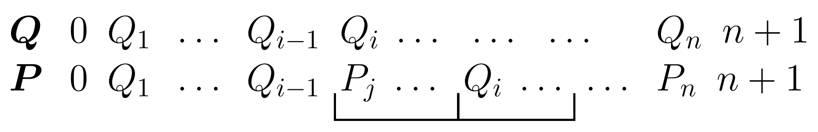

$\boldsymbol{Q}$ 0 Q_1 ... Q_{i-1} Q_i ... ... Q_n $n+1$
$\boldsymbol{P}$ 0 Q_1 ... Q_{i-1} P_j ... Q_i ... ... P_n $n+1$

Figure 3.1: There is always a transposition that removes one breakpoint

$$0\ 2\ 3\ 7\ \underline{5\ 6\ 1}\ 4\ 8 \longrightarrow 0\ 2\ \underline{3\ 7\ 1\ 6\ 5\ 4}\ 8 \longrightarrow 0\ 2\ 4\ \underline{5\ 6\ 1\ 7\ 3}\ 8 \longrightarrow 0\ 2\ 4\ 3\ 7\ 1\ 6\ 5\ 8$$

3.2.3 Transposition Distance

The Transposition Distance (Definition 1.3.2) forms a very natural companion measure to the Reversal Distance. We shall give a transformation that embeds Transposition distance into Hamming distance and has an approximation factor of 2.

Definition 3.2.7 *We define $T(P)$, a binary matrix for a permutation P such that $T(P)[i,j] = 1$ if j immediately follows i in P. So*

$$\begin{array}{lll} P^{-1}[i] + 1 = P^{-1}[j] & \implies & T(P)[i,j] = 1 \\ \text{otherwise} & \implies & T(P)[i,j] = 0 \end{array}$$

From $T(P)$ it is possible to uniquely determine P. Although $T(P)$ has $O(n^2)$ entries, only $n+1$ of these are set to 1, the rest are 0.

Definition 3.2.8 *Define a* Transposition Breakpoint *in a permutation P relative to another permutation Q as a location, i, such that $P[i+1]$ does not immediately follow $P[i]$ when it occurs in Q,* [2] *that is $Q^{-1}[P[i]] + 1 \neq Q^{-1}[P[i+1]]$. Let the total number of such transposition breakpoints between P and Q be denoted as $tb(P,Q)$.*

Theorem 3.2.3 $t(P,Q) \leq \frac{1}{2}||T(P) - T(Q)||_H \leq 3t(P,Q)$

Proof. Observe that to convert P to Q we must remove all breakpoints, since $tb(Q,Q) = 0$. A single transposition affects three locations and so could 'fix' at most three breakpoints — this gives a lower bound. Also, we can always fix at least one breakpoint per transposition using the trivial greedy algorithm: find the first location where $P[i] \neq Q[i]$, searching from the left. Find $P^{-1}[Q[i]]$ (it must be at a location greater than i), and choose a block starting there and extending to the next transposition breakpoint. Move this block into place so that now $Q[i-1]$ is next to $Q[i]$. This transposition removes the transposition breakpoint caused because $Q[i]$ was not next to $Q[i-1]$ in P. We know that there was also a transposition breakpoint at the location where $Q[i]$ was in P, and at the end of the move. Hence, we cannot have introduced any new transposition breakpoints into P and we have removed one. This is always possible, and so we have the upper bound. This is illustrated in Figure 3.1. Hence $t(P,Q) \leq tb(P,Q) \leq 3t(P,Q)$.

We now need to show that $||T(P) - T(Q)||_H = 2tb(P,Q)$. $T(P)[i,j] = 1$ and $T(Q)[i,j] = 0$ if and only if there is a transposition breakpoint in Q at the location of i, so summing these contributions generates $tb(P,Q)$. A symmetrical argument holds when $T(P)[i,j] = 0$ and $T(Q)[i,j] = 1$. Because $tb(P,Q) = tb(Q,P)$, then these two cases summed generate exactly $||T(P) - T(Q)||_H = 2tb(P,Q)$. □

This is sufficient to show that we can find a 3-approximation. In fact, this can be improved using the work of others.

[2] *As usual, we extend all permutations so that the first symbol is 0 and their last is $n+1$.*

Theorem 3.2.4 *Any method which can sort a permutation by removing b transposition breakpoints in m moves means that* $t(P,Q) \leq \frac{m}{2b}||T(P) - T(Q)||_H \leq \frac{3m}{b}t(P,Q)$

Proof. Relabelling both permutations consistently does not change the transposition distance between them. So, by analogy with reversal distance $tb(P,Q) = tb(Q^{-1} \circ P, I) = \frac{1}{2}||T(P) - T(Q)||_H = \frac{1}{2}||T(Q^{-1} \circ P) - T(I)||_H$. In other words, $\frac{1}{2}||T(P) - T(Q)||_H$ counts exactly the number of transposition breakpoints between the permutation $Q^{-1} \circ P$ and the identity permutation. Any method which guarantees to remove b transposition breakpoints in m moves will be able to sort this permutation in $m * tb(P,Q)/b$ moves, and we know that since no more than 3 breakpoints can be removed in any move, so at least $tb(P,Q)/3$ moves are necessary. Also, every move that is made on $Q^{-1} \circ P$ can be carried out on P, and will remove the same number of breakpoints. So $t(P,Q) \leq \frac{m}{2b}||T(P) - T(Q)||_H \leq \frac{3m}{b}t(P,Q)$ as required. □

Observe that our original theorem is the case where $b = m = 1$.

Corollary 3.2.1 $t(P,Q) \leq \frac{1}{3}||T(P) - T(Q)||_H \leq 2t(P,Q)$

This corollary follows from the work presented in [EEK+01], which proves in Lemma 5.1 that in two transpositions it is always possible to remove three breakpoints, setting $b = 3$ and $m = 2$ in the above theorem. Consequently, $t(P,Q) \leq \frac{2}{3}tb(P,Q)$.

Example. Suppose $P = 0\ 4\ 3\ 1\ 6\ 7\ 5\ 2\ 8$ and $Q = 0\ 3\ 1\ 6\ 4\ 7\ 2\ 5\ 8$. Then as before we can parse P into contiguous subsequences of Q with the gaps corresponding to the transposition breakpoints: $P =$ (0) (4) (3 1 6) (7) (5) (2) (8). So $tb(P,Q) = 6$. From the above theorem and corollary, this means that $t(P,Q)$ is between 2 and 4.

T(P)	0	1	2	3	4	5	6	7	8
0	0	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0
2	0	0	0	0	0	0	0	0	1
3	0	1	0	0	0	0	0	0	0
4	0	0	0	1	0	0	0	0	0
5	0	0	1	0	0	0	0	0	0
6	0	0	0	0	0	0	0	1	0
7	0	0	0	0	0	1	0	0	0
8	0	0	0	0	0	0	0	0	0

T(Q)	0	1	2	3	4	5	6	7	8
0	0	0	0	1	0	0	0	0	0
1	0	0	0	0	0	0	1	0	0
2	0	0	0	0	0	1	0	0	0
3	0	1	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	1	0
5	0	0	0	0	0	0	0	0	1
6	0	0	0	0	1	0	0	0	0
7	0	0	1	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0

$||T(P) - T(Q)||_H = 2tb(P,Q) = 12$, and so $2 \leq t(P,Q) \leq 4$. In this case, the lower bound is tight, we need only two moves to transform P into Q:

$$0\ \underline{4}\ \underline{3\ 1\ 6}\ 7\ 5\ 2\ 8 \longrightarrow 0\ 3\ 1\ 6\ 4\ 7\ \underline{5}\ \underline{2}\ 8 \longrightarrow 0\ 3\ 1\ 6\ 4\ 7\ 2\ 5\ 8$$

Why this approach won't work for strings

As with Swap distance, we might hope that looking at pairs of characters might extend to equivalent string distances. Sadly, this is not the case. Consider the two strings $x = a\ b\ a\ b\ a\ b\ \ldots\ a\ c\ a\ c\ a\ c\ \ldots\ a$ and $y = a\ b\ a\ c\ a\ b\ a\ c\ a\ \ldots\ a$ We can arrange it so that if we just consider adjacent pairs of characters, as we have for reversal distances and transposition distances, then these two strings are indistinguishable: the number of adjacent pairs $(a,b), (b,a), (a,c)$ and (c,a) is the same. But their transposition or reversal distance is $\Omega(n)$. So to be able to handle string distances under this general approach of encoding features of the string in a vector fashion (with bits indicating the presence or absence of features in the string), we need to look at more than just character adjacencies: substrings of all lengths may need to be considered. This will have implications on the quality of approximations possible for these distances, since editing operations could affect either many such features, or very few.

3.2.4 Permutation Edit Distance

The Permutation Edit distance or Ulam Metric, $d(P, Q)$ (Definition 1.3.4) is closely related to the longest common subsequence, and in turn to the longest increasing subsequence. It can be found exactly in time $O(n \log \log n)$ using an appropriate data structure [HS77]. Somewhat contrarily, we will show an approximation scheme for this distance with a $\log n$ factor,[3] because our goal is to produce an embedding which is computable in the sketch model, where the Longest Increasing Subsequence algorithm will not work. This embeds the distance into the Intersection size. Unlike the earlier embeddings, this is not symmetric, and uses two different matrix transformations.

Definition 3.2.9 *We shall define $A(P)$ as an $n \times n$ binary matrix derived from a permutation P of length n. $A_k(P)[i, j]$ is set to one if a symbol i occurs a distance of exactly 2^k before j in P. Otherwise, $A_k(P)[i, j] = 0$. $A(P)$ is formed by taking the union of the matrices $A_0 \ldots A_{\lceil \log n \rceil - 1}$. That is,*

$$\begin{array}{ll} \exists k \in \mathcal{N} : (P^{-1}[i] + 2^k = P^{-1}[j]) & \Longrightarrow A(P)[i, j] = 1 \\ \text{otherwise} & \Longrightarrow A(P)[i, j] = 0 \end{array}$$

Definition 3.2.10 *Let $B(Q)$ be an $n \times n$ binary matrix defined on a permutation Q such that $B(Q)[i, j]$ is zero if i occurs before j in Q. Otherwise $B(Q)[i, j] = 1$. Thus,*

$$\begin{array}{ll} Q^{-1}[i] < Q^{-1}[j] & \Longrightarrow B(Q)[i, j] = 0 \\ \text{otherwise} & \Longrightarrow B(Q)[i, j] = 1 \end{array}$$

Note that $A(P)$ is a binary matrix, and $n \log n - 2^{\lceil \log n \rceil} + 1$ entries are 1. $B(Q)$ is also a binary matrix, and $n^2/2 + n/2$ entries are 1.

Definition 3.2.11 *Finally, define $D(P, Q)$ as the size of the intersection between $A(P)$ and $B(Q)$. Put another way, this intersection can be calculated using multiplication of the elements of the matrices, pairwise:*

$$D(P, Q) = \sum_{i,j} (A(P)[i, j] \times B(Q)[i, j])$$

The effect of this intersection is shown graphically in Figure 3.2.

Theorem 3.2.5 $d(P, Q) \leq D(P, Q) \leq \lceil \log n \rceil \cdot d(P, Q)$.

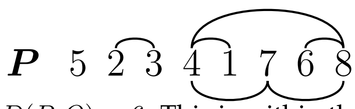
Proof.
(i) $D(P, Q) \leq \lceil \log n \rceil \cdot d(P, Q)$
Consider the pairs (i, j) such that $A(P)[i, j] = B(Q)[i, j] = 1$. The number of such pairs is exactly $D(P, Q)$. Each of these pairs has i occurring before j in P, but the other way round in Q, and so one of either i or j must be moved to turn P into Q. So in effect, these pairs represent a "to-do" list of changes that must be made. By construction of A, any symbol i appears at most $\log n$ times amongst these pairs. Hence whenever a move is made, at most $\log n$ pairs can be removed from this to-do list. It therefore follows that in each move, D can change by at most $\log n$. If at every step we change D by at most $\log n$, then this bounds the minimum number of operations possible to transform P into Q as $d(P, Q)\lceil \log n \rceil \geq D(P, Q)$.
(ii) $d(P, Q) \leq D(P, Q)$
We shall show the bound by concentrating on the fact that an optimal edit sequence preserves a Longest

[3]Here and throughout, all logarithms are taken to base 2

$$\boldsymbol{P} \quad \underline{5}\ 2\ \underline{3}\ 4\ \underline{1}\ \underline{7}\ \underline{6}\ 8$$

$$\boldsymbol{Q} \quad \underline{5}\ 8\ \underline{3}\ \underline{1}\ 2\ \underline{7}\ \underline{6}\ 4$$

Two permutations, P and Q. A longest common subsequence of length 5 is underlined, so the permutation editing distance, $d(P, Q)$ is 3. We can illustrate $D(P, Q)$ by marking on P all of the pairs which contribute to the intersection size of the transforms:



There are 6 such pairs and so $D(P, Q) = 6$. This is within the approximation bounds, since $d(P, Q) = 3 \leq 6 \leq 3 \times 3$.

Figure 3.2: Illustrating Permutation Edit Distance embedding

Common Subsequence of the two sequences. Note that an optimal edit sequence will have length $n - LCS(P, Q)$: every symbol that is not moved must form part of a common subsequence of P and Q and so an optimal edit scheme will ensure that this common subsequence is as long as possible. Consider the relabelling of Q so that for all i, $Q[i]$ is relabelled with i. We analyse the effect of applying this relabelling to P and examine its longest increasing subsequence. Call this relabelled sequence P'. It is defined as $P'[i] = Q^{-1}[P[i]]$. Clearly, every common subsequence of P and Q corresponds to a common subsequence of P' and I since we have just relabelled distinct symbols. In particular, the length of the longest common subsequence of P and Q is not altered. Because Q is replaced by a strictly increasing sequence, it follows that each Longest Common Subsequence of P and Q corresponds exactly to each Longest Increasing Subsequence of P', whose length is denoted by $LIS(P')$. What $D(P, Q)$ told us was that we should count 1 if a symbol is 2^k to the right of the i's location in P but is anywhere to the left of i in Q. When we relabel according to Q, exactly the same location pairs will contribute to $D(P', I)$ as contributed to $D(P, Q)$, since we have just given new names to symbols. Because we are dealing with I, this means we count 1 for each pair of symbols i, j where $i > j$ and j occurs 2^k symbols to the right of i for some integer k. This we can derive from the definition: $A(P')[i, j]$ only counts 1 when i and j are separated by 2^k symbols, and $B(I)$ counts when $I^{-1}[i] > I^{-1}[j]$. Since $I^{-1}[x] = x$ for any x in the permutation, this simplifies to $i > j$.

In order to characterise the Longest Increasing Subsequence of P', we shall split P' into two subsequences, one of which consists only of the symbols at odd locations in P', and the other of the symbols which occur at even locations. Let $odd(P')$ be the sequence formed as $P'[1]P'[3] \dots P'[2i - 1] \dots P'[2\lfloor(|P| + 1)/2\rfloor - 1]$, and similarly $even(P') = P[2] \dots P[2i] \dots P[2\lfloor|P|/2\rfloor]$. Symbols of P' will now be referred to as 'odd symbols' or 'even symbols': this refers only to their location, not whether the value of a symbol is odd or even. Suppose $LIS(odd(P'))$ is the length of a longest increasing subsequence of symbols at odd locations in P', and $LIS(even(P'))$ is similarly defined for the even symbols. Define $b(P')$ as the number of locations ('sequence breakpoints') where $P'[i] > P'[i + 1]$. Formally then, $b(P') = |\{i | P'[i] > P'[i + 1]\}|$.

Lemma 3.2.3 *$LIS(P') \geq LIS(odd(P')) + LIS(even(P')) - b(P')$.*

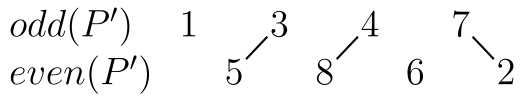
Proof. Let S_{even} represent an increasing sequence of even symbols whose length is $LIS(even(P))$, and define S_{odd} similarly. We shall see how we can build a longer increasing subsequence starting from each of the subsequences of even and odd symbols. Consider a symbol of S_{even}, $P'[i]$ and the subsequent symbol of S_{even}, $P'[j]$. There is at least one odd symbol (possibly more) separating these two symbols

We notionally relabel Q to the identity permutation, and apply the same relabelling to P to get P'.

$$\boldsymbol{Q} \quad 5\ 8\ 3\ 1\ 2\ 7\ 6\ 4 \longrightarrow 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8$$

$$\boldsymbol{P} \quad 5\ 2\ 3\ 4\ 1\ 7\ 6\ 8 \longrightarrow 1\ 5\ 3\ 8\ 4\ 6\ 7\ 2$$

We can then separate P' into two interleaved sequences of odd and even locations, and mark the breakpoints between adjacent elements:



The longest increasing subsequence of $odd(P')$ is the whole sequence, 1 3 4 7. We can attempt to extend the increasing subsequence by inserting members of $even(P')$: between 1 and 3 there is 5, which cannot extend the subsequence, and there is a breakpoint between 5 and 3; likewise, 8 cannot go between 3 and 4, and there is a breakpoint between 8 and 4. 6 can go between 4 and 7, but 2 cannot go after 7, and there is a breakpoint between 7 and 2. Hence $LIS(P') \geq 2LIS(odd(P')) - b(P')$ since $LIS(P') = 5$, $LIS(odd(P')) = 4$ and $b(P') = 3$. Similarly, $LIS(even(P')) = 2$, so $LIS(P') \geq 2LIS(even(P')) - b(P')$. Together, $LIS(P') \geq LIS(odd(P')) + LIS(even(P')) - b(P')$.

Figure 3.3: Analysing the Longest Increasing Sequence

when they occur in P'. Now, either all odd symbols that occur at locations between i and j have values between $P'[i]$ and $P'[j]$, in which case we could extend the increasing sequence S_{even} by including at least one of these symbols; or else there is at least one symbol which is less than $P'[i]$ or greater than $P'[j]$ — in which case, then there is a contribution of at least one to $b(P')$ involving this intervening symbol. This allows us to conclude that from the increasing sequence S_{even}, then we can form an increasing sequence of length at least $2LIS(even(P')) - b(P')$, as there are $LIS(even(P')) - 1$ consecutive pairs of symbols from S_{even}, and in addition we can also consider the sequence before the first symbol of S_{even}. Similarly, from S_{odd}, we can find an increasing sequence of length at least $2LIS(odd(P')) - 1 - b(P')$. Further, depending on whether $|P'|$ is odd or even, we can always increase one of these bounds by 1, by considering the effect of the last member of S_{odd} and the subsequent even symbols if $|P'|$ is even, or the effect with the last of S_{even} and subsequent odd symbols if $|P'|$ is odd. We know that each of these generated increasing sequences of P' is of length at most $LIS(P')$ by definition of $LIS(P')$. Summing these, we find that $2LIS(odd(P')) + 2LIS(even(P')) - 2b(P') \leq 2LIS(P')$. This process is illustrated in Figure 3.3. □

We now introduce some new terminology to assist in the proof.

Definition 3.2.12 *$P'_{x,y}$ is the subsequence of P' given by $P'[1.2^x - y]P'[2.2^x - y] \ldots P'[w2^x - y] \ldots P'[2^x\lfloor(|P| + y)/2^x\rfloor - y]$*

Observe that $P'_{0,0} = P'$ while $odd(P') = P'_{1,1}$ and $even(P') = P'_{1,0}$. We now make use of the following lemma:

Lemma 3.2.4 *$LIS(P'_{x,y}) \geq LIS(P'_{x+1,y}) + LIS(P'_{x+1,y+2^x}) - b(P'_{x,y})$*

Proof. Observe that $P'_{x,y}$ is the sequence $P'[w2^x - y]$ for integer $w = 1, 2, 3 \ldots$. So $odd(P'_{x,y})$ consists of those elements of $P'_{x,y}$ indexed by an odd value of w. Hence $odd(P'_{x,y}) = P'[(2v-1)2^x - y] = P'[v2^{x+1} - y - 2^x]$ for integer $v = 1, 2, 3 \ldots$, which defines $P'_{x+1,y+2^x}$. Similarly, $even(P'_{x,y})$ consists of $P'[(2v)2^x - y] = P'[2^{x+1} - y]$, which defines $P'_{x+1,y}$. The proof follows by taking Lemma 3.2.3 and substituting $P'_{x,y}$ for the sequence P', and using the two above observations on $even()$ and $odd()$. □

Lemma 3.2.5

$$\sum_{x=0}^{\lceil \log |P| \rceil} \left(\sum_{y=0}^{2^x - 1} b(P'_{x,y}) \right) = D(P, Q)$$

Proof.

$$\begin{aligned}
\textstyle\sum_x \sum_y b(P'_{x,y}) &= \textstyle\sum_x \sum_{y=0}^{2^x-1} |\{w | P'[w2^x + y] > P'[(w+1)2^x + y]\}| \\
&= \textstyle\sum_x |\{z | P'[z] > P'[z + 2^x]\}| \\
&= \textstyle\sum_x |\{z | Q^{-1}[P[z]] > Q^{-1}[P[z + 2^x]]\}| \\
&= \textstyle\sum_x |\{(i,j) | (Q^{-1}[i] > Q^{-1}[j]) \wedge (P[z] = i) \wedge (P[z + 2^x] = j)\}| \\
&= \textstyle\sum_x |\{(i,j) | (P^{-1}[i] + 2^x = P^{-1}[j]) \wedge (Q^{-1}[i] > Q^{-1}[j])\}| \\
&= \textstyle\sum_x \sum_{i,j} A_x(P)[i,j] \times Q[i,j] \\
&= \textstyle\sum_{i,j} A(P)[i,j] \times B(Q)[i,j] \\
&= D(P, Q)
\end{aligned}$$

□

To complete the proof, we apply Lemma 3.2.4 repeatedly to $P' = P'_{0,0}$, and the sequences generated thereby until we can split the sequences $P'_{x,y}$ no further. After the last split, all that remains are $|P'| = |P|$ single symbols, which each constitute a trivial increasing subsequence of length one. Telescoping the inequality of Lemma 3.2.3, we find that

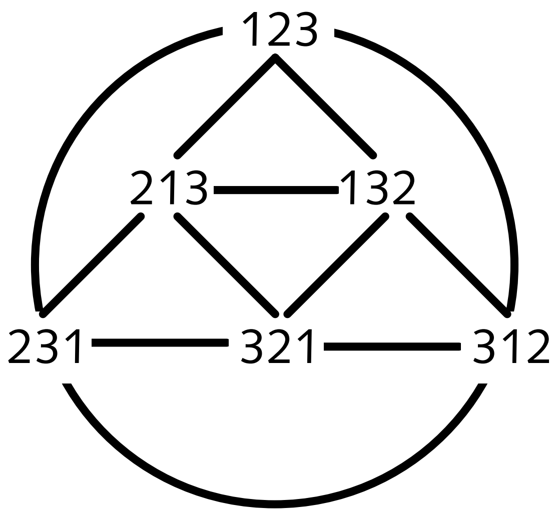
$$\begin{aligned}
LIS(P'_{0,0}) &\geq LIS(P'_{1,0}) + LIS(P'_{1,1}) - b(P'_{0,0}) \\
&\geq LIS(P'_{2,0}) + LIS(P'_{2,2}) - b(P'_{1,0}) + LIS(P'_{2,1}) + LIS(P'_{2,3}) - b(P'_{1,1}) - b(P'_{0,0}) \\
&\ldots \\
&\geq \textstyle\sum_{w=0}^{|P|-1} LIS(P'_{\lceil \log |P| \rceil, w}) - \sum_{i=0}^{\lceil \log |P| \rceil} \sum_{j=1}^{2^i - 1} b(P'_{i,j}) \\
&= \textstyle\sum_{w=0}^{|P|-1} LIS(P'[w+1]) - D(P, Q) \\
&= |P| - D(P, Q)
\end{aligned}$$

Hence we conclude that $LCS(P, Q) = LIS(P') \geq |P| - D(P, Q)$. Rearranging and substituting, we find $D(P, Q) \geq n - LCS(P, Q) = d(P, Q)$, as required. □

We observe that these bounds can be tight: consider the distance between the permutation P defined by $P[2i] = 2i - 1, P[2i - 1] = 2i$; and the identity permutation I. Then we have $D(P, I) = d(P, I) = |P|/2$, achieving the lower bound. On the other hand, take Q defined by $Q[|Q|] = 1$ and $Q[i] = i + 1$ for all $1 \leq i < |Q|$. The $d(Q, I) = 1$ but $D(Q, I) = \lceil \log |Q| \rceil$. This achieves the upper bound.

3.2.5 Hardness of Estimating Permutation Distances

For permutation edit distance, we have seen how to transform this problem into one of finding intersection size. Although we can make sketches for the intersection size as described in Section 2.3.3, these are only additive approximations, and so are only accurate when the intersection size is relatively large. Ideally, we should instead like to embed into a space like the Hamming space, where we know we can make better approximations irrespective of the size of the intersection. Figure 3.4 gives an exact



P	123	132	213	231	312	321
f(P)	[0,0,0,0]	[0,0,1,1]	[1,0,0,1]	[1,1,0,0]	[0,1,1,0]	[1,1,1,1]

The top graph illustrates the structure of the metric space, by drawing the graph of permutations at edit distance one from each other. Below it is a table showing an isometric embedding of permutations into the Hamming space.

Figure 3.4: An exact embedding of Permutation Edit Distance into Hamming distance

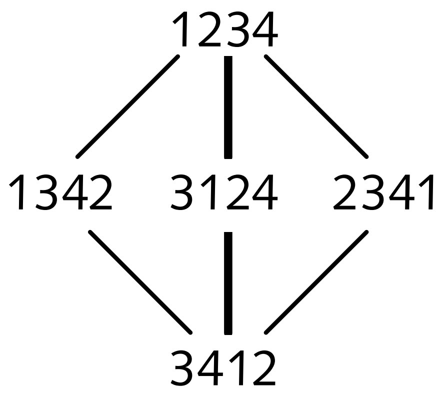


Figure 3.5: A set of permutations that forms $K_{2,3}$ under permutation edit distance

embedding of Permutation Edit Distance on permutations of size 3 into the Hamming distance. Here, $||f(P) - f(Q)||_H = 2d(P, Q)$. However, for permutations of length four or more, we can prove a lower bound on the distortion of any embedding of this metric into L_1 space.

Theorem 3.2.6 *Any embedding of permutation distance on permutations of length 4 or more into L_1 will have a distortion of at least $4/3$.*

Proof. We consider "forbidden subgraphs" which are difficult to embed into the target space. The graph $K_{2,3}$ — the complete bipartite graph between one set of two nodes and another of three nodes — cannot be embedded into L_1 space with distortion better than $4/3$ [Mat02]. Figure 3.5 shows such a subgraph of permutation edit space: therefore for permutations of length four, a distortion of $4/3$ is unavoidable. For longer permutations of length n, we can use the same example and simply append $5 \dots n$ to the end of each permutation shown. Therefore the lower bound applies to all permutations of length ≥ 4. □

Note that the Hamming distance is a subspace of L_1 — that is, if we consider the L_1 distance restricted to considering only binary vectors, then this is exactly the Hamming distance. So this bound also applies to embeddings into Hamming space, since it applies to any embedding into L_1 space, which includes all embeddings into Hamming space. This lower bound does not apply to Transposition distance — although each permutation edit operation is a transposition, when we look at the permutations 1 2 3 4 and 3 4 1 2, we see that their transposition distance is 1, and so the example given does not form $K_{2,3}$ under transposition distance.

Similarly for Euclidean distance, the star graph $K_{1,3}$ cannot be embedded into Euclidean space with distortion better than $\sqrt{4/3}$.[4] This can give a lower bound for embeddings of Permutation Edit Distance, Transposition Distance and Reversal distance of $\sqrt{4/3}$. However, this is less interesting, since our earlier embeddings were all into Hamming space and L_1 space, which are not affected by this bound.

[4]This observation is folklore, but follows from simple geometry. Let the three points in Euclidean space representing the degree one nodes be a, b and c. We will consider d, the minimum (Euclidean) distance between any pair of a, b, c. Let the maximum distance from the degree three node to the each of a, b and c be m. Then $d^2 \leq \frac{1}{3}((a-b)\cdot(a-b)+(b-c)\cdot(b-c)+(a-c)\cdot(a-c)) \leq \frac{1}{3}(2(a^2+b^2+c^2)-2(a\cdot b+b\cdot c+a\cdot c)) \leq \frac{1}{3}(3(a^2+b^2+c^2)-(a+b+c)\cdot(a+b+c)) \leq \frac{1}{3}3(a^2+b^2+c^2) \leq 3m^2$. Hence $d/m \leq \sqrt{3}$; however, in the graph this ratio is 2. Therefore, the distortion is lower bounded by $2/\sqrt{3} = \sqrt{4/3}$.

	$i_a\, i_b\, i_c$	$i_a\, i_c\, i_b$
$i_c\, i_b\, i_a$	2	1
$i_b\, i_a\, i_c$	1	1

Figure 3.6: Permutation Edit Distance between the two pairs of possibilities

These lower bounds on embedding Permutation Edit distance into sketchable spaces do not rule out the possibility that there is an alternative way to make sketches for this metric. However we will now give some further evidence which suggests that this will not be easy. We show that estimating this distance in the same communication setting is provably hard, by doing the reverse embedding of intersection size into permutation distance.

Theorem 3.2.7 *Estimating the permutation edit distance with any constant probability* $1-\delta$ *requires* $\Omega(n)$ *bits of communication.*

Proof. We shall show that if we could estimate the permutation edit distance in a communication complexity setting then we could estimate the intersection size with the same probability. Let A and B each hold a bit-string, $\boldsymbol{a}$ and $\boldsymbol{b}$ respectively, of length n. They wish to communicate in order to discover the value of $|\boldsymbol{a} \wedge \boldsymbol{b}|$ with some constant probability $1-\delta$. We use the following transformation into permutation edit distance: A considers each $i = 1, 2, \ldots n$ in order, and begins to construct an output permutation, which is initially empty. If $\boldsymbol{a}_i = 1$ then A appends the sequence $i_a\ i_b\ i_c$ to the permutation; otherwise, A appends $i_a\ i_c\ i_b$. B performs a similar transformation as follows: for each $i \leq n$, if $\boldsymbol{b}_i = 1$ then B appends $i_c\ i_b\ i_a$, else B appends $i_b\ i_a\ i_c$. If we consider the permutation edit distance between these generated permutations, we observe that for each i, there is a contribution of 2 to the permutation edit distance if $\boldsymbol{a}_i = 1 \wedge \boldsymbol{b}_i = 1$, and 1 otherwise. This is laid out in Figure 3.6. Hence the permutation edit distance is $n + i(\boldsymbol{a}, \boldsymbol{b})$, and so if any communication scheme could estimate the permutation edit distance with constant probability, then it could also estimate the intersection size with constant probability. Because Intersection Size has been shown to require $\Omega(n)$ bits of communication in the randomized communication complexity model (see Section 2.3.3 for details), it follows that permutation edit distance on permutations of length n requires $\Omega(n/3) = \Omega(n)$ bits also. $\square$

However, this is only a negative result for *estimating* the permutation edit distance; it does not extend to showing that it is hard to approximate it (see Section 2.1 for the distinction between an estimation and an approximation). This gap leaves scope either for a way to approximate the permutation edit distance, or an improved hardness proof that shows there can be no such approximation. The next result is a much stronger negative result for the dual problem to Permutation Edit Distance, that of longest common subsequence.

Theorem 3.2.8 *Approximating the Longest Common Subsequence of two sequences drawn from a universe of size n with no repetitions in a communication complexity model requires* $\Omega(n)$ *bits of communication.*

Proof. We show by contradiction that such a scheme would imply the existence of an efficient scheme to approximate intersection size. Consider two subsets of $\{1 \ldots n\}$, A and B, and form a sequence from each as the elements of A (and B) in sorted order. The longest common subsequence of these sequences is exactly the intersection size, $|A \cap B|$. Since finding intersection size requires $\Omega(n)$ bits of communication (Section 2.3.3) in the probabilistic model, then the longest common subsequence must also require this much communication. $\square$

Corollary 3.2.2 *Approximating the Longest Common Subsequence of two strings on an alphabet σ requires $\Omega(|\sigma|)$ bits of communication.*

The corollary follows immediately by considering a string that is a permutation of a subset of σ and using the same proof above. This corollary is weaker than the original theorem, since for strings of length $O(n)$ drawn from a constant alphabet, the lower bound is insignificant. The main theorem says something interesting: that there is no hope for sketching to find the Longest Common Subsequence of sequences, and consequentially it is unlikely that there are Approximate Nearest Neighbors and related geometric problems based on the Longest Common Subsequence measure that are more efficient than the naïve solutions based on scanning the entire collection of sequences.

3.2.6 Extensions

These embedding techniques can also be adapted for a large range of permutation distances. Thus far we have limited ourselves to strict permutations. We now consider extensions of the above embeddings for cases where we relax these requirements and mix operations: we consider the effect of allowing reversals, transpositions and edits together; we also either allow the set of symbols in each sequence to be non-identical, or additionally allow one sequence to contain repetitions, that is to be a string. To be precise, we describe the following extensions to the embeddings of Section 3.2. Each can be described in terms of the permitted operations, and the domain of the inputs:

1. Reversals, Transpositions and Edits (this distance is denoted by τ)
2. Reversals, transpositions, edits and indels between permutation pairs (distance denoted by τ');
3. Reversals, transpositions, edits and indels between a permutation and a string (τ'');
4. Reversals or transpositions with indels (distance denoted r' and t')
5. Reversals or transpositions between a permutation and a string (r'' and t'').

There are many other possible combinations of operations that can be embedded into vector distances: these have very similar embeddings and are omitted.

These extensions are for the most part fairly straightforward and build on results that have already been seen. Moreover, they had a certain soporific effect on readers of earlier versions of this work. To maintain the reader's interest, these extensions have therefore been moved to an appendix (Appendix A), where they can be enjoyed at leisure. Some of the results are used in later sections, so a table summarises all the embeddings of permutation distances covered here. These are recorded in Figure 3.7, the important points being the space embedded into and the approximation factor.

3.3 Applications of the Embeddings

We have built up quite a repertoire of embeddings now. We can immediately find algorithmic applications of our embeddings. On the whole, these rely on the results for the spaces described in Chapter 2.

3.3.1 Sketching for Permutation Distances

The notion of a short sketch to approximate distances with is described in Definition 2.1.3. We shall now show how to sketch for the notions of permutation distance.

Editing Operations	Denoted	Between	Approx	Embedding	Section
Swap	swap	Perm-Perm	1	Hamming	3.2.1
Swap	swap'	String-String	1	Hamming	3.2.1
Reversal	r	Perm-Perm	2	Hamming	3.2.2
Reversal	r'	Seq-Seq	2	Hamming	A.1.2
Reversal	r''	Seq-String	2	L_1	A.1.2
Transposition	t	Perm-Perm	2	Hamming	3.2.3
Transposition	t'	Seq-Seq	2	Hamming	A.1.2
Transposition	t''	Seq-String	2	L_1	A.1.2
Reversal, Transpositions, Edits	τ	Perm-Perm	3	Hamming	A.1.1
Reversals, Indels, Transpositions, Edits	τ'	Seq-Seq	3	Hamming	A.1.2
Reversals, Indels, Transpositions, Edits	τ''	Seq-String	3	L_1	A.1.2
Permutation Edit Distance	d	Perm-Perm	$\log n$	Intersection	3.2.4

The different distances, their symbol, the objects that they relate, the approximation factor and the vector space the embedding is into. Perm-perm means that the the distance is only defined between mutual permutations; seq-seq means that the sequences are drawn from a universe of size ℓ with no repetitions, and seq-string means that one is a sequence and the other is a string.

Figure 3.7: Summary of the different sequence distance measures

Reversal and Transposition Distance

Theorem 3.3.1 *Sketches for the Transposition distance or Reversal distance can be computed using $O(\frac{1}{\epsilon^2}\log(1/\delta)\log n)$ bits of storage such that the Reversal distance (respectively Transposition distance) can be estimated from a pair of sketches, accurate up to a factor of $2+\epsilon$ with probability $1-\delta$.*

Proof. Because we have transformations into the Hamming distance, we just have to take the bit-vector representation and apply the technique given in the proof of Theorem 2.3.1. We can then find a 2-approximation with a factor of $1+\epsilon$. Multiplying through, and rescaling ϵ allows us to make sketches which allow the distance to be found to a $2+\epsilon$ factor. □

These sketches can be made in the ordered streaming model: note that our sketching method assumes that any non-specified value is zero. Since each one bit in the matrices comes from information about adjacent pairs in the permutation, we can parse the permutation as a stream of tuples, so $\dots i, j, k \dots$ is viewed as $\dots (i,j),(j,k)\dots$. Although the matrix space is $O(n^2)$, the size of the synopsis will of course depend only on $\log n$. The processing time is dependent only on the linear number of non-zero entries in our matrices, rather than the total quadratic number of entries. These same results also apply to any of the embeddings listed in Figure 3.7 that embed into spaces that we can sketch (Hamming or L_1). If the approximation in the embedding is c, then we can sketch these distances accurately up to an approximation factor of $c(1+\epsilon)$.

Permutation Edit Distance

We have seen that there are various hardness results in the communication complexity model for Permutation Edit Distance. The consequence of these results is that sketching in the sense of Definition 2.1.3 is not possible. Nevertheless, we can still make some weak 'guesses' of the distance (these are not strong enough results to be called estimates or approximations).

Theorem 3.3.2 *Rough sketches for the Permutation Edit Distance can be computed of size $O(\log n)$ such that the expectation of the sketched distance is a $\log 2n$ approximation of the edit distance.*

Proof. We take advantage of the embeddings in Section 3.2.4 of Permutation Edit Distance using matrices $A(P)$ and $B(Q)$. These are into the intersection size. We use one of the methods outlined in Section 2.3.3 to roughly sketch the intersection size of the matrices $A(P)$ and $B(Q)$, which gives a $\log n$ approximation of the edit distance.

We shall focus on using the Jaccard coefficient of set resemblance, since this gives a method that scales well to different set sizes. This tries to approximate $r = \frac{|A\cap B|}{|A\cup B|}$, whereas we are interested in $|A\cap B|$. However, in Section 3.2.4 we saw that A and B are of fixed weight, and that $|A| = n\lceil\log_2 n\rceil - 2^{\lceil\log_2 n\rceil} + 1$ and $|B| = n^2/2 + n/2$. Consequently, $n^2/2 \le |A\cup B| \le n^2/2 + n\lceil\log n\rceil$. We are interested in the value of $|A\cap B| = r|A\cup B|$. So if we use $rn^2/2$ as the approximation of $|A\cap B|$, then $rn^2/2 \le |A\cap B| \le r(n^2/2 + n\lceil\log n\rceil)$ This is a $r(n^2/2+n\lceil\log n\rceil)/(rn^2/2) = 1 + \frac{2}{n}\lceil\log n\rceil$ approximation. Since $|A\cap B|$ is a $\lceil\log n\rceil$ approximation for the edit distance, this is consequently a $\lceil\log n\rceil + \frac{2}{n}\lceil\log n\rceil^2$ approximation for the edit distance. For $n \ge 98$, then $\frac{2}{n}\lceil\log n\rceil^2 \le 1$, hence this is a $\log 2n$-approximation for the permutation edit distance. The overall size of this rough sketch is the $O(\log n)$ bits for the set resemblance sketch. □

3.3.2 Approximating Pairwise Distances

The embeddings allow distance approximations to be made efficiently in a communication setting. We have the following scenario: there are two communicants, A and B, who each hold a permutation of $\{1\dots n\}$, P and Q respectively, and they wish to communicate in such a way as to calculate the approximate distance between their permutations.

Theorem 3.3.3 *There is a single round communication protocol to allow reversal or transposition distance approximation up to a factor of $2 + \epsilon$ with a message of size $O(\frac{1}{\epsilon^2} \log(1/\delta) \log n)$ bits. The protocol succeeds with probability $1 - \delta$.*

Proof. This follows immediately from the fact that we can sketch these distances. Any sketch can be sent as a message, and the recipient can make a sketch of their own sequence, and compare the two to find a distance approximation. Each sketch is a vector of dimension $O(1/\epsilon^2 \log 1/\delta)$ and each entry in the sketch requires $O(\log n)$ bits to represent it. We also need to send the random bits used to construct the sketch: the size of these is also $O(1/\epsilon^2 \log 1/\delta \log n)$ bits. See Sections 2.2.2 and 2.2.3 for more details. □

Now suppose that we have a number of permutations, and we wish to be able to rapidly find the approximate distance between any pair of them. Traditional methods would suggest that for each pair, we should take one and relabel it as the identity permutation, and then solve the sorting by reversals or sorting by transpositions problem for the correspondingly relabelled permutation. We show that, given a near linear amount of pre-processing, this problem can be solved exponentially faster.

Corollary 3.3.1 *With a near linear amount of pre-processing of m permutations of length n, the Reversal distance (or Transposition distance) between any pair can be approximated in time $O(\frac{1}{\epsilon^2} \log n \log m)$. This is a $2 + \epsilon$ approximation with constant probability.*

This is achieved by computing a sketch for each sequence in advance. Any pairwise distance can be approximated by comparing the sketches of the pair of interest in time linear in the size of the sketches As in the proof of Lemma 2.2.1, we must set $\delta = o(m^{-2})$ to ensure that over all $O(m^2)$ possible comparisons the probability of any one failing is at most constant. Again, these results apply to any of the embeddings listed in Figure 3.7 that can be sketched, with corresponding approximation factors.

A similar result holds for Permutation Edit Distance using the rough sketches described above. Since the embedding is non-symmetric, we can create sketches for both $B(Q)$ and $A(P)$. However, we cannot make the same guarantees of accuracy, so these results are less interesting for us.

3.3.3 Approximate Nearest Neighbors and Clustering

We first saw problems of Approximate Nearest Neighbors and Clustering defined in Section 2.4. They were described in terms of a vector distance and a set of vectors. We now talk about the same problems but defined for a permutation distance and a set of permutations, and we can take Definition 1.5.2 and consider it in this setting. The problem is to pre-process a collection of permutations so that given a new query permutation, the closest permutation from the collection can be found. The crux here is to avoid the dimensionality curse: that is, design a polynomial space data structure that answers queries in time polynomial in the query and sublinear in the collection size. Such questions may be of interest in the context of processing large number of (biological) sequences in order to find patterns of similarity between them, and given new sequences (representing, for example, a new virus) to quickly be able to find the sequence in the database which is most similar. These are also of interest from a more general point of view, in order to discover what can and cannot be accomplished for a variety of distances, not just the vector distances that have been the topic of most study.

Theorem 3.3.4 *We can find approximate nearest neighbors under Reversal distance (respectively Transposition distance and compound distances thereof) up to a factor of $2 + \epsilon$ with query time $O(\ell \log \ell \cdot m^{1/(1+\epsilon)})$, where m is the number of sequences in the database, and ℓ the size of the universe from which sequence symbols are drawn.*

Proof. This follows immediately by adapting the method for Approximate Nearest Neighbors described in Section 2.4.1. Some care is needed, since for efficiency we need to ensure that the sampling at the root of the Locality Sensitive Hash functions used therein does not attempt to sample directly from the quadratic ($O(\ell^2)$) space of the matrices of the embeddings. Recall how the procedure based on Locality Sensitive Hash functions works: firstly, a number of bit locations are queried to create a new bit-string. This bit-string is then hashed into a hash table. We consider an alternative method to reach the same result: in outline, we will invert the sampling mechanism, so given a location, we can find out whether that location is included in the set of those locations that are sampled. We can then create the hash value by applying the hash function for this location. By using the linear fingerprint function described in Section 2.1.3, then this will be easy to achieve: we have to add on the appropriate amount to the running total to create the Locality Sensitive Hash value.

We now flesh out the details of this approach, by considering the binary matrices described in Section 3.2: these have size $O(\ell^2)$ when we allow each symbol of our sequences to be drawn from a universe of size $O(\ell)$. However, we only need to consider the $O(\ell)$ locations where there are bits set to 1 in these matrices: all other values are 0. From these, we can compute the value of the hash function, since the 0 values add nothing to the value of the hash function and so can be omitted. We can process these 1 values efficiently, since they are generated by adjacent pairs in the permutations. Hence with a linear pass over the permutation, we can evaluate the hash function, provided we can perform the "is this location sampled" test efficiently. Recall the structure of the hash functions that we wish to compute from Section 2.4.1, for the Hamming distance: each hash function is made by picking k locations uniformly at random with replacement from the bit-string, and this is done independently l times to generate l different hash functions. If we specify each hash function as a bit-matrix by picking k locations uniformly at random and setting these to 1, then we can compute the hash efficiently: for each pair of adjacent values in the permutation, we look up this pair in each of the bit-matrices. If the ith matrix has a 1 in this location, then we are to count this pair towards the ith hash function. The overall time cost for evaluating all these hash functions is then $O(l \cdot \ell)$, which gives a cost for finding approximate nearest neighbors in time $O(\ell \log \ell m^{1/1+\epsilon})$, which is no more costly than the original protocol for Hamming distance. □

Theorem 3.3.5 *$2\log 2n$-Approximate Furthest Neighbors can be found under Permutation Edit Distance in expected time $O(m^{1-1/(2\ln n)} \ln m + n)$ per query with polynomial time pre-processing.*

Proof. We can use the procedure for Approximate Furthest Neighbors using Locality Sensitive Hash functions described in Section 2.4.1. We can now apply some specifics of the transformation into intersection distance to show that this procedure is tractable. Recall that the parameter $r = \frac{|A(P)\cap B(Q)|}{|A(P)\cup B(Q)|}$ is set by a binary search looking for a distance r at which we have an approximate neighbor. The range of distance values is discrete, because we are looking at the intersection size of two matrices, and so the permitted values will be polynomial in n. If $r = 0$ then $|A(P) \cap B(Q)| = 0$ and so $P = Q$. This means that if we are ever testing the case $r = 0$, we can deal with this separately by making fingerprints of each sequence and building these into a hash table. If we are testing the case $r = 0$ for a query permutation Q, we first look the hash of Q up in the hash table. If it collides with the hash of any other permutation(s), then we compare the query to the permutation(s) to see if we have an exact match, and report this. Otherwise, we have a lower bound on r, $r \geq 1/n^2$, since we are comparing two matrices of size n^2, and we know there is at least one location where they differ, as they are not identical. We know that there are $O(m^{1-2\epsilon/3\ln 1/r} \log n \ln m/ \ln 1/r)$ hash function evaluations per query. By the construction of $A(P)$ and $B(Q)$, then $\frac{|A(P)\cap B(Q)|}{|A(P)\cup B(Q)|} \leq \frac{2\log n}{n}$. Putting in these bounds on r, this is $O(m^{1-\epsilon/3\ln n} \log n \ln m/(\ln(2\log n) - \ln n))$ which is $O(m^{1-\epsilon/3\ln n} \ln m)$ hash function evaluations. We now have to consider the cost of evaluating each hash function. Given a random permutation R we have to find $\min R(A(P))$. Let us assume that in the pre-processing stage, we apply these permutations to the transforms, $A(P)$ for every given permutation P. Then, at the query time we have to find $\min R(B(Q))$. Recall that $B(Q)$ is (conceptually) a binary matrix where half the bits are 0 and half are 1. We therefore expect that if R is chosen at random, then finding this function takes $O(1)$ probes of this matrix. Note that we would not actually construct the matrix $B(Q)$, but rather apply R to $B(Q)$ by testing the relative position of pairs in Q as listed by R. We do not have to construct and store a huge quantity of random permutations R in advance. These can be made randomly when required. Since we use R to find $\min(R(B(Q)))$ and $\min(R(A(P)))$, then for any R we expect to need the first $O(\log n)$ values from R over its whole life. So we can store a prefix of R and extend this if necessary, or else created R pseudo-randomly on the fly based on a small amount of randomness. Since the Jaccard coefficient of set resemblance gives a $\log 2n$ approximation, we may as well set ϵ to some constant value, say 1. Since for any query permutation Q, we need to construct Q^{-1}, then this gives the stated time bounds. □

Theorem 3.3.6 *A set of k centers can be found under Reversal distance (respectively, Transposition distance), that is guaranteed to be within a factor of $4+\epsilon$ of the optimum clustering.*

Proof. Again, we adapt existing results for the Hamming space given in Section 2.4.3. For each permutation we can compute a sketch that will allow rapid approximation of the distance. We can then use Algorithm 2.4.1 on these sketches to find a set of centers. By Theorem 2.4.3, the approximation is of the stated quality. The time of this algorithm is $O(mn \log m)$ to make the sketches and $O(km \log m)$ to find the clustering, totalling $O(m \log m(n+k))$. Note that if we used a pairwise comparison method, then each comparison would take time at least $\Theta(n)$, and the time for this clustering would be $\Theta(kmn)$. Hence this method is superior for large enough k and n. □

3.3.4 Approximate Pattern Matching with Permutations

We shall show how properties of the embeddings into vector distances can be harnessed to give solutions to the problem of Approximate Pattern Matching for some of the permutation distances of interest. As described in Section 1.5.2, we want to compute for each i the value of $D[i]$, the cost of aligning the pattern permutation against a text at position i. Since some of the distances we are considering are NP-hard to find, it would be unreasonable to ask for the exact value of $D[i]$, and so we shall instead find approximations $\hat{D}[i]$. Directly applying existing distance computations or naïvely using the transformations of our distances would be expensive; we take advantage of the fact that because the embeddings are based on pairwise comparisons, the approximate cost can be calculated incrementally with only a small amount of work.

For generality, we shall insist that the pattern, P is a permutation, but we shall allow the text to be a string (that is, it can contain more than one occurrence of any element). We shall make use of the Pruning Lemma (Lemma 1.5.1). Because the distances we consider can change the sequence length by one per operation, we can apply this lemma. So we only need to consider the alignment of the pattern P of length m against each subsequence of the text of length m to get a 2-approximation to the optimal alignment.

Theorem 3.3.7
(i) Approximate pattern matching for reversal distance can be solved in time and space $O(n+m)$; each $D[i]$ is approximated to a factor of 5.
(ii) Approximate pattern matching for transposition distance can be solved in time and space $O(n+m)$; each $D[i]$ is approximated to a factor of 4.

Proof. We must allow insertions and deletions to our sequences since in this scenario we cannot insist that we will always find exact permutations at each alignment location. Therefore, we shall use the results of the extended embeddings given in Section A.1.2 using R'' and T''. It is important to note that although these embeddings and this proof are described in terms of quadratic sized matrices, we do not construct these matrices, but instead concentrate only on the linear number of non-zero entries in these figurative matrices. We shall prove both claims together, since we take advantage of common properties of the embeddings.

Suppose we know the cost of aligning $T[i \dots i+m-1]$ against P, and we now want to find the cost for $T[i+1 \dots i+m]$. This is equivalent to adding a character to one end of $T[i \dots i+m-1]$ and removing one from the other. So only two adjacencies are affected — at the start and end of the subsequence. This affects only a constant number of symbol pairs in our matrices. Consequently, we need only perform a constant amount of work to update our count of the number of transposition or reversal breakpoints, provided we have pre-computed the inverse of the pattern P in time $O(m)$. To begin the process, we imagine aligning P with a location to the left of T so that there is no overlap of pattern and text. Initially, the distance between P and $T[-m \dots 0]$ is defined as m. From this position, we can advance the pattern by one location at a time and do a constant amount of work to update the count. The total time required is $O(n+m)$. Some subtlety is needed to design a data structure to support this counting, but recall that our pattern P is a fixed sequence with no repetitions. This can be stored as a vector, with the adjacency information. We have to record how many copies of adjacent pairs have been seen in the current part of the text being analysed, these are either pairs that are in P and can be recorded in P's data structure, or classified as "not in P" and counted. □

Theorem 3.3.8 *Approximate Permutation Matching can be solved for Permutation Edit Distance in time $O(n \log m)$, approximating each $D[i]$ up to a factor of $2 \log m$.*

Proof. We make use of the extended transformation in Section 3.2.6 for permutation edit distance, and so our result will be accurate up to a factor of $\log m$ for each alignment; the pruning lemma then tells us that in turn these are a 2-approximation of the optimal alignment. We can use the trick of relabelling P as $1\ 2 \dots m$, and relabelling T accordingly as we go along. Suppose we have found the cost of matching $T[i \dots i+m-1]$ against P. We can advance this match by one location to the right by comparing $T[i+m]$ with the $\log m$ locations $T[i+m-1]$, $T[i+m-2]$, $T[i+m-4] \dots$. Each pair of the form $T[i+m-2^k] > T[i+m]$ that we find adds one to our total. At the same time, we maintain a record of the L_1 difference between the number of symbols in P missing from $T[i \dots i+m-1]$ (since each of these must participate in an insertion operation to transform $T[i \dots i+m-1]$ into P). This can be updated in constant time using $O(|P|)$ space. We can step the left end of a match by one symbol in constant time if we also keep a record for each $T[i]$ of how many comparisons it caused to fail from symbols to the right — we reduce the count by this much to find the cost of $T[i+1 \dots i+m]$ from $T[i \dots i+m]$. In total we do $O(\log m)$ work per step, giving a total running time of $O(n \log m)$. □

Of course, it may well be the case that the best alignment of P against T at location i would choose a substring whose length was not exactly m. This method is not conducive to finding the "optimal" alignment; however, note that moving the left end of the alignment has cost only $O(1)$ per move. It is therefore possible to perform a small local search in an $O(\log m)$ sized neighbourhood around the left hand end of the alignment to find better (approximate) local alignments without affecting the asymptotic time bounds on this procedure.

3.4 Discussion

Motivated by biological scenarios, we have studied problems of computing distances between permutations as well as matching permutations in sequences, and solving permutation proximity problems. In summary, the main points of this chapter have been:

- The embeddings of permutation distances into well known spaces, such as Hamming space, L_1 or the Set Intersection space using matrices that capture the relative layout of symbols in the permutation. These embeddings are approximately distance-preserving, that is, they preserve original distances up to a small constant or logarithmic factor.

- Using these embeddings, we gained several results, including communication complexity protocols to estimate the permutation distances accurately; efficient solutions for approximately answering nearest and furthest neighbor problems with permutations; and algorithms for finding permutation distances in the streaming model.

- We considered a class of permutation matching problems which are a subset of our approximate pattern matching problems. We saw linear or near-linear time algorithms for approximately solving permutation matching problems; in contrast, the corresponding string problems take significantly longer.

Permutation editing and matching is not only of interest for Computational Biology, but also as combinatorial problems of independent interest, and as a foundational mechanism towards understanding the fundamental complexity of editing and matching strings. In general we would expect that permutations are easier to deal with than strings: certainly they are no harder, since any string algorithm is applicable to permutations, whereas the converse is not true.

Next we go on to take this general idea of embedding sequence distances, and apply it to string distances. We have already seen the first string distances dealt with in this way: Hamming distance is one of our basic tools, and Swap distance proves to be easy to deal with for strings. The next string distances we shall consider will require more sophisticated techniques. Although we use the same general approach — transform a sequence into a bit-vector representation so that vector distances approximate the original string distance — the transformations now become much more involved. The dimensionality of the space being embedded into increases massively and so has to be handled with care. The quality of the approximations decreases significantly, going from small constant factors to growing (slowly) with the size of the object. We shall return to this discrepancy later.

Chapter 4

Strings and Substrings

Ages ago, Alex, Allen and Alva arrived at Antibes, and Alva allowing all, allowing anyone, against Alex's admonition, against Allen's angry assertion: another African amusement... anyhow, as all argued, an awesome African army assembled and arduously advanced against an African anthill, assiduously annihilating ant after ant, and afterward, Alex astonishingly accuses Albert as also accepting Africa's antipodal ant annexation. Albert argumentatively answers at another apartment. Answers: ants are Amesian. Ants are Amesian?

Africa again: Antelopes, alligators, ants and attractive Alva, are arousing all anglar Africans, also arousing author's analytically aggressive anticipations, again and again. Anyhow author apprehends Alva anatomically, affirmatively and also accurately.

[Abi74]

4.1 Introduction

In this chapter we will present embeddings of some of the string distances discussed in Chapter 1 into vector distances such as Hamming distance and L_1 distance. From these we are able to develop a number of solutions to geometric and pattern matching problems in the string edit distance spaces. We shall use the problem of approximate pattern matching under string distances as a motivating problem to guide the development of our solutions.

Approximate String Pattern Matching

In the Approximate String Pattern Matching problem studied in the Combinatorial Pattern Matching area, we are given a text string t of length n and a pattern string p of length $m < n$. The *approximate pattern matching problem under edit distance* is to compute the minimum string edit distance between p and any prefix of $t[i : n]$ for each i; we denote this distance by $D[i]$ as usual. It is well known that this problem can be solved in $O(mn)$ time using dynamic programming [Gus97]. The open problem is whether this quadratic bound can be improved substantially in the worst case.

There has been some progress on this open problem. Masek and Paterson [MP80] used the Four Russians method to improve the bound to $O(mn/\log m)$, which remains the best known bound in general to this date. Progress since then has been obtained by relaxing the problem in a number of ways.

- Restrict $D[i]$'s of interest.

  Specifically, the restricted goal is to only determine i's for which $D[i] < k$ for a given parameter k. By again adapting the dynamic programming approach a solution can be found in $O(kn)$ time and space in this case [LV86, Mye86]. An improvement was presented by Ṣahinalp and Vishkin [ṢV96] (since improved [CH98]) with an $O(n\,\mathrm{poly}(k/m))$ time algorithm which is significantly better. These algorithms still have running time of $\Omega(nm)$ in the worst case.

- Consider simpler string distances.

  If we restrict the string distances to exclude insertions and deletions, we obtain Hamming distance. Abrahamson [Abr87] gave a $O(n\,\text{poly-log}\,n\sqrt{m})$ time solution breaking the quadratic bound; since then, it has been improved to $O(n\,\text{poly-log}\,n\sqrt{k})$ [ALP00]. Karloff gave an $O(n\log^3 n)$ algorithm to approximate Hamming distances to a $1+\epsilon$ factor [Kar93]. This was tightened to $O(n\log n)$ by Indyk [Ind00]. Hamming distance results however sidestep the fundamental difficulty in the string edit distance problem, namely, the need to consider nontrivial *alignment* of the pattern against text when characters are inserted or deleted.

Results

In this chapter, we give a near linear time algorithm for solving the approximate pattern matching problem for a string distance. We consider the case where the edit distance in question is the string edit distance with moves. As usual, we will be approximating the distance rather than finding it exactly. In this case, the quality of the approximation depends on the length of the pattern sequence being considered. The resulting algorithm has to consider nontrivial alignment between the text and the pattern, and obtains a significantly subquadratic algorithm in the worst case.

The main pattern matching result is a deterministic algorithm for the string edit distance matching problem with moves which runs in time $O(n\log n)$. The output is the matching array D where each $D[i]$ is approximated to within a $O(\log n\log^* n)$ factor. The approach relies on an embedding of strings into vectors in the L_1 space. The L_1 distance between two such vectors is an $O(\log n\log^* n)$

approximation of the string edit distance with moves between the two original strings. This is a general approach, and can be used to solve many other questions of interest beyond the core approximate pattern matching problem. These include string similarity search problems such as indexing for string nearest neighbors, outliers, clustering and so on, under this metric of edit distance with moves. We are also able to vary the embedding and give results on related block edit distances such as the compression distance.

All of these results rely at the core on a few components. Foremost, is a parsing of strings into a hierarchy of substrings. This relies on deterministic coin tossing (also known as local symmetry breaking) that is a well known technique in parallel algorithms [CV86, GPS87, AM91] with applications to string algorithms [ṢV94, ṢV96, MSU97, CPṢV00, ABR00, MṢ00]. In its application to string matching, precise methods for obtaining hierarchical substrings differ from one application instance to another, and are fairly sophisticated: in some cases, they produce non-trees, in other cases trees of degree 4 or more etc. Inspired by these techniques is this simpler hierarchical parsing procedure called *Edit Sensitive Parsing* (ESP) that produces a tree of degree 3. Furthermore, it is computable in one pass in the data stream model. ESP should not be perceived to be a novel parsing technique, as it draws strongly on early works; however, it is an attempt to simplify the technical description of applying deterministic coin tossing to obtain hierarchical decomposition of strings.

4.2 Embedding String Edit Distance with Moves into L_1 Space

In the following sections, we describe how to embed strings into a vector space so that $d()$, the string edit distance with substring moves, will be approximated by vector distances. Consider any string a over an alphabet set σ. We will embed it as $V(a)$, a vector with an exponential number of dimensions, $O(|\sigma|^{|a|})$; however, the number of dimensions in which the vector is nonzero will be quite small, in fact, $O(|a|)$. This embedding V will be computable in near linear time, and it will have the approximation preserving property we seek.

At the high level, our approach will *parse* a into special substrings, and consider the multi-set $T(a)$ of all such substrings. The size of $T(a)$ will be at most $2|a|$. Then, $V(a)$ will be the "characteristic" vector for the multi-set $T(a)$, containing for each substring present in $T(a)$ the number of occurrences of that substring in the multi-set. The technical crux is the parsing of a into its special substrings to generate $T(a)$. This procedure is called *Edit Sensitive Parsing*, or ESP for short. In what follows, first ESP is described, and then the vector embedding is given and its approximation preserving properties are proven.

4.2.1 Edit Sensitive Parsing

We will build a parse tree, called the *ESP tree* (denoted $ET(a)$), for a string a: a will be parsed into hierarchical substrings corresponding to the nodes of $ET(a)$. The goal is that string edit operations only have a localised effect on the ET. An obvious parse tree will have strings of length 2^i, that is, $a[k2^i : ((k+1)2^i - 1)]$ for all integers k and i; this will yield a binary parse tree. But if a is edited by a single character insertion or deletion to obtain a', a and a' will get parsed by this approach into two different multi-sets of hierarchical substrings. These can be similar or very different depending on the location of the edit operation, and the resulting embedding will not be approximation preserving.

Given a string a, we now show how to hierarchically build its ESP tree in $O(\log |a|)$ iterations. Each iteration generates a new level of the tree, where each level contains between a half and a third of the number of nodes in the level from which it is derived. At each iteration i, we start with a string a_i and *partition* it into *blocks* of length 2 or 3. We replace each such block $a_i[j : k]$ by its *name*, $hash(a_i[j : k])$, where h is a one-to-one hash function on strings of length at most 3. Then a_{i+1} consists of the $hash()$

values for the blocks in the order in which they appear in a_i. So $|a_i|/3 \le |a_{i+1}| \le |a_i|/2$ and the height of the structure is $O(\log |a|)$. We assume $a_0 = a$, and the iterations continue until we are left with a string of length 1. The ESP tree of a consists of levels such that there is a node at level i for each of the blocks of a_{i-1}; their children are the nodes in level $i-1$ that correspond to the symbols in the block. Each character of $a_0 = a$ is a leaf node. We also denote by σ_0 the alphabet σ itself, and the set of names in a_i as σ_i, the alphabet at level i.

It remains for us to specify how to partition the string a_i at iteration i. This will be based on designating some local features as "landmarks" of the string. A landmark (say $a_i[j]$) has the property that if a_i is transformed into a'_i by an edit operation (say character insertion at k) far away from j, so $|k-j| >> 1$), our partitioning strategy will ensure that $a'_i[j]$ will still be designated a landmark. In other words, an edit operation on $a_i[k]$ will only affect j being a landmark if j is close to k. This will have the effect that each edit operation will only change $O(\max_j |k_j - j|)$ nodes of the ESP tree at every level, where k_j is the closest unaffected landmark to j. In order to inflict the minimal number of changes to the ESP tree, we would like this quantity to be as small as possible, but still require a_i's to be geometrically decreasing in size.

In what follows, we will describe our method for marking landmarks and partitioning a_i into blocks more precisely. Repeated characters — any substring of the form $\mathtt{a}^i$ for some character $\mathtt{a}$ — will make a good landmark, since these repeated characters will be identifiable irrespective of any inserts or deletes that take place around them. But we cannot guarantee that there will be repeated characters often enough (or at all) to build a low degree tree. Another good way to make these landmarks is to define a total order on the alphabet σ, and pick out characters that are locally maximal under this ordering. However, the distance between such maxima can be $\theta(|\sigma|)$, and string edit operations will have an impact distance $\Theta(|\sigma|)$ away, which is too large. Repeatedly using such a procedure to parse the string hierarchically, as we do, will aggravate this problem. We will combine these ideas and overcome the deficiencies by using a procedure of alphabet reduction to guarantee that these landmarks do occur close enough together. We canonically parse any string into maximal non-overlapping substrings of three types:

1. Maximal contiguous substrings of a_i that consist of a repeated symbol (so they are of the form $\mathtt{a}^l$ for $\mathtt{a} \in \sigma_i$ where $l > 1$),

2. "Long" substrings of length at least $\log^* |\sigma_{i-1}|$ not of type 1 above.

3. "Short" substrings of length less than $\log^* |\sigma_{i-1}|$ not of type 1.

Each such substring is called a *metablock*. We process each metablock as described below to generate the next level in the parsing.

4.2.2 Parsing of Different Metablocks

Type 2: Long strings without repeats

The discussion here is similar to those in [GPS87] and [MSU97]. Suppose we are given a string s in which no two adjacent symbols are identical and which is counted as a metablock of type 2. We will carry out a procedure on it which will enable it to be parsed into nodes of two or three symbols.

Given a sequence s with no repeats (i.e., $s[i] \neq s[i+1]$ for $i = 1 \ldots |s|-1$), we will designate at most $|s|/2$ and at least $|s|/3$ substrings of s as *nodes*. The concatenation of these nodes gives s. The first stage consists of iterating an alphabet reduction technique. This is effectively the same procedure as the Deterministic Coin Tossing in [GPS87], but applied to strings.

(i)	c	a	b	a	g	e	h	e	a	d	b	a	g
(ii)	010	000	001	000	110	100	111	100	000	011	001	000	110
(iii)	-	010	001	000	011	010	001	000	100	001	010	000	011
(iv)	-	2	1	0	3	2	1	0	4	1	2	0	3
(v)	-	$\boxed{2}$	1	$\boxed{0}$	1	$\boxed{2}$	1	0	$\boxed{2}$	1	$\boxed{2}$	0	$\boxed{1}$

The original text, drawn from an alphabet of size 8 (i), is written out as binary integers (ii). Following one round of alphabet reduction, the new alphabet is size 6 (iii), and the new text is rewritten as integers (iv). A final stage of alphabet reduction brings the alphabet size to 3 (v) and local maxima and some local minima are used as landmarks (denoted by boxes)

Figure 4.1: The process of alphabet reduction and landmark finding

Alphabet reduction.

For each symbol $s[i]$ compute a new label, as follows. $s[i-1]$ is the left neighbour of $s[i]$, and consider $s[i]$ and $s[i-1]$ represented as binary integers. Denote by l the index of the least significant bit in which $s[i]$ differs from $s[i-1]$, so $l = \min\{j | s[i] \neq s[i-1] \mod 2^{j-1}\}$. Let $bit(l, s[i])$ be the value of $s[i]$ at that bit location, so $bit(l, s[i]) = \lfloor s[i]/2^l \rfloor \mod 2$. Form $label(s[i])$ as $2l + bit(l, s[i])$ — in other words, as the index l followed by the value at that index.

Lemma 4.2.1 *For any i, if $s[i] \neq s[i+1]$ then $label(s[i]) \neq label(s[i+1])$.*

Proof. Suppose that the least significant bit position at which $s[i]$ differs from $s[i+1]$ is the same as that at which $s[i]$ differs from $s[i-1]$ (otherwise, $label(s[i]) \neq label(s[i+1])$). But the bit values at this location in each character must differ, and hence $label(s[i]) \neq label(s[i+1])$. □

Following this procedure, we generate a new sequence. If the original alphabet was size τ, then the new alphabet is sized $2 \log |\tau|$. We now iterate and perform the alphabet reduction until the size of the alphabet no longer shrinks. This iteration is orthogonal to the iteration that constructs the ESP tree of a; we are iterating on s which is a sequence with no identical adjacent symbols. This takes $\log^* |\tau|$ iterations. Note that there will be no labels for the first $\log^* |\tau|$ characters.

Lemma 4.2.2 *After the final iteration of alphabet reduction, the alphabet size is 6.*

Proof. At each iteration of tagging, the alphabet size goes from $|\sigma|$ to $2\lceil \log |\tau| \rceil$. If $|\tau| > 6$, then $2\lceil \log |\tau| \rceil$ is strictly less than this quantity. □

Since s did not have identical adjacent symbols, neither does the final sequence of labels on s using Lemma 4.2.1 repeatedly.

Finally, we perform three passes over the sequence of symbols to reduce the alphabet from $\{0 \dots 5\}$ to $\{0, 1, 2\}$: first we replace each 3 with the least element from $\{0, 1, 2\}$ that does not neighbour the 3, then do the same for each 4 and 5. This generates a sequence of labels drawn from the alphabet {0,1,2} where no adjacent characters are identical. Denote this sequence as s'.

Finding landmarks

We can now pick out special locations, known as *landmarks*, from this sequence that are sufficiently close together. We first select any position i which is a *local maximum*, that is, $s'[i-1] < s'[i] > s'[i+1]$, as a landmark. Two maxima could still have three intervening labels, so in addition we select as a landmark any i which is a local minimum that is, $s'[i-1] > s'[i] < s'[i+1]$, and is not adjacent to an already chosen landmark. An example of the whole process is given in Figure 4.1.

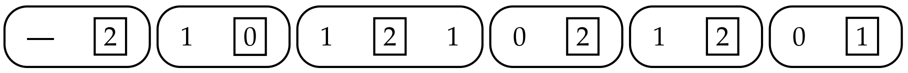


Figure 4.2: Given the landmark characters, the nodes are formed.

Lemma 4.2.3 *For any two successive landmark positions i and j, we have $2 \leq |i - j| \leq 3$.*

Proof. By our marking procedure, we insist that no adjacent pair of tags are marked — since we cannot have two adjacent maxima, and we specifically forbid marking any local minimum which is next to a maximum. Simple case analysis shows that the separation of landmark positions is at most two intervening symbols. □

Lemma 4.2.4 *Determining the closest landmark to position i depends on only $\log^* |\tau| + 5$ contiguous positions to the left and 5 to the right.*

Proof. After one iteration of alphabet reduction, each label depends only on the symbol to its left. We repeat this $\log^* |\tau|$ times, hence the label at position i depends on $\log^* |\tau|$ symbols to its left. When we perform the final step of alphabet reduction from an alphabet of size six to one of size three, the final symbol at position i depends on at most three additional symbols to its left and to its right. We must mark any position that is a local maximum, and then any that is a local minimum not adjacent to a local maximum; hence we must examine at most two labels to the left of i and two labels to the right, which in turn each depend on $\log^* |\tau| + 3$ symbols to the left and 3 to the right. The total dependency is therefore as stated. □

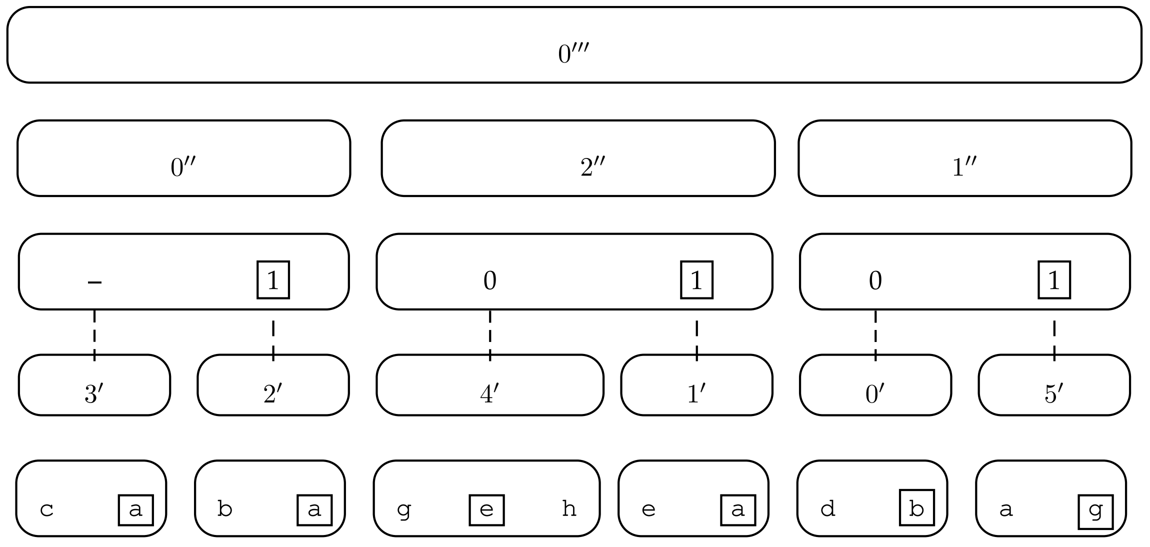
Now we show how to partition s into blocks of length 2 or 3 around the landmarks. We treat the leftmost $\log^* |\sigma_{i-1}|$ symbols of the substring as if they were a short metablock (type 3, the procedure for which is described below). The other positions are treated as follows. Since there are either one or two positions between each landmark, it is simply a matter of dealing with different cases and boundary conditions deterministically. We make each position part of the block generated by its closest landmark, breaking ties to the right (see Figure 4.2). Consequent of Lemma 4.2.3 each block is now of length two or three.

Type 1 (Repeating metablocks) and Type 3 (Short metablocks)

Recall that we seek "landmarks" which can be identified easily based only on a local neighbourhood. Then we can treat repeating metablocks as large landmarks. Type 1 and Type 3 blocks can each be parsed in a regular fashion, the details we give for completeness. Metablocks of length one would be attached to the repeating metablock to the left or the right, with preference to the left when both are possible, and parsed as described below. Metablocks of length two or three are retained as blocks without further partitioning, while a metablock of length four is divided into two blocks of length two. In any metablock of length five or more, we parse the leftmost three symbols as a block and iterate on the remainder.

4.2.3 Constructing $ET(a)$

Having partitioned a_i into blocks of 2 or 3 symbols, we construct a_{i+1} by replacing each block b by $hash(i, b)$ where $hash$ is a perfect one-to-one (hash) function. Each block is a node in the parse tree, and its children are the 2 or 3 nodes from which it was formed. Note that blocks of different levels can use different hash functions for computing names, so we focus on any given level i, noting that we wish to ensure that two blocks get the same name if and only if they represent the same substring, and they occur at the same level in the parsing. If we use randomisation, $hash()$ can be computed for any block



Following the generation of the first level nodes, we relabel from the alphabet $\{0', 1' \ldots\}$. We parse this string, generating three second level nodes which we relabel from $\{0'', 1'' \ldots\}$, which in turn become a single top level node.

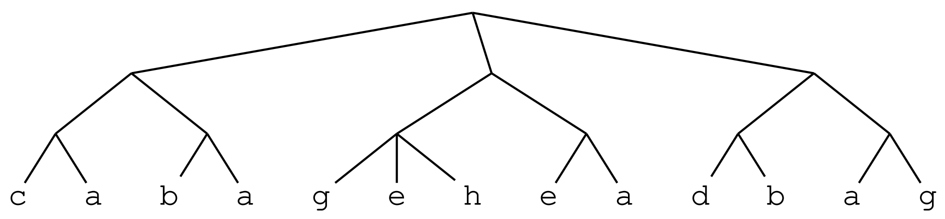
Figure 4.3: The hierarchy is formed by repeated application of alphabet reduction and landmark finding

(recall they are of length 2 or 3) in $O(1)$ time using Karp-Rabin fingerprints (from Section 2.1.3); if we set the base large enough, then we guarantee the probability of a collision between *any* pair to be small. In total, we have at most n blocks to deal with, and we wish them to each be given a distinct name. We are setting the parameter δ of the Karp-Rabin hash functions, and there are $\binom{n}{2}$ pairs of blocks which can collide. The probability of no collisions is bounded by $1 - \binom{n}{2}\delta$, making us choose $\delta = 2\delta'/n^2$ for some chosen probability δ'. Then the procedure succeeds with probability at least $1 - \delta'$, and no blocks collide.

For deterministic solutions, we can use the algorithm in [KMR72].

Karp-Miller-Rosenberg labelling The algorithm given in [KMR72] describes a method of efficiently computing a naming function for a string that ensures that two substrings are given the same name if and only if they are identical. The algorithm proceeds by doubling: if we have a naming scheme for all substrings of length m then these are used to give names to all substrings of length $2m$. Clearly, the characters themselves can be used to name all strings of length 1. From these we can create names for all substrings of length $2, 4, 8 \ldots n$. These naming functions have the property that $hash_k(a[i : i + 2^k - 1]) = hash_k(a[j : j + 2^k - 1]) \iff a[i : i + 2^k - 1] = a[j : j + 2^k - 1]$. With careful implementation as described in [KMR72], this naming can be performed in time $O(|a| \log |a|)$. A name can then be given to *any* substring in time $O(1)$ using lookups: the label for $a[l : r]$ can be formed using two labels. Let $k = \lfloor \log_2(r - l + 1) \rfloor$. Then the label for $a[l : r]$ is the triple $(hash_k(a[l : l + 2^k]), hash_k(a[r - 2^k : r]), r - l - 1)$. Clearly, two substrings are assigned the same label by this procedure if and only if they are identical.

Following this relabelling of the sequence, we have generated a new sequence a_{i+1}; we then iterate this procedure until the sequence is of length 1: this is then the root of the tree. Let $|a_i|$ be the number of nodes in $ET(a)$ at level i. Since the first (leaf) level is formed from the characters of the original string, $|a_0| = |a|$. We have $|a_i|/3 \leq |a_{i+1}| \leq \lfloor |a_i|/2 \rfloor$. Therefore, $\frac{3}{2}|a| \leq \sum_i |a_i| \leq 2|a|$. Hence



$T(a) = \{$ `c, a, b, a, g, e, h, e, a, d, b, a, g, ca, ba, geh, ea, db, ag, caba, gehea, dbag, cabageheadbag`$\}$

Figure 4.4: The hierarchical structure of nodes is represented as a parse tree on the string a.

for any i, $|\sigma_i| \leq |a|$ (recall that this $hash()$ is one-to-one) and so $\log^* |\sigma_i| \leq \log^* |a|$. The example is completed in Figure 4.3.

Theorem 4.2.1 *Given a string a, its ESP tree $ET(a)$ can be computed in time $O(|a| \log^* |a|)$.*

4.3 Properties of ESP

We can compute $ET(a)$ for any string a as described above (see Figure 4.4). Each node x in $ET(a)$ represents a substring of a given by the concatenation of the leaf nodes in the subtree rooted at x.

Definition 4.3.1 *Define the multi-set $T(a)$ as all substrings of a that are represented by the nodes of $ET(a)$ (over all levels). We define $V(a)$ to be the "characteristic vector" of $T(a)$, that is, $V(a)[i, x]$ is the number of times a substring x appears in $T(a)$ at level i. Finally, we define $V_i(a)$ as the characteristic vector restricted to only nodes which occur at level i in $ET(a)$.*

Note that $T(a)$ comprises at most $2|a|$ strings of length at most $|a|$. $V(a)$ is an $O(|\sigma|^{|a|})$ dimensional vector since its domain is any string that may be present in $T(a)$; however, it is a (highly) sparse vector since at most $2|a|$ components are nonzero. In an implementation, this set could be efficiently stored using pointers into the string a.

As usual, we denote the standard L_1 distance between two vectors u and v by $||u - v||_1$. By definition, $||V(a) - V(b)||_1 = \sum_{x \in T(a) \cup T(b)} |V(a)[x] - V(b)[x]|$. Recall that $d(a, b)$ denotes the edit distance with moves between strings a and b. Our main theorem on this structure shows that $V()$ is an approximation preserving embedding of string edit distance with moves.

Theorem 4.3.1 *For strings b and a, let n be $\max(|a|, |b|)$. Then*

$$d(a, b) \leq 2||V(b) - V(a)||_1 \leq 16 \log n \log^* n \; d(a, b)$$

4.3.1 Upper Bound Proof

$$||V(b) - V(a)||_1 \leq 8 \log n \; \log^* n \cdot d(a, b)$$

Proof. To show this bound on the L_1 distance, we consider the effect of the editing operations, and demonstrate that each one causes a contribution to the L_1 distance that is bounded by $O(\log n \log^* n)$. Note again that, as mentioned in the introduction, edit operations are allowed to "overlap" on blocks that were previously the subject of moves and other operations. We give a Lemma which is similar to Lemma 4.2.4 but which applies to any string, not just those with no adjacent repeated characters.

Lemma 4.3.1 *The closest landmark to any symbol of a_i is determined by at most $\log^* |\sigma_i| + 5$ consecutive symbols of a_i to the left, and at most 5 consecutive symbols of a_i to the right.*

Proof. Given a symbol of a_i, say $a_i[j]$, we show how to find the closest landmark.
Type 1 Repeating metablock Recall that a long repeat of a symbol a is treated as a single, large landmark. $a_i[j]$ is included in such a meta-block if $a_i[j] = a_i[j+1]$ or if $a_i[j] = a_i[j-1]$. We also consider $a_i[j]$ to be part of a repeating substring if $a_i[j-1] = a_i[j-2]$; $a_i[j+1] = a_i[j+2]$; and $a_i[j] \neq a_i[j+1]$ and $a_i[j] \neq a_i[j-1]$ — this is the special case of a metablock of length one. In total, only 2 consecutive symbols to the left and right need to be examined.
Types 2 and 3 Non-repeating metablocks If it is determined that $a_i[j]$ is not part of a repeating metablock, then we have to decide whether it is in a short or long metablock. We examine the substring $a_i[j - \log^* |\sigma_i| - 3 : j-1]$. If there is any k such that $a_i[k] = a_i[k-1]$ then there is a repeating metablock terminating at position k. This is a landmark, and so we parse $a_i[j]$ as part of a short metablock, starting from $a_i[k+1]$ (recall that the first $\log^* |\sigma_i|$ symbols of a long metablock get parsed as if they were in a short metablock). Examining the substring $a_i[j+1 : j+5]$ allows us to determine if there is another repeating metablock this close to position j, and hence we can determine what node to form containing $a_i[j]$. If there is no repeating metablock evident in $a_i[j - \log^* |\sigma_i| - 3 : j-1]$ then it is possible to apply the alphabet reduction technique to find a landmark. From Lemma 4.2.4, we know that this can be done by examining $\log^* |\sigma_i| + 5$ consecutive symbols to the left and 5 to the right. □

This ability to find the nearest landmark to a symbol by examining only a bounded number of consecutive neighbouring symbols means that if an editing operation occurs outside of this region, the same landmark will be found, and so the same node will be formed containing that symbol. This allows us to prove the following lemma.

Lemma 4.3.2 *Inserting $k \leq \log^* n + 10$ consecutive characters into a to get a' means $||V_i(a) - V_i(a')||_1 \leq 2(\log^* n + 10)$ for all levels i.*

Proof. We shall make use of Lemma 4.3.1 to show this. We have a contribution to the L_1 distance from the insertion itself, plus its effect on the surrounding locality. Consider the total number of symbols at level i that are parsed into different nodes after the insertion compared to the nodes beforehand. Let the number of symbols at level i which are parsed differently as a consequence of the insertion be M_i. Lemma 4.3.1 means that in a non-repeating metablock, any symbol more than 5 positions to the left, or $\log^* |\sigma_i| + 5$ positions to the right of any symbols which have changed, will find the same closest landmark as it did before, and so will be formed into the same node. Therefore it will not contribute to M_i. Similarly, for a repeating metablock (type 1), any symbol inside the block will be parsed into the same node (that is, into a triple of that symbol), except for the last 4 symbols, which depend on the length of the block. So for a repeating metablock, $M_i \leq 4$. The number of symbols from the level below which are parsed differently into nodes as a consequence of the insertion is at most $M_{i-1}/2$, and there is a region of at most 5 symbols to the left and $\log^* |\sigma_i| + 5$ symbols to the right which will be parsed differently at level i. Because $|\sigma_i| \leq |a| \leq n$ as previously observed, we can therefore form the recurrence, $M_i \leq M_{i-1}/2 + \log^* n + 10$. If $M_{i-1} \leq 2(\log^* n + 10)$ then $M_i \leq 2(\log^* n + 10)$. From the insertion itself, $M_0 \leq \log^* n + 10$. Finally $||V_i(a) - V_i(a')||_1 \leq 2(M_{i-1}/2)$, since we could lose $M_{i-1}/2$ old nodes, and gain this many new nodes. □

Lemma 4.3.3 *Deleting $k \leq \log^* n + 10$ consecutive symbols from a to get a' means $||V_i(a) - V_i(a')||_1 \leq 2(\log^* n + 10)$.*

Proof. Observe that a deletion of a sequence of labels is precisely the dual to an insertion of that sequence at the same location. If we imagine that a sequence of characters is inserted, then deleted, the resultant string is identical to the original string. Therefore, the number of affected nodes must be bounded by the same amount as for an insertion, as described in Lemma 4.3.2. □

We combine these two lemmas to show that editing operations have only a bounded effect on the parse tree.

Lemma 4.3.4 *If a single permitted edit operation transforms a string a into a' then $||V(a) - V(a')||_1 \leq 8 \log n(\log^* n + 10)$.*

Proof. We consider each allowable operation in turn.
Character edit operations. The case for insertion follows immediately from Lemma 4.3.2 since the effect of the character insertion affects the parsing of at most $2(\log^* n + 10)$ symbols at each level and there are at most $\log_2 n$ levels. In total then $||V(a) - V(a')||_1 \leq 2 \log n(\log^* n + 10)$. Similarly, the case for deletion follows immediately from Lemma 4.3.3. Finally, the case for a replacement operation (if allowed) is shown by noting that a character replacement can be considered to be a deletion immediately adjacent to an insertion.
Substring Moves. If the substring being moved is at most $\log^* n + 10$ in length, then a move can be thought of as a deletion of the substring followed by its re-insertion elsewhere. From Lemma 4.3.2 and Lemma 4.3.3, then $||V(a) - V(a')||_1 \leq 4 \log n(\log^* n + 10)$. Otherwise, we consider the parsing of the substring using ESP. Consider a character in a non-repeating metablock which is more than $\log^* n + 5$ characters from the start of the substring and more than 5 characters from the end. Then according to Lemma 4.3.1, only characters within the substring being moved determine how that character is parsed. Hence the parsing of all such characters, and so the contribution to $V(a)$, is independent of the location of this substring in the string. Only the first $\log^* n + 5$ and last 5 characters of the substring will affect the parsing of the string. We can treat these as the deletion of two substrings of length $k \leq \log^* n + 10$ and their re-insertion elsewhere. For a repeating metablock, if this extends to the boundary of the substring being moved then still only 4 symbols of the block can be parsed into different nodes. So by appealing to Lemmas 4.3.2 and 4.3.3 then $||V(a) - V(a')|| \leq 8 \log n(\log^* n + 10)$. □

Lemma 4.3.4 shows that each allowable operation affects the L_1 distance of a transform by at most $8 \log n(\log^* n + 10)$. Suppose we begin with a, and perform a series of d editing operations, generating $a_1, a_2, \ldots a_d$. At the conclusion, $a_d = b$, so $||V(a_d) - V(b)||_1 = 0$. We begin with a quantity $||V(b) - V(a)||_1$, and we also know that at each step from the above argument $||V(a_j) - V(a_{j+1})|| \leq 8 \log n(\log^* n + 10)$. Hence, if $d(a, b)$ is the minimum number of operations to transform a into b, then $d(a, b)$ must be at least $||V(b) - V(a)||_1 / 8 \log n(\log^* n + 10)$, giving a bound of $d(a, b) \cdot 8 \log n(\log^* n + 10)$.
□

4.3.2 Lower Bound Proof

$$d(a, b) \leq 2||V(b) - V(a)||_1$$

Here, we shall prove a slightly more general statement, since we do not need to take account of any of the special properties of the parsing; instead, we need only assume that the parse structure built on the strings has bounded degree (in this case three), and forms a tree whose leaves are the characters of the string. Our technique is to show a particular way we can use the 'credit' from the potential function $||V(b) - V(a)||_1$ to transform a into b. We give a constructive proof, although the computational efficiency of the construction is not important. For the purpose of this proof, we treat the parse trees as if they were static tree structures, so following an editing operation, we do not need to consider the effect this has on the parse structure.

Lemma 4.3.5 *If the trees which represent the transforms have degree at most k, then the tree $ET(b)$ can be made from the tree $ET(a)$ using no more than $(k-1)||V(a) - V(b)||_1$ move, insert and delete operations.*

Proof. We first ensure that any good features of a are preserved. In a top-down, left to right pass over the tree of a, we 'protect' certain nodes — we place a mark on any node x that occurs in the parse tree of both a and b, provided that the total number of nodes marked as protected does not exceed $V_i(b)[x]$. If a node is protected, then all its descendents become protected. The number of marked copies of any node x is $\min(V(a)[x], V(b)[x])$. Once this has been done, the actual editing commences, with the restriction that we do not allow any edit operation to split or change a protected node.

We shall proceed bottom-up in $\log n$ rounds ensuring that after round i when we have created a_i that $||V_i(a) - V_i(b)||_1 = 0$. The base case to create a_0 deals with individual characters, and is trivial: for any symbol $\mathtt{c} \in \sigma$, if $V_0(a)[\mathtt{c}] > V_0(b)[\mathtt{c}]$ then we delete the $(V_0(a)[\mathtt{c}] - V_0(b)[\mathtt{c}])$ unmarked copies of $\mathtt{c}$ from a; else if $V_0(a)[\mathtt{c}] < V_0(b)[\mathtt{c}]$ then at the end of a we insert $(V_0(b)[\mathtt{c}] - V_0(a)[\mathtt{c}])$ copies of $\mathtt{c}$. In each case we perform exactly $|V_0(b)[\mathtt{c}] - V_0(a)[\mathtt{c}]|$ operations, which is the contribution to $||V_0(b) - V_0(a)||_1$ from symbol $\mathtt{c}$. a_0 then has the property that $||V_0(b_0) - V_0(a)||_1 = 0$.

Each subsequent case follows an inductive argument: assuming we have enough nodes of level $i-1$ (so $||V_{i-1}(a) - V_{i-1}(b_{i-1})||_1 = 0$), we show how to make a_i using just $(k-1)||V_i(a) - V_i(b)||_1$ move operations. Consider each node x at level i in the tree $ET(b)$. If $V_i(a)[x] \geq V_i(b)[x]$, then we would have protected $V_i(b)[x]$ copies of x and not altered these. The remaining copies of x will be split to form other nodes. Else $V_i(b)[x] > V_i(a)[x]$ and we would have protected $V_i(a)[x]$ copies of x. Hence we need to build $V_i(b)[x] - V_i(a)[x]$ new copies of x, and the contribution from x to $||V_i(a) - V_i(b)||_1$ is exactly $V_i(b)[x] - V_i(a)[x]$: this gives us the credit to build each copy of x. To make each of the copies of x, we need to bring together at most k nodes from level $i-1$. So pick one of these, and move the other $k-1$ into place around it (note that we can move any node from level $i-1$ so long as its *parent* is not protected). We do not care *where* the node is made — this will be taken care of at higher levels. Because $||V_{i-1}(a) - V_{i-1}(b_{i-1})||_1 = 0$ we know that there are enough nodes from level $i-1$ to build every level i node in b. We then require at most $k-1$ move operations to form each copy of x by moving unprotected nodes. □

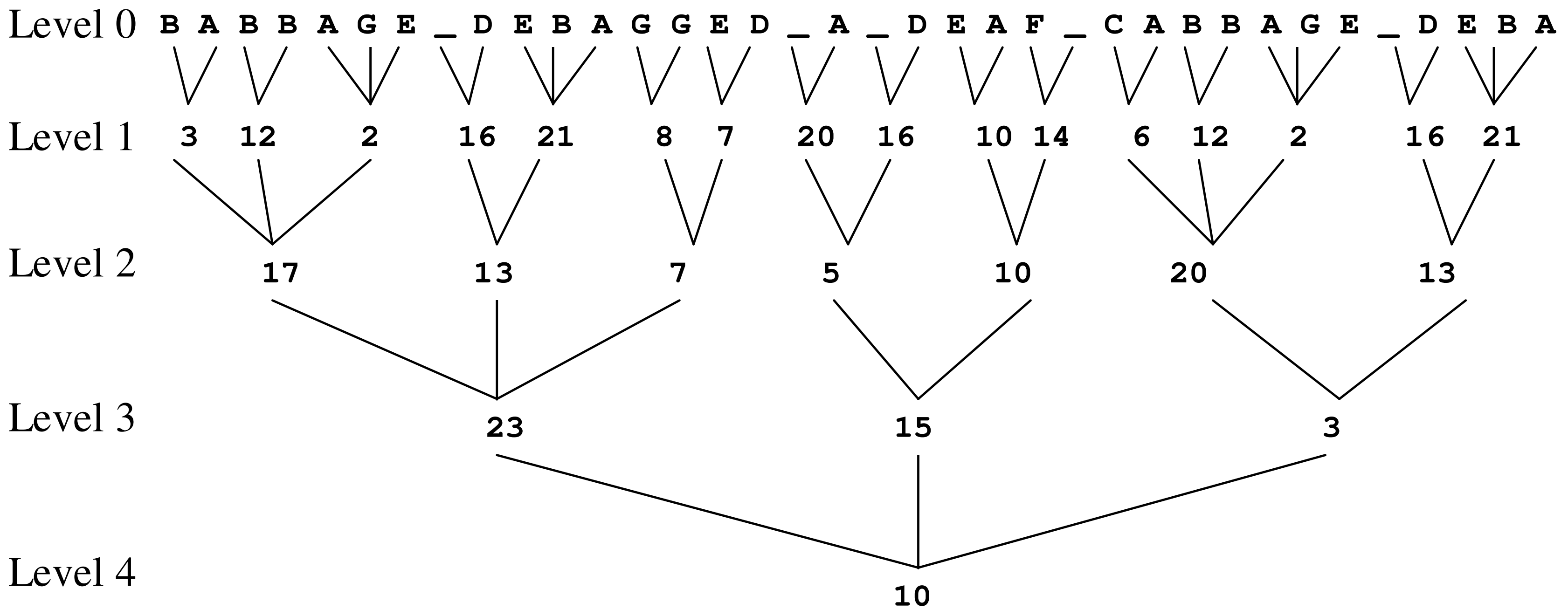
Since this inductive argument holds, and we use at most $k-1 = 2$ moves for each contribution to the L_1 distance, the claim follows.

Example

An extended example of this embedding is given in Figures 4.5, 4.6, 4.7 and 4.8. It shows how the embedding into a vector is found from the parsing, and then how one string may be converted into another using a number of operations linear in the size of the L_1 difference of their representing vectors.

4.4 Embedding for other block edit distances

The ESP approach can be adapted to handle similar string distance measures, which allow additional operations such as substring reversals, linear scaling and copying, amongst others. These can be handled for the most part by altering the methodology for the alphabet reduction step which ensures that, for example, substrings are parsed in the same way as their reverse. This approach to this kind of parsing was discussed in [MŞ00], and we do not go into further detail. Instead, we focus on the compression distance, since here the embedding is into Hamming distance rather than L_1 distance. It makes use of exactly the same ESP parsing of strings, suggesting the wider applicability of this technique.

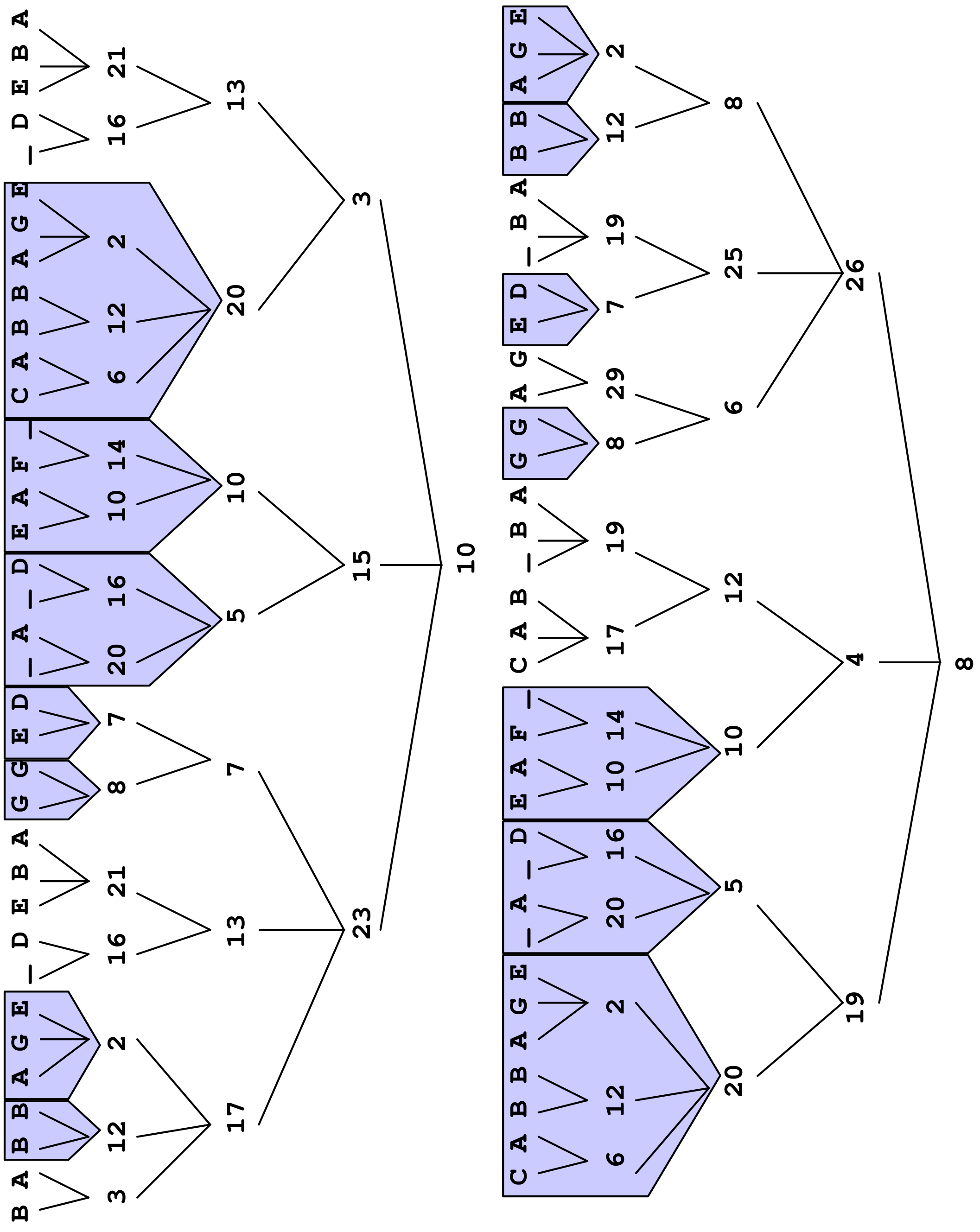


(0,**A**)	(0,**B**)	(0,**C**)	(0,**D**)	(0,**E**)	(0,**F**)	(0,**G**)	(0,_)	(1,**2**)	(1,**3**)	(1,**6**)	(1,**7**)	(1,**8**)	(1,**10**)	(1,**12**)
8	7	1	4	6	1	4	5	2	1	1	1	1	1	2

(1,**14**)	(1,**16**)	(1,**20**)	(1,**21**)	(2,**5**)	(2,**7**)	(2,**10**)	(2,**13**)	(2,**17**)	(2,**20**)	(3,**3**)	(3,**15**)	(3,**23**)	(4,**10**)
1	3	1	2	1	1	1	2	1	1	1	1	1	1

Each entry in the vector corresponds to a pair of the level and the name of the node. The entry then counts how many copies of this node occur in the parsing. So there are 8 copies of the character '**A**', 2 copies of node 12 in level one, and a single copy of node 10 at level four. Any entries not present in this vector representation are zero.

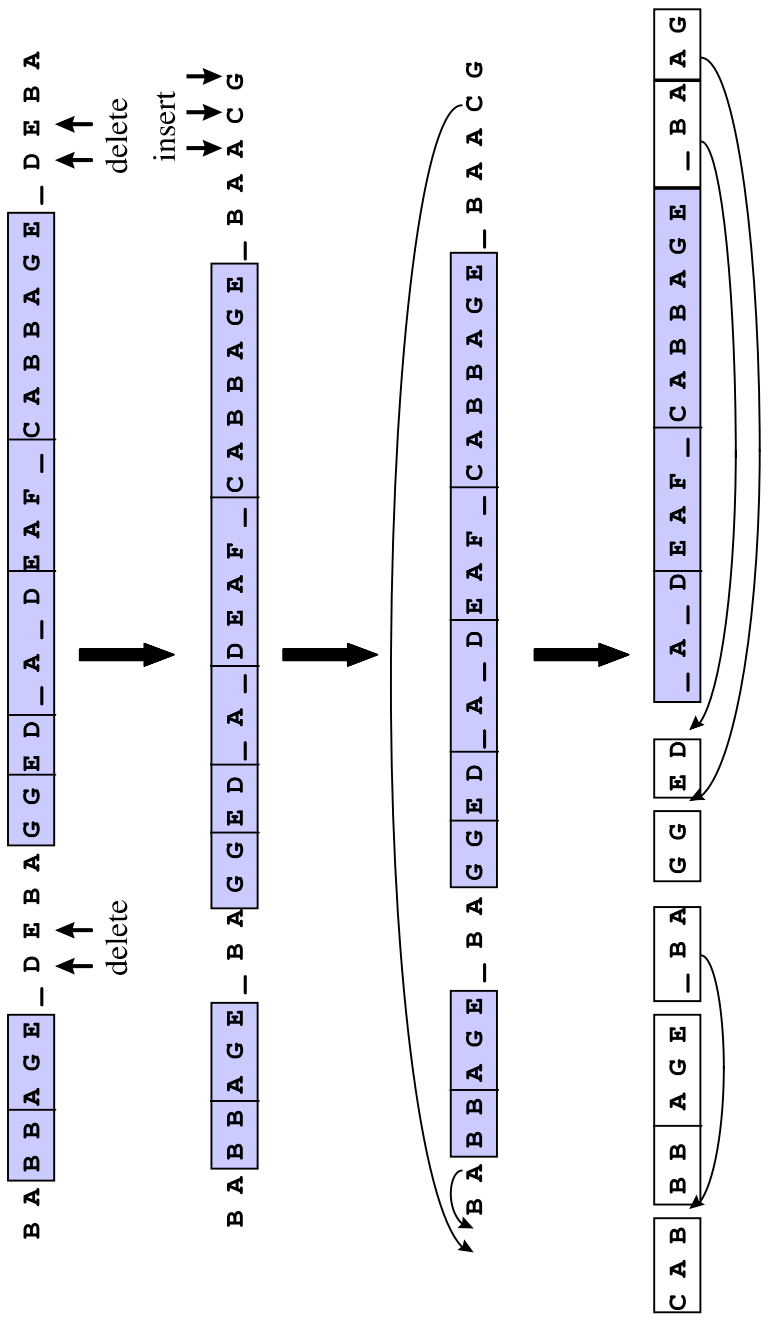
Figure 4.5: The hierarchical parsing induces a vector representation



B A B B A G E _ D E B A G G E D _ A _ D E A F _ C A B B A G E _ D E B A
C A B B A G E _ A _ D E A F _ C A B _ B A G G A G E D _ B A B B A G E

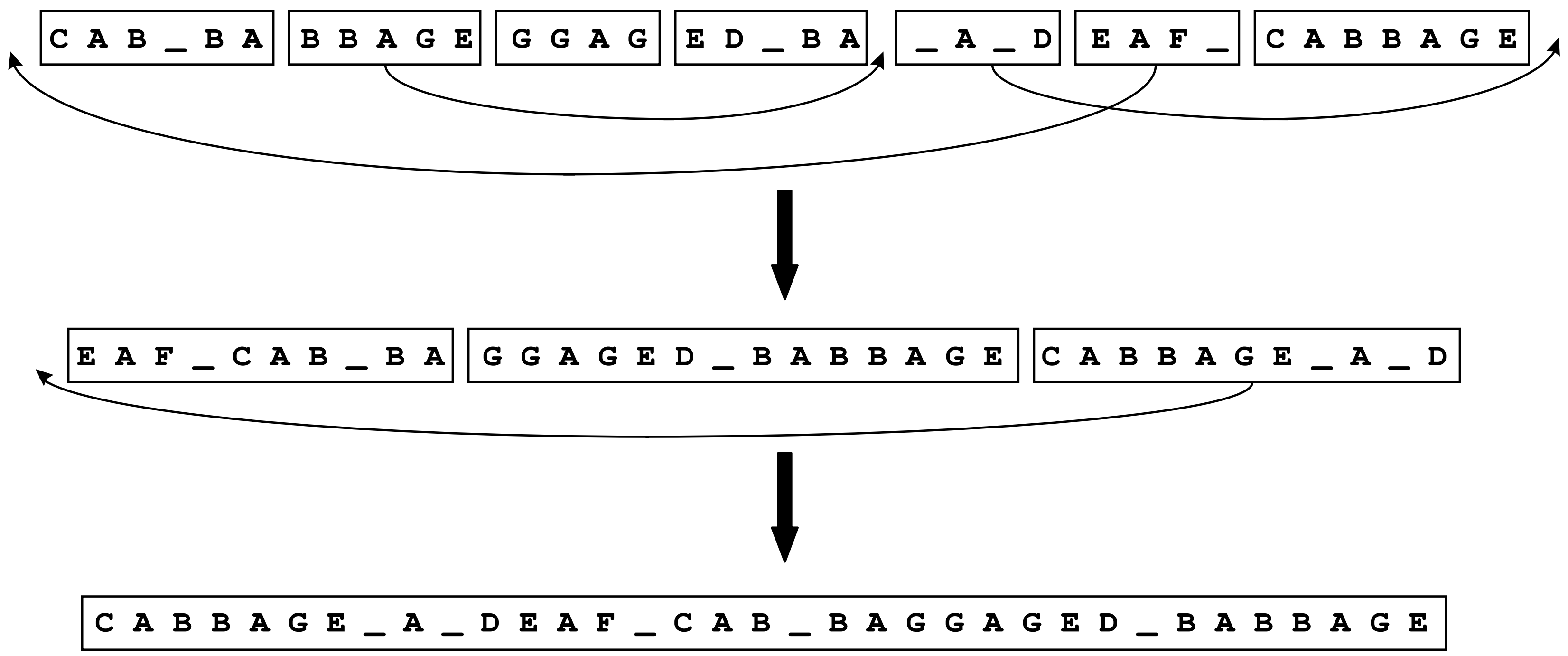
We mark as protected enough nodes that are common to both parsings, ensuring that we do not protect more nodes in the original string than are needed in the target. Following this step, we will not break up protected nodes.

Figure 4.6: For the lower bound proof, we mark nodes that are common to both strings



We first delete any characters that there is an excess of (ensuring deletions take place outside of protected blocks). Then characters that there are too few of are inserted at the end. This completes the base case. We then form the nodes of level 1 by moving characters. Studying Figure 4.5, the level 1 nodes are CA, BB, AGE, _A, _D, EA, F_,CAB, _BA, GG, AG, ED, _BA, BB, AGE. To make the level 2 nodes, we need to make CAB_BA, GGAG, ED_BA, BBAGE (other nodes are protected). This is done by moving nodes of level 1.

Figure 4.7: The first stages of editing the string



Next, we move around the nodes of level 2 to form the nodes of level 3, which are: `CABBAGE_A_D`, `EAF_CAB_BA`, `GGAGED_BABBAGE`. Finally, we move nodes of level 3 around to form the target string.

Figure 4.8: The target string is completed

4.4.1 Compression Distance

For this embedding we need to use the idea of the Symmetric Difference of Multi-sets. This is defined to be the symmetric difference of the supporting sets of the multi-sets. We will deal with $T(a)\Delta T(b)$ which indicates the symmetric difference of the supporting sets of the multi-sets $T(a)$ and $T(b)$. This is equivalent to $H_0(V(a), V(b))$, the zero based Hamming difference between their vector representations (Definition 1.2.9). For the purpose of the following theorem, we treat these multi-sets $T(a)$ and $T(b)$ as their support sets.

Theorem 4.4.1 *For strings a and b with* $\max(|a|, |b|) = n$,

$$c(a, b) \leq 3|T(a)\ \Delta\ T(b)| = O(\log n \log^* n) \cdot c(a, b)$$

Proof. We need to give a proof analogous to that for Theorem 4.3.1. Again, we show an upper and lower bound.

Upper bound

As in the proof of Theorem 4.3.1, we will show that each permitted edit operation has a limited effect. In particular, we show that if an edit operation transforms a into a', then $|T(a)\Delta T(a')| = O(\log n \log^* n)$. We already know that every move, insert, delete or change operation affects the L_1 distance by at most $O(\log n \log^* n)$. Since $H_0(V(b), V(a)) \leq ||V(b) - V(a)||_1$ (that is, the Hamming distance of two integer valued vectors is less than the L_1 distance, Lemma 2.3.2), then we only need to consider the extra operations, of copying and uncopying a block.

For copy operations, we consider a copy of a (long) block of a generating a'. Note that any element within the block that is more than $\log^* |\sigma| + 5$ from the left end of the block, or 5 from the right end, will be oblivious of whatever goes on outside the block, and so will be parsed in the same way in both copies. So these do not affect the Hamming distance of the parsings. Therefore, the copy operation will have an effect no greater than that of inserting the first $\log^* |\sigma| + 5$ and last 5 elements, since we can neglect the centre of the block from after the first $\log^* |\sigma| + 5$ characters to 5 from the end. We have seen that such an insertion affects at most $8 \log n(\log^* n + 10)$ existing nodes, and it will add only $O(\log^* n)$ nodes for the inserted blocks themselves, so $|T(a)\Delta T(a')| = O(\log n \log^* n)$. For a block deletion, note that we can only delete a block that has a copy elsewhere in the sequence, so that deletion is the inverse operation of copying (uncopying). For this case also, $|T(a)\Delta T(a')| = O(\log n \log^* n)$. Consequently, each operation can contribute $O(\log n \log^* n)$ to the Hamming distance, and so the total zero based Hamming distance between $V(a)$ and $V(b)$ can be at most $c(a, b)O(\log n \log^* n)$.

Lower bound.

To show $c(a, b) \leq 3|T(b)\ \Delta\ T(a)|$, we first show how given a we can build the compound string ab (a followed by b) using a number of operations linear in the zero based Hamming difference between the transforms. A special case to deal with is when b is a substring of a, or vice-versa. We shall consider this case shortly, but in the meantime we assume that this is not the case.

Lemma 4.4.1 *The compound string ab can be built from string a using no more than $3|T(b)\backslash T(a)|$ copy and insert operations.*

Proof. Again, we will treat set operations on multi-sets in terms of their support sets, so $T(b)\backslash T(a)$ is the set of items present in b but not present in a. We begin with the highest level node of $ET(b)$ (note that there will be precisely one, which represents the whole sequence b). We shall proceed inductively on the nodes of $ET(b)$. Since we assume that b is not a substring of a, and vice-versa, then this node is not present in a, in which case it contributes to $T(b)\backslash T(a)$. We will use three units of potential from this element of the difference, and allocate these by giving one unit to each of its children (recall that each level-i node is formed from at most 3 level-$(i-1)$ nodes). The induction considers each node that has been given a credit from the top down. The base case is if a node under consideration is a leaf node. This character can be inserted at unit cost. Otherwise, the node represents a substring of b. If this child node is present in a, or already present in the partially built string b, then the unit credit can be used to copy the string that the node represents. Otherwise, we can apply this argument recursively (since this node is present in b but not in a and hence contributes to the set difference) to build this node by forming its children in place. □

We can use exactly the same argument symmetrically to argue that starting from b we can build the compound string ab using $c(b, ab) \leq 3|T(b)\backslash T(a)|$ operations. Since c is a metric, it is symmetric and so $c(ab, b) \leq 3|T(b)\backslash T(a)|$. Using the triangle inequality we have that $c(a, b) \leq c(a, ab) + c(ab, b) \leq 3|T(b)\backslash T(a)| + 3|T(a)\backslash T(b)| = 3|T(a) \Delta T(b)|$.

We finally dispense with the case that a is a substring of b. In this case we fix a, and attempt to build b around a in a similar way to before: for each node in b that is not present, we use credit to build its children. We only require credit for nodes that are in b but not in a, hence there are only $3|T(b)\backslash T(a)| \leq 3|T(a) \Delta T(b)|$ operations. The case where b is a substring of a follows by symmetry of the metric c. □

This theorem essentially states that if we consider the embedding of the ESP transformation into zero based Hamming space instead of L_1 space, then it approximates the compression distance. We construct the vectors in the Hamming space as vectors which take values from $\{0, 1\}$. They record 1 for a substring if that substring is present in the parsing of the string, and 0 otherwise. This is currently the best approximation known for this string edit distance. [CPṢV00] showed an $O(\log^2 n \log^* n)$ approximation which was improved to $O(\log n(\log^* n)^2)$ in [MṢ00]; here and in [CM02], we improve it modestly to $O(\log n \log^* n)$. This approach also allows us to apply many of the applications in this chapter to this compression distance as well as the edit distance with moves, as we will see later.

4.4.2 Unconstrained Deletes

In Definition 1.4.7 we described a particular edit distance which resembles the compression distance but additionally allows a deletion operation of an arbitrary substring at unit cost. We can show a result which relates the compression distance with unconstrained deletes, du, to the ESP transform.

Theorem 4.4.2 $\frac{1}{4}du(a, b) \leq |T(b)\backslash T(a)| \leq du(a, b) \cdot 4 \log n(\log^* n + 10)$

Proof. As usual, let n denote $\max(|a|, |b|)$, that is, the length of the longest string. The proof of this follows from our earlier results, with some variations. Again, we prove this in two parts, for the two bounds.

Lower bound

This we do again by construction. In fact, the proof is almost complete from previous results. We consider different possibilities: (1) the two strings are the same — If $b = a$ then $du(a, b) = 0$, and we need take no action. In all other cases then, $T(a) \neq T(b)$ then $|T(b)\backslash T(a)| \geq 1$ We can force this condition by adding a new symbol to the alphabet, \$, and rewriting the strings as $\$a\$$ and $\$b\$$. This does not affect the distance, but removes the case where b occurs as a node in the parsing of a. (2) b is a substring of a. In this case with at most two deletion operations, we can turn a into b. (3) All other cases. Using Lemma 4.4.1 we can construct the string ab from the string a using $3|T(b)\backslash T(a)|$ operations. We can then with one delete turn this into b. This extra operation is paid for by increasing the constant 3 to 4. Hence in all cases $du(a, b) \leq 4|T(b)\backslash T(a)|$.

Upper Bound

Again, most of the work has already been done for this. We observe in the upper bound of Theorem 4.4.1 that each permitted operation affects the symmetric difference by at most $8\log n(\log^* n + 10)$. This we observe arises from Theorem 4.3.1, which shows that each permitted operation affects $|T(b)\backslash T(a)|$ (and symmetrically, $|T(a)\backslash T(b)|$) by at most $4(\log^* n + 10)$. We need to make the same observation about the new operation, unconstrained deletes. This is fine: as with moves and copies, we consider long blocks whose centre is parsed by ESP. Following a delete of a substring, various nodes disappear from the parsing. However, although these nodes can affect $T(a)\backslash T(b)$ (since they were nodes that were present but are no longer), they do not directly contribute to $T(b)\backslash T(a)$, since this is interested in nodes that are in b but not in a. We only have to consider new nodes that are introduced. This happens following a deletion, since it can cause a local neighbourhood around the location of the deletion to be reparsed. However, as we have seen many times, surrounding the location of any edit operation, only a $O(\log^* n)$ region is reparsed. Hence, for this operation, only $2\log n \log^* n$ nodes can contribute to $|T(b)\backslash T(a)|$.

Putting these two parts together is sufficient to show that finding the set difference of the two sets $T(a)$ and $T(b)$ is sufficient to approximate the compression distance with unconstrained deletes, up to a $O(\log n \log^* n)$ factor. □

We will return to this result in Chapter 6, where it is necessary for some of the algorithms described there.

4.4.3 LZ Distance

The structure of the formation of the goal string in the lower bound of the above proof allows us to relate these distances to LZ distance and data compression. Notice that the target string is formed by copying substrings that either occur in the original string or occur earlier in the partly formed string. This fits exactly into the model of Lempel-Ziv compression, in which a compressed string is formed by copying substrings from a dictionary, if we treat the concatenation of the original string and the partly built string as defining the dictionary. Hence we can relate our LZ distance (Definition 1.4.5) to the other string edit distances.

Theorem 4.4.3 $LZ(a,b) \leq 3|T(b)\backslash T(a)| \leq O(\log n \log^* n)LZ(a,b)$

Proof. Note that, as observed above, the string ab can be formed from a using $3|T(b)\backslash T(a)|$ copy operations (from Theorem 4.4.2). This fits into the model of the LZ distance (in which b is built using copying operations only) and so $LZ(a,b) \leq 3|T(b)\backslash T(a)| \leq O(\log n \log^* n)du(a,b)$. But also compression distance with unconstrained deletes is a more powerful distance than LZ distance, and so $du(a,b) \leq LZ(a,b)$. Combining these gets the desired result. □

Corollary 4.4.1 $c(a,b) \leq LZ(a,b) + LZ(b,a) \leq O(\log n \log^* n)c(a,b)$

This follows since we can build ab using $LZ(a,b)$ copy operations, and then build b from ab using $LZ(b,a)$ uncopy operations.

Relation to Data Compression

We have already argued informally that both the LZ distance and the compression distance are closely related to notions of compression. The compression distance with unconstrained deletes counts how many quite general operations are required to describe a goal string in terms of another. In particular, suppose that the original string is the empty string. Adapting Theorem 4.4.1, we find that

$du(0, a) \leq 3||V(a)||_H = O(\log n \log^* n) \cdot du(0, a)$. This gives a statement about compressibility: suppose that we are allowed to describe a string using an arbitrary interleaving of move, copy, unconstrained deletes, and character inserts, deletes and changes. Then the minimum number of operations is $du(0, a)$, which we will write as $c(a)$.

An optimal Lempel-Ziv compressed form of a string can be found by repeatedly parsing the longest unparsed prefix of the string into a substring that has already been seen [ZL77, MŞ99]. This is at least as good as that produced by using $3|T(a)|$ copy operations, since this is a compression algorithm in the Lempel-Ziv framework. This then allows us to state a relation between "best possible compression", using very powerful operations of unrestricted copying, moving and deletion, and Lempel-Ziv, which is restricted to copying earlier substrings. Let $LZ(a)$ be the number of operations in the minimal Lempel-Ziv representation of a, then:

$$c(a) \leq LZ(a) \leq 3|T(a)| = O(\log n \log^* n)c(a)$$

In other words, Lempel-Ziv compression is competitive with unrestricted compression, up to a factor $O(\log n \log^* n)$.

4.4.4 Q-gram distance

Many previous works have used q-grams as methods of comparing texts, especially those from a database standpoint. We shall consider one formal definition in particular, that of Ukkonen [Ukk92]. A q-gram is simply a substring of length q. The q-grams of a string are all its substrings of length q. In [Ukk92], a similarity measure is defined in a similar way to our vector, $V(a)$. Here a vector, $G_q(a)$ is induced by the string a as recording the frequency of each q-gram v in a. The similarity of two strings is then given by $||G_q(a) - G_q(b)||_1$.

Clearly, this measure is symmetric, and it satisfies the triangle inequality. However, note that if $||G_q(a) - G_q(b)||_1 = 0$, it is not necessarily the case that $b = a$. Consider, for example, $b = W\mathsf{c}^q X \mathsf{c}^q Y \mathsf{c}^q Z$ and $a = W\mathsf{c}^q Y \mathsf{c}^q X \mathsf{c}^q Z$, where W, X, Y, Z are arbitrary strings, and c some character. Then $G_q(a) = G_q(b)$ and so this does not represent a metric: strings a and b can be quite dissimilar, while having a low (zero) distance as measured in this way. On the other hand, for non-adversarially constructed strings, this measure will be a good discriminator in many situations. It is easy to compute, and in particular it is trivial to construct sketches for this distance in the ordered streaming model. Hence q-grams have often been used in a variety of applications for string similarity.

From our perspective, there are several points of interest about q-grams. The first is to note that these form a very basic exact embedding of a string distance (that is, q-gram distance) into the L_1 distance. This can be computed easily in the ordered streaming model using the algorithms of Section 2.2.4, with additional $O(q)$ memory to store the last q characters. We can also cast our embeddings as being a justification of previous heuristic methods: we have shown that it is possible to use a method based on registering the number of substrings found by a particular parsing method to get a distortion bounded approximation of a string distance of interest. This is a post hoc justification of previous ad hoc approaches which have used counting the number of substrings found by a particular parsing (such as q-grams) to measure string similarity. It is also shown in [Ukk92] how to solve the approximate pattern matching problem under this distance measure. Using some observations on the structure of this distance, and with some careful use of data structures, it is shown that this problem can be solved in time $O(n \log(m - q) + m|\sigma|)$ for a text of length n and a pattern of length m. We should also compare our negative results on strings with q-grams — that these are insufficient to capture any distance metrics — with the results for permutations in Chapter 3. These showed that for several important permutation distances it is sufficient to consider 2-grams (q-grams of length 2), or similar pairs, to approximate them.

4.5 Solving the Approximate Pattern Matching Problem for String Edit Distance with Moves

In this section, we present an algorithm to solve the approximate pattern matching problem for string edit distance with moves. For any string a, we will assume that $V(a)$ can be stored in $O(|a|)$ space by listing only the non-zero components of $|a|$. More formally, we store $V(a)[x]$ if it is non-zero in a table indexed by $hash(x)$, and we store x as a pointer into a together with $|x|$.

The result below on pairwise string comparison follows immediately from Theorems 4.2.1 and 4.3.1 together with the observation that given $V(b)$ and $V(a)$, $||V(b) - V(a)||_1$ can be found in $O(|a|+|b|)$ time.

Theorem 4.5.1 *Given strings a and b with $n = \max(|a|, |b|)$, there exists a deterministic algorithm to approximate $d(a, b)$ accurate up to an $O(\log n \log^* n)$ factor in $O(n \log^* n)$ time with $O(n)$ space.*

4.5.1 Using the Pruning Lemma

In order to go on to solve the string edit distance problem, we need to "compare" a pattern p of length m against $t[i : n]$ for each i, and there are $O(n)$ such "comparisons" to be made. Further, we need to compute the distance between p and $t[i : k]$ for all possible $k \geq i$ in order to compute the best alignment starting at position i, which presents $O(n^2)$ subproblems in general. The classical dynamic programming algorithm for Levenshtein edit distance performs all necessary comparisons in a total of $O(mn)$ time in the worst case by using the dependence amongst the subproblems. Our algorithm will take a different approach. We make use of the Pruning Lemma, which is Lemma 1.5.1. This reduces the number of comparisons we have to make down to $O(n)$, although it introduces a further factor of 2 into the quality of the approximation. Hence, it prunes candidates away from the quadratic number of distance computations that a straightforward procedure would entail.

Still, we cannot directly apply Theorem 4.5.1 to compute $d(p, t[l : l + m - 1])$ for all l, because that will be expensive. It will be desirable to use the answer for $d(p, t[l : l + m - 1])$ to compute $d(p, t[l + 1 : l + m])$ more efficiently. In what follows, we will give a more general procedure that will help compute $d(p, t[l : l + m - 1])$ very fast for every l in order, by using further properties of ESP.

4.5.2 ESP subtrees

Given a string a and its corresponding ESP tree, $ET(a)$, we show that the subtree of $ET(a)$ induced by the substring $a[l : r]$ has the same edit-sensitive properties as the whole tree.

Definition 4.5.1 *Let $ET_i(a)_j$ be the jth node in level i of the parsing of a.*
We define an ESP Subtree *of a, $EST(a, l, r)$ as the subtree of $ET(a)$ containing the leaf nodes corresponding to $a[l]$ to $a[r]$, and all of their ancestors. Define $range(ET_i(a)_j)$ as the set of values $[p \ldots q]$ so that the leaf labels of the subtree rooted at $ET_i(a)_j$ correspond to the substring $a[p : q]$.*
Formally, we find all nodes of $ET_i(a)_j$ where $[l \ldots r] \cap range(ET_i(a)_j) \neq \emptyset$. The name of a node (substring) in $ET_i(a)_j$ is $hash(a[range(ET_i(a)_j) \cap [l \ldots r]])$.

This yields a proper subtree of $ET(a)$, since a node is included in the subtree if and only if at least one of its children is included (as the ranges of the children partition the range of the parent). As before, we can define a vector representation of this tree.

Definition 4.5.2 *Define $VS(a, l, r)$ as the characteristic vector of $EST(a)$ by analogy with $V(a)$, that is, $VS(a, l, r)[x]$ is the number of times the substring x is represented as a node in $EST(a, l, r)$.*

Note that $EST(a, 1, |a|) = ET(a)$, but in general it is not the case that $EST(a, l, r) = ET(a[l : r])$. However, $EST(a, l, r)$ shares the properties of the edit sensitive parsing. We can now state a theorem that is analogous to Theorem 4.3.1.

Theorem 4.5.2

$$d(a[l_p : r_p], b[l_q : r_q]) \leq 2||VS(b, l_p, r_p]) - VS(a, l_q, r_q])||_1 = O(\log n \log^* n) d(a[l_p : r_p], b[l_q : r_q])$$

Proof. Certainly, since Lemma 4.3.5 makes no assumptions about the structure of the tree, then the lower bound holds: the editing distance is no more than twice the size of the difference between the ESP subtrees.

For the upper bound, consider applying the necessary editing operations to the substrings of a and b. We study the effect on the original ESP trees, $ET(a)$ and $ET(b)$. Theorem 4.3.1 proved that each editing operation can cause a difference of at most $O(\log n \log^* n)$ between $V(a)$ and $V(a')$. It follows that the difference in $VS(a, l, r)$ must be bounded by the same amount: it is not possible that any more nodes are deleted or removed, since the nodes of $EST(a, l, r)$ are a subset of the nodes of $ET(a)$. Therefore, by the same reasoning as in Theorem 4.3.1, the total difference $||VS(b, l_p, r_p) - VS(a, l_q, r_q)||_1 = d(a[l_p : r_p], b[l_q : r_q]) \cdot O(\log n \log^* n)$. □

We need one final lemma before proceeding to build an algorithm to solve the String Edit Distance problem with moves.

Lemma 4.5.1 *$VS(a, l + 1, r + 1)$ can be computed from $VS(a, l, r))$ in time $O(\log |a|)$.*

Proof. Recall that a node is included in $EST(a, l, r)$ if and only if one of its children is. A leaf node corresponding to $a[i]$ is included if and only if $i \in [l \ldots r]$. This gives a simple procedure for finding $EST(a, l + 1, r + 1)$ from $EST(a, l, r)$, and so for finding $VS(a, l + 1, r + 1)$: (1) At the left hand end, let x be the node corresponding to $a[l]$ in $EST(a, l, r)$. We must remove x from $EST(a, l, r)$. We must also adjust every ancestor of x to ensure that their name is correct, and remove any ancestors which do not contain $a[l]$. (2) At the right hand end let y be the node corresponding to $a[r + 1]$ in $ET(a)$. We must add y to $EST(a, l, r)$, and set the parent of y to be its parent in $ET(a)$, adding any ancestor if it is not present. We then adjust every ancestor of y to ensure that their name is correct. Since in both cases we only consider ancestors of one leaf node, and the depth of the tree is $O(\log |a|)$, it follows that this procedure takes time $O(\log |a|)$. □

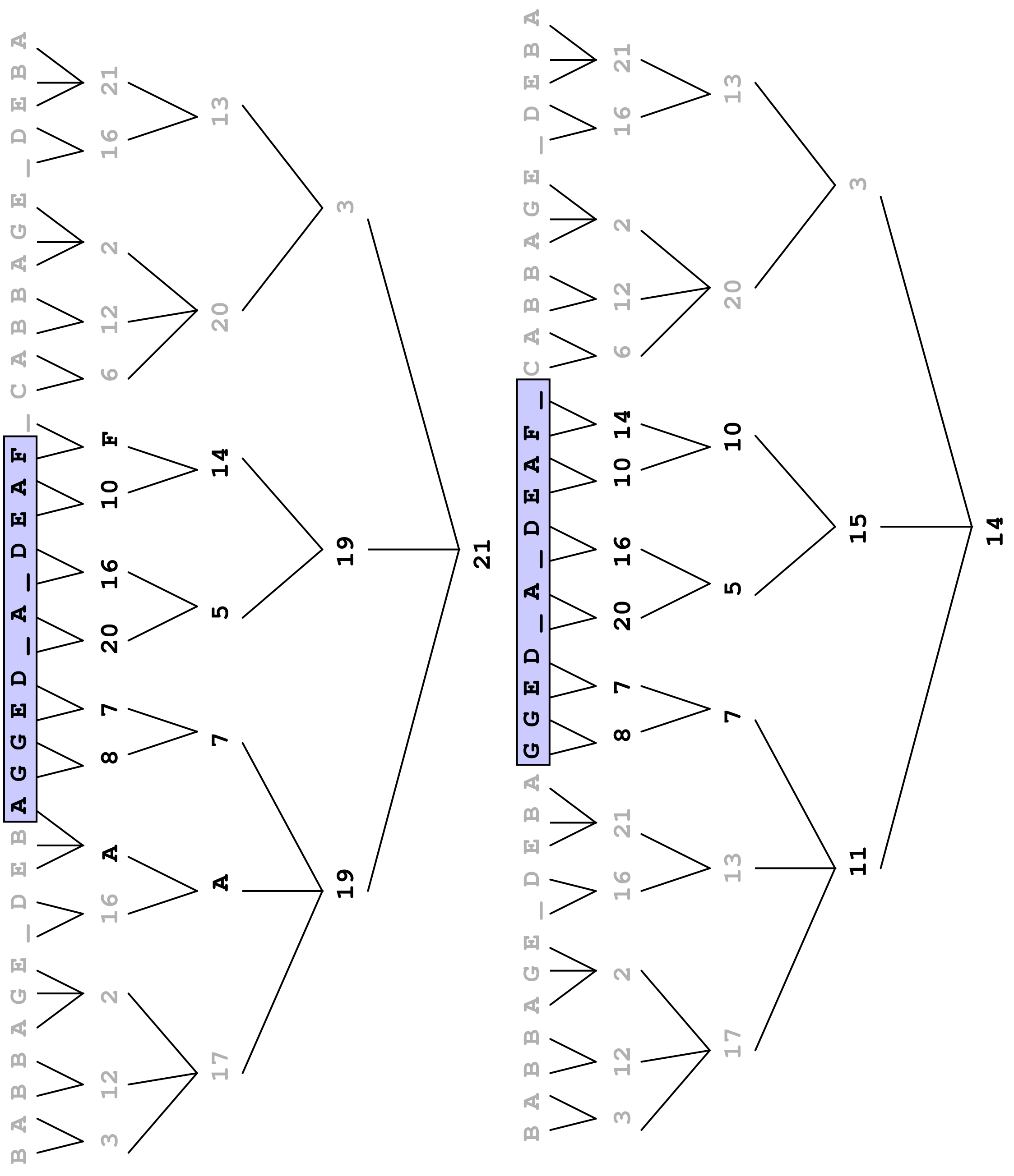
An ESP subtree, and this process of computing $VS(a, l + 1, r + 1)$ from $VS(a, l, r)$ is illustrated in Figure 4.9.

4.5.3 Approximate Pattern Matching Algorithm

Combining these results allows us to solve the main problem we study in this chapter.

Theorem 4.5.3 *Given text t and pattern p, we can solve the approximate pattern matching problem for the string edit distance with moves, that is, compute an $O(\log n \log^* n)$ approximation to $D[i] = \min_{i \leq k \leq n} d(p, t[i : k])$ for all i, in time $O(n \log n)$.*

Proof. Our algorithm is as follows: given pattern p of length m and text t of length n, we compute $ET(p)$ and $ET(t)$ in time $O(n \log^* n)$ as per Theorem 4.2.1. We then compute $EST(t, 1, m)$. This can be carried out in time at worst $O(n)$ since we have to perform a pre-order traversal of $ET(t)$ to discover which nodes are in $EST(t, 1, m)$. From this we can compute $\hat{D}[1] = ||VS(t, 1, m) - VS(p, 1, m)||_1$. We then iteratively compute $||VS(t, i + 1, i + m) - VS(p, 1, m)||_1$ from $||VS(t, i, i + m - 1) - VS(p, 1, m)||_1$


B A B B A G E _ D E B A G G E D _ A _ D E A F _ C A B B A G E _ D E B A
3 12 2 16 A 8 7 20 16 10 F 6 12 2 16 21
17 A 7 5 14 20 13
19 19 3
21
B A B B A G E _ D E B A G G E D _ A _ D E A F _ C A B B A G E _ D E B A
3 12 2 16 21 8 7 20 16 10 14 6 12 2 16 21
17 13 7 5 10 20 13
11 15 3
14

The top figure shows how an ESP subtree can be derived from a substring. Note how the names of nodes that only contain some of their children are different from their names normally. The lower figure then shows how the substring can be advanced by one position. We need to do only a constant amount of work at each level in the tree to update the names of the nodes that are included in the subtree. However, nodes internal to the subtree are not affected.

Figure 4.9: An ESP subtree can be iteratively computed

by using Lemma 4.5.1 to find which nodes to add to or remove from $EST(t, i, i+m-1)$ and adjusting the count of the difference appropriately. This takes n comparisons, each of which takes $O(\log n)$ time. By Theorem 4.5.2 and Lemma 1.5.1, $D[i] \leq \hat{D}[i] \leq O(\log n \log^* n)D[i]$. □

If $\log^* n$ is $O(\log m)$ as one would expect for any reasonable sized pattern and text, then a tighter analysis can show the running time to be $O(n \log m)$. This is because we only need to consider the lower $\log m$ levels of the parse trees; above this $EST(t, i, i+m-1)$ has only a single node in each level.

Relation to classical edit distance

So far we have not related this work to the problem of classical Levenshtein edit distance, in which substring moves are not allowed. However, there are several parts of this work which can be applied directly to this problem. Firstly, Lemma 1.5.1 can be applied to classical edit distance, suggesting a simplification that can be made for solving approximate pattern alignment under that distance. Secondly, although the lower bound of Theorem 4.3.1 requires the use of the move operation, the upper bound does not, and this holds in this case for the edit distance, so that $||T(b) - T(a)||_1 \leq O(\log n \log^* n)e(a, b)$. In fact, it is reasonable to suppose that for genuine data, using the ESP transform T would give good results for pairwise comparisons — that is, given three strings a, b, c then $e(a, c) \leq e(b, c) \iff ||T(c) - T(a)||_1 \leq ||T(c) - T(b)||_1$. This claim is borne out by experiments reported in [MṢ02]. However, in general it is possible to contrive counterexamples to this relationship.

4.6 Applications to Geometric Problems

The embedding of strings into vector spaces in an approximately distance preserving manner has many other applications directly, and with extensions. In this section, we will describe some of the important results we obtain. In contrast to our previous results, which have all been deterministic, many of these applications make use of randomised techniques.

4.6.1 Approximate Nearest and Furthest Neighbors

A fundamental open problem in string matching is that of approximate string indexing. Specifically, we are given a collection C of strings that may be pre-processed. Given a query string q, the goal is to find the string $c \in C$ closest to q under string edit distance, that is, $\forall x \in C : d(q, c) \leq d(q, x)$. This is precisely the Approximate Nearest Neighbors problem of Section 1.5.4 under a string distance.

Here, we focus on edit distance with moves, and let d denote this function. The approximate version of the nearest neighbors problem is to find $c \in C$ such that $\forall x \in C : d(q, c) \leq f \cdot d(q, x)$ where f is the factor of approximation. Let $m = |C|$ be the number of strings in the collection C, and n the length of the longest string in the collection. The challenge here is again to break the "curse" of high-dimensionality, and provide schemes with polynomial pre-processing whose query cost is $o(kn)$. That is, schemes which take less time to respond to queries than it takes to examine the whole collection.

We will make use of our embedding into L_1 distance and the results we saw in Section 2.4.1.

Theorem 4.6.1 *With polynomial time pre-processing of a collection C, queries for approximate nearest neighbors under edit distance with moves can be answered in time $O(nm^{1/2} \log n)$ finding a string from C that is an $O(\log n \log^* n)$ approximation of the nearest neighbor with constant probability.*

Proof. Essentially, we use the same basic idea that we used in Section 3.3.3. We use the existing algorithms for approximate nearest neighbors, and take care that our use of embeddings does not slow things down significantly. Firstly, we note that although we have described algorithms for Hamming distance in Section 2.4.1, we can apply these to L_1 distance by observing that each entry in our vector $V(a)$ is bounded by n, since we cannot have more than n copies of any node in a string of length n. So we can use the unary embedding of L_1 into Hamming space as outlined in Section 7. Since we are dealing with a large approximation factor inherent from the embedding into L_1 space, when choosing the parameter ϵ for the Approximate Nearest Neighbor algorithm, we shall fix this as a constant, say 1. This gives us the number of hash functions to compute as $m^{1/(1+\epsilon)} = m^{1/2}$.

As with our use of permutations, we need to take care that when applying the Locality Sensitive Hash functions, we are not trying to sample from the whole (exponentially large) vector, since most of these entries are zero — as stated before, only a linear number of entries in $V(a)$ are non-zero. So we use the same trick, we take a pass over the non-zero entries of the vector $V(a)$, held in some suitable data structure, and use a hash function to determine whether this entry would be sampled by the locality sensitive hash function. That is, in the precomputation phase, we first compute the locality sensitive hash function by picking m locations to sample, and store these in a table indexed by the name of that location. To compute this function for a new query requires a single pass over all the non-zero entries of $V(a)$: for each entry, we look it up in the table we have computed. If it is present, then we include this location in our computation of the hash function. Else, we do not. There are $O(n)$ names to look up, hence each of the $O(m^{1/2} \log n)$ hash functions is computed in time $O(n)$ (we assume a data structure supporting constant time look-ups). This absorbs the cost of computing the ESP parsing and embedding of the query string. □

Corollary 4.6.1 *With polynomial time pre-processing of a collection C, compression distance approximate nearest neighbors queries can be answered in time $O(nk^{1/2} \log n)$ finding a string from C that is an $O(\log n \log^* n)$ approximation of the nearest neighbor with constant probability.*

Proof. It is straightforward to switch the above discussion from edit distance with moves to compression distance: it is simply a matter of changing from using the L_1 embedding to using the (zero-based) Hamming distance embedding. So here, we do not need to use the unary embedding of L_1 into Hamming space, but can instead directly sample from $V(a)$ using locality sensitive hash functions on those entries that are non-zero. In all other respects, the algorithm is identical. □

The approach for Nearest Neighbors also applies to the Furthest Neighbors problem where, given a set of strings we must pre-process these so as to be able to rapidly find the furthest string in the collection from a query string.

Lemma 4.6.1 *A set D of k strings of length at most n can be pre-processed so given q with constant probability we find c satisfying $\forall x \in D : d(q, c) \geq d(q, x)/O(\log n \log^* n)$ in time $O(n \log k + \log^3 k)$.*

Proof. We will make use of the technique described in Section 2.4.2 which applies in L_2 space. So we first use a simple transformation to embed from L_1 space into L_2 space: we express our quantities in unary, recalling that no count can be greater than n. We can then use the method outlined in Theorem 2.2.1 to embed each transformed string from this unary Hamming space into L_2^2 space of dimensionality $\ell = O(\log k)$. The furthest neighbor results from Section 2.4.2 then immediately imply that an approximate $O(1)$-furthest neighbor in the target space can be found in time $O(\ell^2 \log k)$ per query following pre-processing of the data and query. The time to process the query string is just that to compute the embedding into L_1 space and then project this into L_2^2 space, which takes total time $O(n \log k)$. □

Again, the same technique can be used for Compression distance, by using the Hamming embedding directly rather than the L_1 distance embedding.

4.6.2 String Outliers

A problem of interest in string data mining is to find "outlier" strings, that is, those that differ substantially from the rest or from some other string. We study the case where we are given a query string, and we want to know if there is an outlier for this query. That is, if there is a string in the collection whose distance is very far from this query. For the purpose of this discussion, we shall consider a string to be an outlier for q if the distance is some constant fraction of the radius of the space (the greatest possible distance between two strings). Certainly, if two strings are unrelated then their distance is $\Omega(n)$. More formally, we are given a set D of k strings each of length at most n that may be pre-processed. Given a query string q, the goal is to find a string $s \in D$ such that $d(q,s) \geq \epsilon n$, for some constant fraction ϵ. We can strengthen our analysis of the embedding to show an improved result for this problem.

Lemma 4.6.2 *For strings a and b, let $n = \max(|a|,|b|)$. Then*

$$d(a,b) \leq 2||V(b) - V(a)||_1 = O(\log(n/d(a,b)) \log^* n) d(a,b)$$

Proof. We improve the upper bound by observing that in the top levels of the tree, there are only a limited number of nodes, so although these might change many times during a sequence of editing operations we always have the bound $||V_i(a) - V_i(b)||_1 \leq |a_i| + |b_i|$. A worst case argument says that the greatest number of nodes that could be affected is when one level in the tree is completely altered. The size of this level is $|a_i| + |b_i| \leq 8(\log^* n + 10)d(a,b)$, and the number of nodes above it in the tree (which may all be affected) is $\sum_{j \geq i} |a_i| + |b_i| \leq 16(\log^* n + 10)d(a,b)$. This follows since we have a tree where each internal node has at least two children. Below this level, we may assume that the worst case is when each edit operation contributes $8(\log^* n + 10)$ to $||V_j(a) - V_j(b)||$ for $j < i$. Thus, writing d for $d(a,b)$, we find $||V(a) - V(b)||_1 \leq d(\log n - \log(d \log^* n))8(\log^* n + 10) + 16d(\log^* n + 10) = O(d \log^* n \log(n/d \log^* n)) = O(d \log(n/d) \log^* n)$. □

We note that since the approximation depends on $\log n/d$, the quality of the approximation actually increases the less alike the strings are. This improved approximation helps in the outliers problem. To solve this problem, we again use the solution to approximate furthest neighbors.

Theorem 4.6.2 *We pre-process a set C of strings in time $O(kn \,\text{poly-log}(kn))$. For a query q, we either return an approximate outlier s or a null value. If returned, s will satisfy the property that $d(q,s) \geq \epsilon n / O(\log^* n)$ with constant probability and if t is an outlier for q in C, then $d(q,s) \geq d(q,t)/O(\log^* n)$; hence it is a $O(\log^* n)$ approximation. This requires time $O(n \log k + \log^3 k)$ per query. If no outlier is reported, then there is no outlier for q.*

Proof. This follows by using the above procedure for finding approximate furthest neighbors under string edit distances. We can reuse the algorithm for furthest neighbors, and analyse it using the improved bound of Lemma 4.6.2. Suppose the distance of the furthest neighbor is approximated as $\hat{d} \geq \epsilon n$. Then the true distance must be at least $\epsilon n / \log^* n$, and so s is returned as an outlier. On the other hand, if no string is a distance of at least $\epsilon n / \log^* n$, then the furthest neighbors procedure can return no string whose approximate distance from the query is $\geq \epsilon n$. So if no outlier is found this way, then there is no outlier. □

Once more, this result also applies to the compression distance with only minor alterations to the proof.

4.6.3 Sketches in the Streaming model

We consider the embedding of a string a into a vector space as before, but now suppose a is truly massive, too large to be contained in main memory. Instead, the string arrives as a stream of characters in order: $(s_1, s_2 \dots s_n)$. So here we are computing with an ordered stream (see Section 2.1.2). The result of our computations is a sketch vector for the string a.

Theorem 4.6.3 *A sketch $sk(V(a))$ can be computed in the streaming model to allow approximation of the string edit distance with moves using $O(\log n \log^* n)$ space. For an appropriate combining function f, then $d(a,b) \leq f(sk(V(b)), sk(V(a))) \leq O(\log n \log^* n) d(a,b)$ with probability $1 - \delta$. Each sketch is a vector of length $O(\log 1/\delta)$ that can be manipulated in time linear in its size. Sketch creation takes total time $O(n \log^* n \log 1/\delta)$.*

Proof. It is precisely the properties of ESP that ensure edit operations have only local effect on the parse structure that also allow us to process the stream with very little space requirements. Since Lemma 4.3.1 tells us that the parsing of any symbol at any level a_j depends only on at most $O(\log^* n)$ adjacent symbols, we only need to have these symbols in memory to make the parsing. This is true at every level in the parsing: only $O(\log^* n)$ nodes at each level need to be held in memory to make the parsing. When we group nodes of level i together to make a node of level $i+1$ we can conceptually “pass up” this new node to the next level in the process. Each node corresponds to addition of one to an entry in $V(a)$. However, we cannot store all the entries of the vector $V(a)$ without using linear space.

Instead, we can store a short summary of $V(a)$ which can be used as a surrogate for distance computations. We use our earlier techniques for stream computations to do this, in particular we use the result of Theorem 2.2.2 to make a sketch of $V(a)$ that will be good for approximating the L_1 distance between this vector and similarly formed sketches. These give a fixed probability $1-\delta$ for getting the answer within a factor of $1 \pm \epsilon$. Since we are already approximating up to large approximation factors, we may as well set ϵ to be some fixed constant here, hence the size of the sketch depends on $O(\log 1/\delta)$. The requirement for a naming function $hash()$ is solved by using Karp-Rabin signatures for the substrings (see Section 2.1.3). These have the useful property that the signature for a long substring can be computed from the signatures of its two or three component substrings (see Lemma 2.1.2). Thus, we can compute entries of $V(a)$ and so build a sketch of $V(a)$ using poly-logarithmic space. Since $V(a)$ can be used to approximate the string edit distance with moves up to a $O(\log n \log^* n)$ factor, it follows that these sketches achieve the same order of approximation. Overall, the total working space needed to create the parsing is $\log n$ levels each keeping information on $\log^* n$ nodes, totalling $O(\log n \log^* n)$ space. □

This type of computation on the data stream is tremendously useful in the case where the string is too large to be stored in memory, and so is held on secondary storage, or is communicated over a network. Sketches allow rapid comparison of strings: hence they can be used in many situations to allow approximate comparisons to be carried out probabilistically in time $O(\log 1/\delta)$ instead of the $O(n)$ time necessary to even inspect both strings. It allows distributed parties to compare their strings using only $O(\log n \log 1/\delta)$ bits of communication, a significantly sublinear amount.

4.6.4 Approximate p-centers problem

The p-centers problem asks us to process a set of k strings of length $O(n)$ and find p strings $C = c_1, \dots c_p$. The centers c_i define a partitioning of the strings into sets C_i where $a \in C_i \iff \forall j : d(a, c_i) \leq d(a, c_j)$. The set of centers C is chosen with the aim that $\text{diameter}(C) = \max_i \max_{a,b \in C_i} d(a,b)$ is minimised. This is another case where the improved bound of Lemma 4.6.2 can help.

Theorem 4.6.4 *A set of p-centers $\hat{C}$ can be found in time $O(k(p+n)\log n)$ so*

$$\forall C : \text{diameter}(\hat{C}) \leq O(\log n \log^* n)\, \text{diameter}(C)$$

Proof. We will use the algorithm of Gonzalez (see Section 2.4.3) which chooses the p-centers from the strings themselves. If the optimal solution is C_{opt} then let $d_{opt} = \text{diameter}(C_{opt})$. Firstly, we create for each string a short summarising sketch as described in Theorem 4.6.3. We set $\delta = O(1/n^2)$ to ensure that the probability of *any* distance comparison falling outside the approximation bounds is at most a small constant. The sketch construction takes time $O(kn \log n \log^* n)$. We next initialise the set of centers with an arbitrary input string. Then repeatedly add to the set the string that maximises the (approximate) distance to the set of centers, until there are p centers. Now consider the $(p+1)$st center, which is a distance d from the other centers. This means that there must be a cluster of diameter d and so $d \leq d_{opt}$. We may have been unlucky, and overestimated all the distances that caused us to pick out the p-centers. If this is the case, then the actual size of these clusters could be as high as $2d_{opt}O(\log n \log^* n)$, but no higher (from the approximation bounds of Lemma 4.6.2). Therefore, in this case the approximation is a factor of $O(\log n \log^* n)$ above the optimal, and we make $O(kp)$ comparisons of sketches. □

4.6.5 Dynamic Indexing

Thus far we have considered strings to be static immutable objects. The *Dynamic Indexing* problem is to maintain a data structure on a set of strings so that, under certain permitted editing operations on individual strings, we can rapidly compute the (approximate) distance between any pair of strings. A similar scenario was adopted in [MSU97] where the problem is to maintain strings under editing operations to rapidly answer equality queries: this is a special case of the general dynamic indexing problem we address. The technique was developed in [ABR00] for dynamic pattern matching: finding exact occurrences of one string in another.

Our situation is that we are given a collection of strings to pre-process. We are then given a sequence of requests to perform on the collection on-line. The requests are of the following types: (1) perform a string edit operation (inserts, deletes, changes, substring moves) on one of the strings (2) perform a split operation on one of the strings — split a string a into two new strings $a[1:i]$ and $a[i+1:|a|]$ for a parameter i. (3) perform a join operation on two strings — create a new string b from a_1 and a_2 as a_1a_2. (4) return an approximation to $d(a,b)$ for any two strings a and b in the collection. We consider a set of strings whose total length is n. For simplicity, we assume here that the operations of split and join are non-persistent — their input strings are lost following the operation.

Theorem 4.6.5 *Following $O(n \log n \log^* n)$ pre-processing time using $O(n \log n)$ storage, approximating the distance between two strings from the collection takes $O(\log n)$ time. This gives an $O(\log n \log^* n)$ approximation with constant probability. Edit operations upon strings of type (1),(2) or (3) above take $O(\log^2 n \log^* n)$ time each.*

Proof. For each string a in the collection, we shall maintain $ET(a)$, and for each node in $ET(a)$ we shall store a sketch of size $O(\log n)$ corresponding to the characteristic vector of the ESP subtree rooted at that node, and a Karp-Rabin signature for the substring corresponding to the concatenation of the leaves of that subtree. We show that each update operation (insertion, deletion, replacement, move, split, join) can be performed efficiently to generate a' from a while ensuring that the parse tree $ET(a)$ is correctly maintained, as well as the ESP subtrees at every node. We again make use of Theorem 4.3.1 and its proof, to show that any edit operation will require the reparsing of no more than $O(\log^* n)$ symbols for each level i in $ET(a)$.

For character inserts, deletes and changes, we only need to reparse an $O(\log^* n)$ region at each level in $ET(a)$. This follows from Lemma 4.3.1. The nodes can be changed, and the stored sketches and signatures updated using arithmetic operations on them. As noted in the proof of Theorem 4.3.1, moves can be handled by retaining the parsing of all but the fringes of the substring that is being moved. Using pointer manipulations, the move of a large substring can be dealt with using only $O(\log n)$ pointer changes. The fringes must be reparsed in the same way as with character operations, at a cost of $O(\log^* n)$ operations per level. Similarly, split and join operations can be handled by just reparsing the $O(\log^* n)$ symbols which are affected by the split or join at every level. We can update the sketches and signatures for each node as this is going on, since these can be manipulated arithmetically. This is a consequence of the composability of these sketches, as described in Theorem 2.2.3.

Requests of type (4) are handled by taking the sketches of the strings in question, and using these to generate the approximation of their distance. Since we can maintain a sketch for every string in a collection, it follows from Theorem 4.6.3 that we can approximate the distance between any pair accurate up to a factor of $O(\log n \log^* n)$. □

4.7 Discussion

This chapter has focussed on studying the main problems as applied to string distances. The main results shown here are:

- A hierarchical parsing of strings that parses substrings in a manner that is resilient to edit operations. This is an attempt to give as simple as possible a presentation of such parsings.
- The embedding of string edit distance with moves into the L_1 metric based on using the Edit Sensitive Parsing of strings, and the detailed proof of the approximation factor of the embeddings.
- Embeddings of related distances such as the Compression distance, and the Compression distance with unconstrained deletes, using the same Edit Sensitive Parsing, and similar proof techniques.
- A solution to Approximate Pattern Matching under string edit distance with moves, and solutions to geometric problems such as Approximate Nearest Neighbors for these string distances.

With only minor modifications, the techniques in this chapter can allow the distances being approximated to incorporate additional operations such as linear scalings, reversals, and similar block operations. However, the outstanding open problem is to understand the standard string edit distance matching problem (or quite simply computing the standard edit distance between two strings in the sketch or stream model) where substring moves are not allowed.

Chapter 5

Stables, Subtables and Streams

Bypasses are devices which allow some people to drive from point A to point B very fast whilst other people dash from point B to point A very fast. People living at point C, being a point directly in between, are often given to wonder what's so great about point A that so many people of point B are so keen to get there, and what's so great about point B that so many people of point A are so keen to get there. They often wish that people would just once and for all work out where the hell they wanted to be.

Mr Prosser wanted to be at point D. Point D wasn't anywhere in particular, it was just any convenient point a very long way from points A, B and C. He would have a nice little cottage at point D, with axes over the door, and spend a pleasant amount of time at point E, which would be the nearest pub to point D. His wife of course wanted climbing roses, but he wanted axes. He didn't know why — he just liked axes.

[Ada79]

5.1 Introduction

We describe an experimental study of some of the dimensionality reduction techniques described in Chapter 2. It is impractical and unnecessary to study every technique there, so instead we focus on applications of the methods for approximating vector L_p norm distances described in Section 2.2.4. Many of the results in other chapters rely on specifically these approximations, and so it is important to show that these work in practice as well as in theory. We consider two applications of these methods, to problems derived from real-world situations. In both cases, these require non-trivial extensions to the sketching methods. We will look at answering questions on massive data streams, to give information about a single stream and about the similarity of two separate streams. Here, the low memory overheads of sketches will be vital. We will also consider problems of data mining on massive tables of data. This gives further challenges for keeping only a manageable amount of additional information, and for doing an amount of work that is much smaller than that implied by the size of the tables.

We will first give outlines of the two situations, and then describe how the sketching procedures were implemented, with extensions to the sketching methods to address particular aspects of the application scenarios. We then report the results of some detailed experimental evaluation of the methods, and give comparisons to existing methods which demonstrate that these dimensionality-reducing embeddings are highly practical.

5.1.1 Data Stream Comparison

As observed in Section 2.1.2, massive streams of data that are too large to be stored in traditional databases are now routinely generated by many computer systems. Processing these streams seems to call for the support of special operations on the data. For example, one of the most basic tasks that arises in data stream processing is to *compare* different data streams, be they from different sources or over different time periods for the same source. Comparison of data streams reveals the structural relationship between them: whether two different streams represent similar underlying behaviour; which of a number of different historical streams does a new stream most resemble; how can a group of streams be organised into subgroups with similar characteristics.

We will consider streams that define vectors. These vectors are updated dynamically as the stream flows past. We will use sketches to compute the Hamming norm of a stream, and also to compare two streams based on their (vector) Hamming distance. We now go on to describe two situations where these computations will tell us valuable information about the streams and about pairs of streams.

Auditing Network Databases

Network managers view information from multiple data stream sources. Routers periodically send traffic information: traces of IP packets and IP flows (which are aggregated IP packet flows) [Net]; there are management routine updates: SNMP traps, card/interface/link status updates, route reachability via pings and other alarms [GKPV01]; configuration information: topology and various routing tables [BSW01]. Network managers need ways to take this continuous stream of diagnostic information and extract meaningful information. The infrastructure for collecting this information is often error-prone because of unreliable transfer (typically UDP and not TCP is used for data collection); network elements fail (links go down); configuration tables have errors; and data is incomplete (not all network elements might be configured to provide diagnostic data).

Continuous monitoring tools are needed to audit different data sources to ensure their integrity. This calls for "slicing and dicing" different data streams and corroborating them with alternative data sources. We give three examples of how this can be accomplished by computing the Hamming distance between data streams.

1. Let a_i be the number of *transit* IP packets sent from IP address i that enter a part of the network, and b_i be the number of IP packets that exit that part from i. We would like to determine the Hamming Distance to find out how many transit flows are losing packets within this part of the network.
2. There are published methods for constructing flows from IP packet traces. We can take traces of IP packets, aggregate them, and generate a flow log from this. This can then be compared with the flows generated by routers [Net, CGJS02], to spot discrepancies in the network.
3. Denial of Service attacks involve flooding a network with a large number of requests from spoofed IP addresses. Since these addresses are faked, responses are not acknowledged. So the Hamming difference between a vector of addresses which issued requests and which sent acknowledgements will be high in this situation [MVS01]. The Hamming norm of the difference between these two vectors provides a quick check for the presence of sustained Denial of Service attacks and other network abnormality and could be incorporated into network monitoring toolkits.

Maintaining distinct values in traditional databases

The Hamming norm of a stream[1] is of large interest in itself. It follows from Definition 1.2.7 that this quantity is precisely the number of distinct items in the stream. For example, let $\boldsymbol{a}$ be a stream representing any attribute of a given database, so a_i is the number of tuples in the database with value i in the attribute of interest. Computing the Hamming norm of $\boldsymbol{a}$ provides the number of distinct values of that attribute taken by the tuples. This is a foundational problem. We consider a traditional database table which is subject to a sequence of insertions and deletions of rows. It is of great importance to query optimisation and otherwise to know the number of distinct values that each attribute of the table assumes. The importance of this problem is highlighted by Charikar, Chaudhuri, Motwani and Narasayya [CCMN00]: "A principled choice of an execution plan by an optimiser heavily depends on the availability of statistical summaries like histograms and the number of distinct values in a column for the tables referenced in the query." Distinct values are also of importance in statistics and scientific computing (see [Gib01, GM99, HNSS95]). Unfortunately, it is provably impossible to approximate this statistic without looking at a large fraction of the database (such as via sampling) [CCMN00]. The sketch algorithm avoids this problem by *maintaining* the desired statistics under database updates, so that we never have to *compute* them from scratch.

There are many other potential applications of Hamming norm computation such as in database auditing and data cleaning. Data cleaning requires finding columns that are mostly similar [DJMS02]; Hamming norm of columns in a table can quickly identify such candidates, even if the rows are arranged in different orders. It is beyond the scope of this work to go into detail on all these applications, so we do not elaborate further on them.

5.1.2 Tabular Data Comparison

Tabular data sets are ubiquitous in data management applications. Traditional relations in database systems are tables. New applications also generate massive tabular datasets — consider the application of cellular telephone networks. These networks have base stations (cell towers) geographically distributed over the country, with each base station being responsible for handling the calls to and from a specific geographic region. Each such base station handles a vast number of calls over different

[1]To simplify the exposition, we will write "a norm of a stream" instead of "a norm of the implicit state vector of a stream".

time periods, and network operators keep statistics on the traffic volume at the base stations over time. This data may be represented by a table indexed by the latitude and longitude of the base station and storing the call volume for a period of time such as an hour.

As another application, consider the representation of the Internet traffic between IP hosts over time. For example, one may visualise a table indexed by destination IP host and discretised time representing the number of bytes of data forwarded at a router to the particular destination for each time period (since the current generation of IP routers route traffic based on destination IP address only, this is valuable information about network congestion and performance that IP routers routinely store and dump). In these network data management scenarios and others, massive tables are generated routinely (see [BCC$^+$00, BGR01] for other examples). While some of the data may be warehoused in traditional relational databases, this is seldom true in case of emerging applications. Here, tabular data is stored and processed in proprietary formats such as compressed flat files [Day]. Thus tabular data is emerging as a data format of independent interest and support within large scale applications [BCH99, BCC$^+$00, GGR00].

Of great interest in tabular data management is the task of *mining* the tables for interesting patterns. For example, in the examples above, the “knowledge” from tabular data analysis drives key network-management tasks such as traffic engineering and flexible capacity planning [FGL$^+$00]. In the cellular calls case, one may be interested in finding geographic regions where the call distribution is similar, for example, can one analyse the call volume patterns to separate dense urban areas, suburban areas, commuter regions, and so on? Can one correlate hours of day behaviour across different geographic regions with the time zones across the world? In the Internet traffic table, one may be interested in finding IP subnets that have similar traffic distribution across different time intervals to isolate web server farms, web crawler farms etc. In the telephone world, mining tabular data helps distinguish fax lines from voice lines [KSS99]. Thus many creative mining questions arise with tabular data. Our informal goal of mining tabular data can be instantiated in one of many standard ways: as clustering, as association rule mining or in other ways. These tasks involve comparing large (possibly arbitrary) portions of the table with each other (possibly many times).

The problem of efficient similarity computation between table portions is a challenge since tabular data is massive, generated at the rate of several terabytes a month in most applications [BCC$^+$00, BCH99, BGR01]. Tabular data becomes very large very quickly, since it grows with the product of its defining characteristics: an extra base station will take thousands of readings a day, and an extra day’s data adds hundreds of thousands of readings. As these databases grow larger, not only does the size of data stored increase, but also the size of the interesting table portions increases. With data storage capacities easily in the terabyte range, any subregions of interest to be “compared” (for example, cell call data for the Los Angeles versus San Francisco areas) can themselves be megabytes or even gigabytes in size. This means that previous assumptions that were commonly made in mining — that comparing two objects is a basic unit of computation — no longer hold. Instead, the metric by which algorithms are judged is no longer just the number of comparisons used, but rather the number of comparisons multiplied by the cost of each comparison.

Clustering is a good example of a mining task affected by growing table sizes. A typical clustering algorithm “compares” each “object” with others many times over, where each comparison involves examining the distance of one object to each of a number of others. Many good algorithms have been described in the literature already such as k-means [JD88], CLARANS [NH94], BIRCH [ZRL96, GGR00], DBSCAN [EKSX96] and CURE [GRS98]. These focus on limiting the number of comparisons. As we described earlier, when objects are large, the cost of comparisons (not just the number of comparisons) affects performance significantly. Orthogonal to the efforts mentioned above, we focus on reducing the cost of each comparison. Since what is important in clustering is often not the exact distances between objects, but rather which of a set of objects another object is closest to, and since known clustering algorithms are in their nature approximate and use randomness, we are able to show that using randomness and approximation in distance estimation does not sacrifice clustering quality by any appreciable amount. Vectors and matrices are routinely compared based on their L_p distances for $p = 1, 2$ or ∞, and more unusually other integral L_p distances [FY00]. Only recently has the similarity behaviour of L_p distances with non-integral p been examined for $0 < p \leq 2$, for example in [AHK01, CIKM02].

This work can be thought of as an application of dimensionality reduction techniques, which have been well explored in databases [AFS93, Fal96]. Previous dimensionality reduction techniques have used the first few components of the Discrete Fourier Transform, or other linear transformations (Discrete Cosine or Wavelet Transforms). This is because the L_2 distance between a pair of Discrete Fourier Transforms preserves the L_2 distance between the original data. However, the step of approximating the distance by using only the first components is a heuristic, based on the observation that for many sequences most of the energy of the signal is concentrated in these components. Although these techniques often work well for the Euclidean (L_2) distance, they do not work for other L_p distances, including the important L_1 distance. This is since there is no equivalent result relating the L_1 distance of sequences transformed in this way to that of the original sequences. Our solution to these problems is based on using the sketches for L_p distances described in Section 2.2.4.

5.2 Sketch Computation

We describe in detail the implementation of the vector sketching procedures based on using stable distributions. For this presentation, we will assume that item identifiers are integers.

The data stream is formed as a stream of tuples where the k'th tuple is $< i, d_k >$. This indicates that we should add the integer d_k to the count for item i. Clearly, we can accommodate subtractions by allowing d_k to be negative. We can then represent the current state represented by the stream as a vector, $\boldsymbol{a}$ such that $a_i = l$ means that over all tuples for item i the total of the d_k's is l. Update operations have the effect that $< i, d_k >$ causes $a_i \leftarrow a_i + d_k$.

5.2.1 Implementing Sketching Using Stable Distributions

We briefly recap the procedure for computing a sketch of a vector to allow the approximation of L_p norms, as described in Section 2.2.4. We compute values $r_{i,j}$ for all i, j where $0 \leq i \leq n$ and $0 \leq j \leq m$ (n is the dimension of the vector $\boldsymbol{a}$, and m the dimension of the sketch vector). Each $r_{i,j}$ is drawn independently from a random stable distribution with parameter p. Our update procedure on receiving tuple $< i, d_k >$ is as follows: we add d_k times $r_{i,j}$ to each entry j in the sketch. That is,

$$\forall 1 \leq j \leq m : sk(\boldsymbol{a})_j \leftarrow sk(\boldsymbol{a})_j + d_k r_{i,j}$$

As a result, at any point $sk(\boldsymbol{a})$ is the dot product $\boldsymbol{r} \cdot \boldsymbol{a}$ (treating $\boldsymbol{r}$ as the matrix of values $r_{i,j}$), so

$$sk(\boldsymbol{a})_j = \sum_{i=1}^{n} r_{i,j} a_i$$

It follows from the property of stable distributions that $\forall j : sk(\boldsymbol{a})_j$ is distributed as $||\boldsymbol{a}||_p X_j$ where X_j is a random variable with p-stable distribution. From this, we can use any $sk(\boldsymbol{a})_j$ to estimate the L_p norm multiplied by the median of the stable distribution with parameter p. We combine these values to get a good estimator for the L_p norm by taking the median of all entries $sk(\boldsymbol{a})_j$. By Theorem 2.2.3, this is an (ϵ, δ) approximation if m is $O(1/\epsilon^2 \log 1/\delta)$.

We need to show that this technique can be implemented in small space. So we do not wish to pre-compute and store all the values $r_{i,j}$. To do so would consume much more space than simply

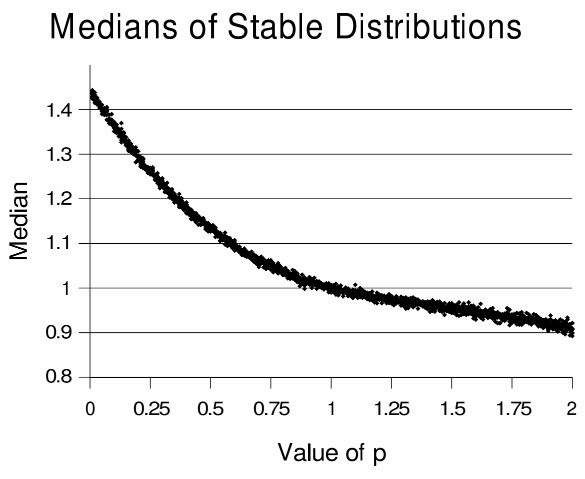


Figure 5.1: The value of the median of stable distributions with parameter $p < 2$

recording each a_i from which the number of distinct items could be easily found. Instead, we will generate $r_{i,j}$ when it is needed. Note that $r_{i,j}$ may be needed several times during the course of the algorithm, and must take the same value each time. We can achieve this by using pseudo-random generators for the generation of the values from the stable distributions. In other words, we will use standard techniques to generate a sequence of pseudo-random numbers from a seed value (see for example [PTVF92]). We will use i as a seed to generate $r_{i,1}$, and then use this stream of pseudo-random numbers $random()$ to generate $r_{i,2}, \ldots r_{i,m}$. This ensures that $r_{i,j}$ takes the same value each time it is used.

We also need to be able to generate values from a stable distribution with a very small stability parameter. This can be done using standard methods such as those described in [Nol]. These take uniform random numbers r_1, r_2 drawn from the range $[0 \ldots 1]$ and output a value drawn from a stable distribution with parameter p. We denote this transform as a (deterministic) function $stable(r_1, r_2, p)$. This function is defined as follows: first, we compute $\theta = \pi(r_1 - \frac{1}{2})$. Then

$$stable(1/2 + \theta/\pi, r_2, p) = \frac{\sin p\theta}{\cos^{1/p}\theta} \cdot \left(\frac{\cos(\theta - p\theta)}{-\ln r_2} \right)^{\frac{1-p}{p}}$$

5.2.2 Median of Stable Distributions

The result we find from our comparison is close to the L_p norm of vectors, multiplied by the median of absolute values from a stable distribution with parameter p. So to find the accurate answer, we need to scale this by an appropriate scaling factor. This follows from the fact that the median of a p-stable distribution is only 1 when $p = 1$ or $p = 2$ (Gaussian distribution). We do not know how to find this scaling factor analytically, so we find it empirically instead. In Figure 5.1 we show the results of using random simulation to find the median of stable distributions for different values of the stability parameter p. Each data point represents the median of 10,000 values chosen at random from a distribution with that parameter. We then take this to the power $1/p$, giving the scaling factor necessary for finding the $(L_p)^p$ distance — this is what we will want when we are using very small values of p to approximate the Hamming norm of a sequence. We repeated the experiment seven times for each value of p as a multiple of 0.01 less than 2. Note that there is a break at 2, where using the Gaussian distribution gives a scaling factor of exactly 1, while stable distributions generated by the transform

Algorithm 5.2.1 *Algorithm to compute sketches of a stream*

```
for $1 \leq i \leq m$ do
  $sk[i] \leftarrow 0.0$
for all tuples $(i, d_k)$ do
  random-init($i$)
  for $1 \leq j \leq m$ do
    $r_1 \leftarrow uniformrandom(0,1)$
    $r_2 \leftarrow uniformrandom(0,1)$
    $sk[j] \leftarrow sk[j] + d_k *$stable$(r_1, r_2, p)$
for $1 \leq j \leq m$ do
  $sk[j] \leftarrow$absolute$(sk[j])$
return median$(sk[j])/B(p)$
```

have a lower median. However the curve is unbroken around 1, where the scaling factor is also 1. For any given value of p, we can use the median of stable distributions for this parameter to scale our results accordingly. We will denote this by the function $B(p)$. This then gives the algorithm for maintaining a sketch for computing the sketches of a vector: see Algorithm 5.2.1.

As a side note, when we use the L_p sketches for clustering purpose only, we do not need to know $B(p)$. This is because, within the clustering algorithms, we do not need to know the value of the distance between any two items, but only the relative size of the distance (so, for example, whether some $\boldsymbol{c}$ is closer to $\boldsymbol{a}$ or to $\boldsymbol{b}$).

Counting Distinct Elements

It is straightforward to restrict our solution above to the problem of counting approximately the number of distinct elements. We will find the L_p norm of the stream, setting p to be small in accordance with Theorem 2.3.3. Initially, we have seen no items. So a_i is zero for all i, and $sk(\boldsymbol{a})_j = 0$ for all j. We then treat every insertion of an element i as a tuple of the form $(i, +1)$, and every deletion as a tuple $(i, -1)$. Following a number of operations, a_i will then represent the number of items i that have been seen. The Hamming norm allows us to find the number of distinct elements: provided for all i, $a_i \geq 0$, then the number of distinct elements is the number of i for which $a_i > 0$.

Note that the behaviour of the sketching algorithm following the removal of an item that has a zero count in $\boldsymbol{a}$ is well-defined. In Theorem 2.2.3, it is the absolute value of the entries a_i from the notional vector $\boldsymbol{a}$ that are found. Hence, if some entry in $\boldsymbol{a}$ is negative (indicating that there have been more removals of that item than insertions), then this will be counted as 1 towards the total of distinct elements. This is the result that we want in situations such as network monitoring and so on.

The ability to sum sketches also allows us to compute sketches in a distributed manner and then combine the results. Suppose that a stream is split into two parts: one observer sees stream $\boldsymbol{a}$, and the other sees $\boldsymbol{b}$. We wish to compute a sketch that is good for the merged stream, that is, the sketch that would be formed if one observer had seen both $\boldsymbol{a}$ and $\boldsymbol{b}$. The value of the underlying vector is clearly $\boldsymbol{a} + \boldsymbol{b}$, which by Observation 2.2.1 can be sketched as $sk(\boldsymbol{a}) + sk(\boldsymbol{b})$. This allows the computation of measurements on distributed data streams.

The theory has so far spoken in terms of vectors. However, it is conceptually simple to shift this theory from one-dimensional vectors to two-dimensional matrices, thanks to the nature of the L_p norms: we can think of any matrix as being represented by a vector that is linearised in some consistent way. It follows that we can replace a vector $\boldsymbol{a}$ with a matrix $\boldsymbol{A}$, and the above argument carries through.

Algorithm 5.2.2 *Algorithm to compute sketches for a table*

```
for $1 \le i \le k$ do
  for $0 \le m \le a-1$ do
    for $0 \le n \le b-1$ do
      $r_1 \leftarrow$ uniformrandom(0,1)
      $r_2 \leftarrow$ uniformrandom(0,1)
      $R[i,m,n] \leftarrow$ stable($r_1, r_2, p$)
for $1 \le i \le k$ do
  for $1 \le j \le x$ do
    for $1 \le l \le y$ do
      for $0 \le m \le a-1$ do
        for $0 \le n \le b-1$ do
          $sketch[i,j,l] \leftarrow sketch[i,j,l] + data[j+m, l+n] * R[i,m,n]$
```

The basic algorithm to compute sketches for all subtables of a table: suppose the table is size $x \times y$, and we want to make sketches of every subtable of size $a \times b$, then we form the dot product of each such subtable with k randomly created matrices.

5.2.3 Faster Sketch Computation

For a given vector or matrix, its sketch is a short real-valued vector as defined previously. Each entry in the sketch is the dot-product of the object (vector or matrix) with a number of randomly created objects (as specified above, using values from stable distributions), necessarily of the same size. When dealing with tabular data, we are interested in computing sketches for *all* subtables of some large set of tabular data. We describe this in two steps:

- We construct the sketch for all subtables of a fixed dimension (size) very fast using the Fast Fourier Transform.
- We choose a small, canonical set of dimensions and construct sketches for all subtables of that dimension. The pool of all such sketches is used to quickly compute the sketch of a subtable of any arbitrary dimension.

Computing sketches for all subtables of a fixed size.

We first focus on computing sketches with a *fixed* subtable size, and computing the sketch for all subtables of that size. The definitions above immediately translate into algorithms. A pre-processing phase can compute the necessary k different $\boldsymbol{R}[i]$ matrices from an appropriate stable distribution. Note that here we are not performing these computations in the stream (performing the clustering is an offline computation rather than an online one), and so we can afford the extra memory required to store the values of the random variables in memory for speed purposes. If so desired, it is possible to dispense with these, at the cost of longer computation times. Where we have tabular data, any subtable of fixed size defines a matrix that we would like to make a sketch of. Each sketch can then be computed by finding the dot product of each of the random matrices with all sub-rectangles of the tabular data. In our scenario, when we are dealing with tabular data, we could consider each sub-rectangle of fixed size in turn, and compute the sketches individually. Let the data have width x, height y, and the subtables of interest have size $a \times b$. The algorithm is outlined in Algorithm 5.2.2. If the total size of the input data is $N = xy$ and the sub-rectangle size is $M = ab$ then the total cost of this approach is $O(kMN)$ computations. This is because for each of the N locations in the data table we must multiply k different random matrices with a subtable of size M. This can be improved, since the basic computation is

Algorithm 5.2.3 *More efficient sketch computation using the Fast Fourier Transform*

```
for $1 \le i \le k$ do
  for $0 \le m \le a-1$ do
    for $0 \le n \le b-1$ do
      $r_1 \leftarrow$ uniformrandom(0,1)
      $r_2 \leftarrow$ uniformrandom(0,1)
      $R[i,m,n] \leftarrow$ stable($r_1, r_2, p$)
$data \leftarrow$ FFT($data$)
for $1 \le i \le k$ do
  $sketch[i] \leftarrow$ InverseFFT(Convolve(FFT($R[i]$),$data$))
```

simply the convolution of two matrices, which can be made more quickly using two-dimensional Fast Fourier Transforms (FFT).

Theorem 5.2.1 *All sketches of fixed size sub-rectangles of the data can be made efficiently using the Fast Fourier Transformation taking time $O(kN \log M)$.*

Proof. We firstly observe that to find the sketch of any sub-rectangle requires that we find the dot product of pre-computed random matrices with that sub-rectangle. If we find the sketches for all sub-rectangles, then we must find the dot product with every sub-rectangle. This is essentially the convolution of each of the random matrices with the data.[2] It is well known that n-dimensional convolutions can be done quickly using the n-dimensional Fourier transform, and so here we can find the 2-dimensional Fourier transform of the data and the matrices, and perform the convolution in the Fourier Domain. Careful use of Fourier Transforms then reduces the time complexity of sketching to $O(kN \log M)$ [PTVF92]. This technique is outlined in Algorithm 5.2.3. □

This is a generalisation of the technique in [IKM00] for time series data on one dimension to tabular data. A further improvement that can be made is to compute the sketches in parallel. Note that each component of the sketches is independent from the others. Therefore, provided parallel access to the data is available, the job of computing sketches can be divided up between multiple processors. Alternatively, the data itself can be divided between multiple computing nodes, and sketches for each subsection computed separately (however, these sections must overlap so all subtables can be sketched, and so the total amount of work required increases slightly).

Canonical sizes and combining sketches

Next we consider computing sketches for many different sizes. Once we have all sketches for sub-rectangles of a particular size, we can combine these to make a sketch for a rectangle up to twice the size in either dimension. Suppose that four independent sets of sketches, $sk^p_w, sk^p_x, sk^p_y, sk^p_z$, have been computed for sub-rectangles of dimensions $a \times b$. Provided $a \le c \le 2a$ and $b \le d \le 2b$, a sketch can be made for the sub-rectangle of size $c \times d$. Let the data be a large table, Z, so the sketch, $sk(Z[i,j])$ will cover the sub-rectangle from $Z[i,j]$ to $Z[i+a, j+b]$.

Definition 5.2.1 *A compound sketch, sk'^p can be formed as follows:*

$$sk'^p(Z[i,j]) = sk^p_w(Z[i,j]) + sk^p_x(Z[i+c-a,j] + sk^p_y(Z[i,j+d-b]) + sk^p_z(Z[i+c-a,j+d-b])$$

[2]Technically, the convolution finds the dot product with the smaller matrix flipped along its horizontal and vertical axes. But since each entry of the random matrices is made in the same way — by drawing from a random distribution — then it doesn't matter if the matrix is notionally upside down and back to front, because it still has the same properties.

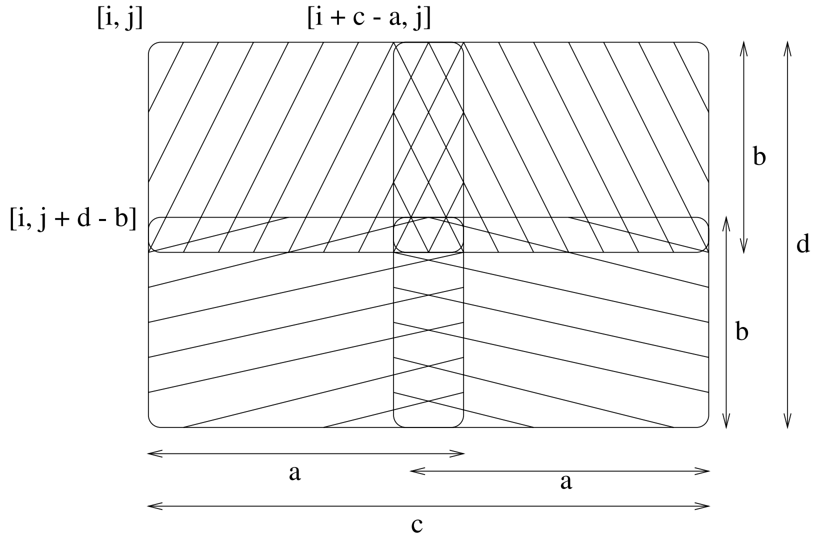


Figure 5.2: Compound sketches can be formed from four sketches that represent the area required.

In effect, the new sketch is formed by tiling the required sub-rectangle with sketches of overlapping rectangles. This is shown in Figure 5.2. From this definition, we state the theorem that shows that this compound sketch still has the desirable properties of a sketch.

Theorem 5.2.2 *For two compound sketches created as above, with probability* $1 - \delta'$*, we have*

$$||\boldsymbol{A} - \boldsymbol{B}||_p(1-\epsilon) \leq B(p)\ \text{median}(|sk'^p(\boldsymbol{A}) - sk'^p(\boldsymbol{B})|) \leq 4(1+\epsilon)||\boldsymbol{A} - \boldsymbol{B}||_p$$

Proof. Recall that B is our scaling factor to make the approximation of the L_p distance. We consider the two extremes: firstly, when $2a = c$ and $2b = d$. Then with probability at least $1 - 4\delta$ we have

$$(1-\epsilon)||\boldsymbol{A} - \boldsymbol{B}||_p \leq B(p)\,\text{median}(sk^p_w(\boldsymbol{A}[0,0]) + sk^p_x(\boldsymbol{A}[a,0]) + sk^p_y(\boldsymbol{A}[b,0]) + sk_z u^p(\boldsymbol{A}[a,b])$$
$$-sk^p_w(\boldsymbol{B}[0,0]) - sk^p_x(\boldsymbol{B}[a,0]) - sk^p_y(\boldsymbol{B}[b,0]) - sk^p_z(\boldsymbol{B}[a,b]))$$

giving the lower bound since these four regions do not overlap, and hence have the same properties as one sketch covering the whole area. The other extreme occurs when $a = c$ and $b = d$, where with probability at least $1 - 4\delta$

$$B(p)\,\text{median}(sk^p_w(\boldsymbol{A}) + sk^p_x(\boldsymbol{A})sk^p_y(\boldsymbol{A}) + sk^p_z(\boldsymbol{A})$$
$$-sk^p_w(\boldsymbol{B}) - sk^p_x(\boldsymbol{B}) - sk^p_y(\boldsymbol{B}) - sk^p_z(\boldsymbol{B})) \leq 4(1+\epsilon)||\boldsymbol{A} - \boldsymbol{B}||_p$$

for the upper bound. □

For intermediate situations, we can consider the arrangement in Figure 5.2. We assume that each of the four sketches estimates the L_p distance accurately up to a factor of ϵ. Then where all four sketches overlap, the contribution to the L_p distance is counted four times. Hence, the L_p distance could be as much as $4(1+\epsilon)$ times the true value. However, this worst case only occurs when the majority of the difference between two subtables is concentrated in this area of overlap. In practice, we would expect the differences to be spread out within the whole subtable, hence the accuracy would be better. Recall that for clustering purposes, it is not the absolute accuracy that is important, but the relative sizes of the errors, as we are comparing one object to some others to see which it is closest to.

Because of this theorem, then by summing four sketches component-wise we can make reasonable sketches from those for smaller overlapping subtables. Therefore we choose a *canonical* collection of sizes: we fix sizes $2^i \times 2^j$ for all integer values of i and j up to the logarithm of the dimensions of the data and compute sketches for all matrices of size $2^i \times 2^j$ using Theorem 5.2.1. Following that we can compute the sketches for any $c \times d$ sized subtable using these sketches and using Theorem 5.2.2. Therefore, we conclude

Theorem 5.2.3 *Given any tabular data of size $N = n \times n$, in time $O(kN \log^3 N)$ we can compute all the $O(\log^2 N)$ sets of sketches corresponding to the canonical sizes. Following that, the sketch for any subtable can be computed in time $O(k)$. For L_p distances of our interest, $k = O(\frac{\log(1/\delta)}{\epsilon^2})$, and each sketch is an $4 + \epsilon$ approximation with probability at least $1 - \delta$.*

Note that there are $O(N^2)$ subtables, and computing sketches for some of these requires $O(N)$ time. However, the procedure above produces sketches for them significantly more efficiently than the time it takes to compute any large subset of such sketches.

All the discussions above can be generalised to tables with more orthogonal components than two; the issues are simple if a small constant number of orthogonal components are involved. This applies for example to the cellular data case which is indexed by the latitude, longitude as well as time period.

5.2.4 Implementation Issues

For computing with stable distributions, we implemented the method of [Nol] to generate stable distributions with arbitrary stability parameters. This uses a method which is similar to the Box-Muller procedure for generating values from Normal distributions. A transformation takes two values drawn from a uniform $[0, 1]$ distribution and outputs a value drawn from the appropriate stable distribution. For computing the Hamming norm, we set the stability parameter p of the stable distribution to be as low as possible. However, as p gets closer to zero, the values generated from stable distributions get significantly larger, gradually exceeding the range of floating point representation. Through experimentation, the smallest value of p that did not generate floating point overflow was found to be 0.02. Hence we set p to this value, and empirically found the median of the stable distribution as generated by this procedure. This median is used in the scaling of the result to get the approximation of the Hamming norm. Note that using $p = 0.02$ means that, if one element occurs a million times, then the contribution to the Hamming norm will be $(10^6)^{0.02} = 1.318$, so this gives a worst case overestimate of 32%. This could be a large source of error, although even this level of error is likely to be acceptable for many applications. In fact we shall see that most of our experiments show an error of much less than 10%

Computing the approximate distance using sketches requires finding the median of a set of numbers. There are many ways to find the median: the simplest is to sort the numbers, then read out the middle value. This approach is somewhat wasteful, since it is not necessary to have the values sorted. At the other extreme, there are methods that are guaranteed to to take a linear amount of time; but in practice, the constant factors make this expensive. Instead, we adopted the randomised procedure which is expected to find the median in linear time (described in [CLR90] and [PTVF92] amongst others). We are confident in adopting this procedure because (1) the data being searched is generated based on randomly chosen values and so is unlikely to create the "bad cases" that can lead to excessive running time and (2) because many thousands of comparisons may be made in the course of the execution of an algorithms, the total time spent finding medians will tend towards the expected (linear) cost.

Clustering Using the k-means Algorithm

Algorithm 5.2.4 *The k-means algorithm*

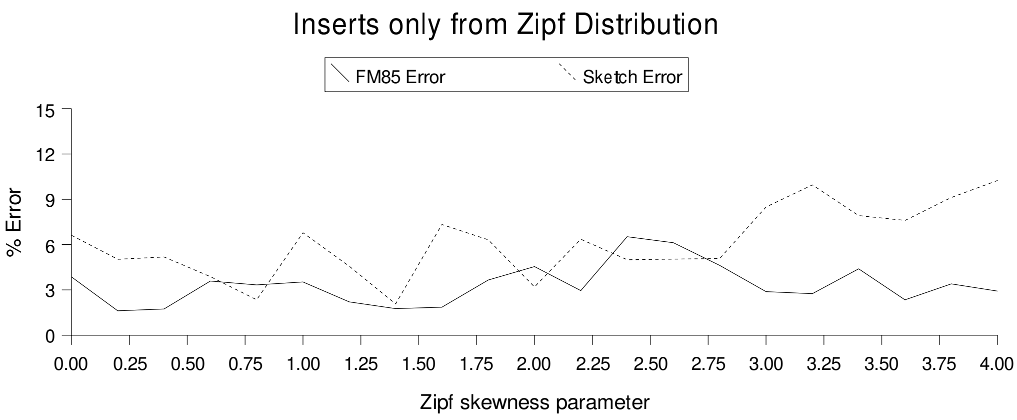
```
for $1 \leq i \leq k$ do
  $kmeans[i] \leftarrow$ random item from data
  $centroid[i] \leftarrow 0$
  $count[i] \leftarrow 0$
repeat
  for all item $\in$ items do
    $mindist \leftarrow 1$
    for $1 \leq i \leq k$ do
      if $dist(item, kmeans[i]) < dist(item, kmeans[mindist])$ then
        $mindist \leftarrow i$
    $cluster[item] \leftarrow mindist$
    $centroid[mindist] \leftarrow centroid[mindist] + item$
    $count[mindist] \leftarrow count[mindist] + 1$
  for $1 \leq i \leq k$ do
    $kmeans[i] \leftarrow centroid[i]/count[i]$
    $centroid[i] \leftarrow 0$
    $count[i] \leftarrow 0$;
until no items reclassified or repetition count exceeded
each item is now classified by $cluster[item]$
```

The use of approximate distance computations is tested by adapting the k-means algorithm. This algorithm is a simple way of generating a clustering with k clusters. The k-means are initialised to be k randomly chosen items from those to be clustered. Each iteration allocates each of the items to be clustered to the mean that it is closest to. Then a set of new means is generated as the average of all the items in each cluster. This procedure is iterated until the clustering is unchanged, or an iteration limit is reached. This is shown in Algorithm 5.2.4. Further details of this algorithm are described in [JD88]. We can apply this algorithm with either exact distance computations or approximate distance computations.

5.3 Stream Based Experiments

We used our method of sketches formed with stable distributions to approximate the Hamming norm of a stream and Hamming distance between streams. We also implemented Probabilistic Counting as described in Section 2.3.2 for approximating the number of distinct elements, since this is the method that comes closest to being able to compute the Hamming norm of a sequence.

Experiments were run on a Sun Enterprise Server on one of its UltraSparc 400MHz processors. To test our methods, we used a mixture of synthetic data generated by random statistical distributions, and real data from network monitoring tools. For this, we obtained 26Mb of streaming NetFlow data [Net], from a major network. We performed a series of experiments, firstly to compare the accuracy of using sketches against existing methods for counting the number of distinct elements. We started by comparing our approach with the probabilistic counting algorithm for the insertions-only case (when there are no deletions). We then investigated the problem for sequences, where both insertions and deletions were allowed. Next we ran experiments on the more general situations presented by network data with sequences where entries in the implicit vectors are allowed to be negative. As mentioned earlier, probabilistic counting techniques fail dramatically when presented with this situation. Finally, we ran experiments for computing the Hamming distance between network data streams and on the union of multiple data streams. We used only the stable distributions method, since this is the only



The relative error of both methods for counting the number of distinct elements as generated by Zipf distributions with varying levels of skewness.

Figure 5.3: Results for insertions based on a Zipf distribution

method which is able to solve this problem using very small working space in the data streams model.

For our experiments, the main measurement that we gathered is how close the approximation was to the correct value. This was done by using exact methods to find the correct answer ($exact$), and then comparing this to the approximation ($approx$). Then a percentage error was calculated simply as $(\max(exact, approx)/\min(exact, approx) - 1) \times 100\%$.

Timing Issues

Under our initial implementation, the time cost of using stable distributions against using probabilistic counting was quite high — a factor of about six or seven times, although still only a few milliseconds per item. The time can be much reduced at the cost of some extra space usage. This is due to the fact that the majority of the processing time is in creating the values of the stable distribution using a transformation from uniform random variables. By creating a look-up table for computing a stable distribution with a fixed stability parameter we could avoid this processing time. This table is indexed by values from a uniform distribution, and a value interpolated linearly from the nearby values in the table. The extra space cost could be shared, if there are multiple concurrent computations to find the Hamming norm of several different data streams (for example, multiple attributes in a database table). This approach would also be suitable for use in embedded monitoring systems, since the method only requires simple arithmetic operations and a small amount of writable memory. However, we do not analyse the quality of the results produced by applying this heuristic. A compromise solution is to use a cache to store the random variables corresponding to some recently encountered attribute values. If there is locality in the attribute values then the number of cache misses will be small.

Insertions Only with Synthetic Data

We tested the two algorithms on synthetic data generated from a Zipf distribution with varying levels of skewness. The results are shown in Figure 5.3. We used sketches that were vectors with 512 entries, against Flajolet-Martin probabilistic counting given the equivalent amount of working space. 100,000 elements were generated from Zipf distributions with parameters ranging from 0 (uniform distribution) up to 4 (highly skewed). The variation in skew tests the ability of the methods to cope with differing

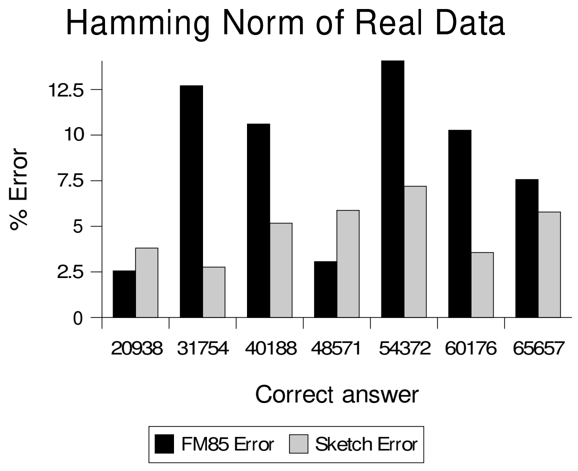


Figure 5.4: Finding the Hamming Norm of Network Data

numbers of distinct items, since low skew generates many distinct items, whereas high skew gives only a few.

Both algorithms produce answers that in most cases are within a few percentage points of the true answer. The worst case is still less than 10% away from the correct answer. This should be compared against sampling based methods as reported in [Gib01], where the error ratio was frequently in excess of 200%. This shows that for comparable amounts of space usage, the two methods are competitive with each other for counting distinct items. The worst case for stable distributions occurs when the number of distinct elements is very low (high skew), and here exact methods could be used with small additional space requirements.

Hamming Norm of Network Data

We examined finding the Hamming norm of the sequence of IP addresses, to find out how many distinct hosts were active. We used exact methods to be able to find the error ratio. We increased the number of items examined from the stream, and looked at between 100,000 and 700,000 items in 100,000 increments. The results presented in Figure 5.4 show that there were between 20,000 and 70,000 distinct IP addresses in the stream. Both probabilistic counting and sketches were used, given a workspace of 8Kb. Again, using sketches is highly competitive with probabilistic counting, and is on the whole more reliable, with an expected error of close to 5% against probabilistic counting, which is nearer to 10%.

Streams based on sequences of inserts and deletes

Our second experiment tested how the methods worked on more dynamic data, with a mixture of insertions and deletions. It also tested how much they depend on the amount of working space. We created a sequence of insertions and deletions of items, to simulate addition and removal of records from a database table. This was done by inserting an element with one probability, p_1, and removing an element with probability p_2, while ensuring that for each element i, the number of such elements seen was never less than zero. Again, 100,000 transactions were carried out to test the implementation.

We ran a sequence of experiments, varying the amount of working space allocated to the counting programs, from 0.5Kb, up to 32Kb. The results are shown in Figure 5.5. The first observation is that the results outdo what we would expect from our theoretical limits from Theorem 2.2.3. Even with only 1Kb of working space, the sketching procedure using stable distributions was able to compute a

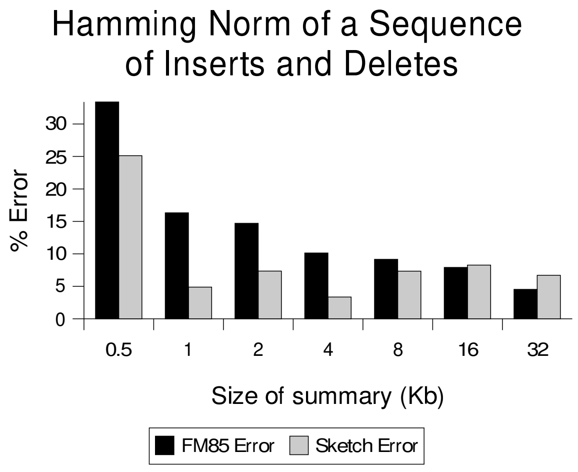


Figure 5.5: Testing performance for a sequence of insertions and deletions

very accurate approximation, correct to within a few percentage points. It is important to note that L_0 sketches were able to nearly equal or better the fidelity of probabilistic counting in every case, and also offer additional functionality (of being able to cope with negative values, and capable of being used to find the difference between streams). There is no immediate explanation for why there is a clear trend of improvement with additional space, but certainly, with a workspace of only a few kilobytes, we can be sure of a result which is highly likely to be within a few percentage points of the correct answer. This is more than good enough for most of the applications we have already mentioned.

Hamming norm of unrestricted streams

We generated a set of synthetic data to test the method's performance on the more general problems presented by Network data. The main purpose of the next experiment was to highlight that existing methods are unable to cope with many data sets. Zipf distributions with a variety of skewness parameters were used. Additionally, when a value was generated, with probability $\frac{1}{2}$ the transaction is an insertion of that element, and with probability $\frac{1}{2}$ it is a deletion of that element. The results are presented in Figure 5.6. When we compare the results of probabilistic counting on this sequence to the Hamming norm of the induced vector, we see massive disparity. The error fraction ranges from 20% to 400% depending on the skew of the distribution, and it is the uniform distribution on which this procedure performs the worst. This is because the induced vector includes negative values, which can occur when there are more deletions than insertions (for example, if we are only seeing part of the transaction stream, then we may see part which is mostly deletions). Probabilistic Counting is unable to cope with negative values, and hence its approximations of the Hamming norm are very far off. On the other hand, using stable distributions gives a result which is consistently close to the correct answer, and in the same region of error as the previous experiments. Probabilistic counting is only competitive at computing the Hamming norm for distributions of high skew. This is when the number of non-zero entries is low (less than 100), and so it could be computed exactly without difficulty.

Hamming Distance between Network Streams

Our second experiment on network data was to investigate finding the Hamming distance (dissimilarity) between two streams. This we did by construction, to observe the effect as the Hamming distance

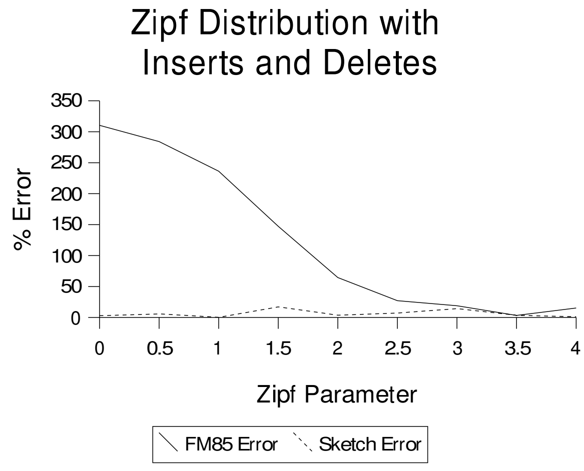


Figure 5.6: Testing performance for a sequence of insertions

increased. We fixed one stream, then constructed a new stream by merging this stream with a second. With probability p we took item i from the first stream, and with probability $1 - p$ we took item i from the second for 100,000 items. We then created sketches for both streams and found their difference using sketches of size 8Kb. The results are shown in the chart in Figure 5.7 as we varied the parameter p from 0.1 to 0.9. Here, it was not possible to compare to existing approximation methods, since no other method is able to find the Hamming distance between streams.

The performance of sketching shows high fidelity. Here, the answers are all within 7% of the true answer, and the trend is for the quality to improve as the size of the Hamming distance between the streams increases. This is because the worst performance of sketches for this problem is observed when the number of different items is low (when the norm is low). Hence sketches are shown to be good tools for this network monitoring task.

5.4 Experimental Results for Clustering

In analysing the performance of sketch-based methods, we wish to evaluate the performance of sketch construction as well as sketch accuracy. In the context of mining, we wish to evaluate the performance of sketching applied to clustering, when compared to other traditional (exact) methods as well as the utility of various L_p norms combined with clustering. To facilitate such assessment, we first define various measures of accuracy. Then we describe the data sets we used, and show experiments that demonstrate the speed and accuracy of sketches. Finally, we apply these sketches to clustering algorithms; here, using L_p for fractional p, we show that it can reveal interesting patterns both visually and quantitatively.

Data Sets

Real data sets obtained from AT&T were used in our experiments. From these we projected a tabular array of values reflecting the call volume in the AT&T network. The data sets give the number of calls collected in intervals of 10 minutes over the day (x-axis) from approximately 20,000 collection stations allocated over the United States spatially ordered based on a mapping of zip code (y-axis). This effectively creates a tabular dataset of approximately 34Mb for each day. We stitched consecutive

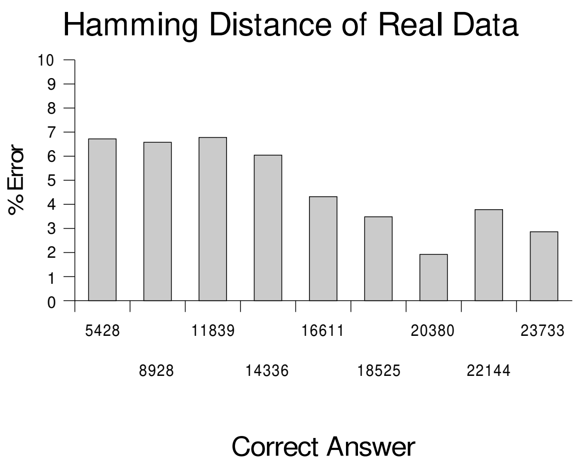


Figure 5.7: Measuring the Hamming distance between network streams

days to obtain data sets of various sizes.

For a separate set of experiments, we constructed synthetic tabular data (128Mb) to test the value of varying the distance parameter p in searching for a known clustering. We divided this dataset into six areas representing $\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}$ and $\frac{1}{16}$ of the data respectively. Each of these pieces was then filled in to mimic six distinct patterns: the values were chosen from random uniform distributions with distinct means in the range 10,000 – 30,000. We then changed about 1% of these values at random to be relatively large or small values that were still plausible (so should not be removed by a pre-filtering stage). We divide the data into small rectangles, which we refer to as tiles. Under any sensible clustering scheme, all tiles in areas created from the same distribution should be grouped together in the clustering. These experiments were run on an UltraSparc 400MHz processor with 1.5Gbytes of memory.

5.4.1 Accuracy Measures

Let the sketched L_p distance between two matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ be denoted by $||\widetilde{\boldsymbol{A} - \boldsymbol{B}}||_p$. We use the following measures to assess sketching accuracy:

Definition 5.4.1 *The* cumulative correctness *of a set of k separate experiments between a set of matrices $\boldsymbol{A}_i$ and $\boldsymbol{B}_i$ is*

$$\frac{\sum_{i=1}^{k} ||\widetilde{\boldsymbol{A}_i - \boldsymbol{B}_i}||_p}{\sum_{i=1}^{k} ||\boldsymbol{A}_i - \boldsymbol{B}_i||_p}$$

This measure gives an idea of, in the long run, how accurate the sketches are. This measure is forgiving towards variations if they are balanced out, so we next introduce a measure to look at the average variation of each comparison.

Definition 5.4.2 *The* average correctness *of a set of k experiments is*

$$1 - \frac{1}{k} \sum_{i=1}^{k} \left| 1 - \frac{||\widetilde{\boldsymbol{A}_i - \boldsymbol{B}_i}||_p}{||\boldsymbol{A}_i - \boldsymbol{B}_i||_p} \right|$$

The above two measures are good for when we are testing the sketches as estimators of the actual distance. In our clustering application however, what is more important is the correctness of pairwise comparisons: testing whether some object $\boldsymbol{c}$ is closer to $\boldsymbol{a}$ or to $\boldsymbol{b}$. The next measure evaluates, out of a number of experiments of this type, which fraction gives the correct answer (ideally, this would be 100%).

Definition 5.4.3 *The* pairwise comparison correctness *of k experiments between three sets of matrices, $\boldsymbol{A}_i, \boldsymbol{B}_i, \boldsymbol{C}_i$ is*

$$\frac{\sum_{i=1}^{k} \mathrm{xor}(||\boldsymbol{C}_i - \boldsymbol{A}_i||_p < ||\boldsymbol{C}_i - \boldsymbol{B}_i||_p, ||\widetilde{\boldsymbol{C}_i - \boldsymbol{A}_i}||_p > ||\widetilde{\boldsymbol{C}_i - \boldsymbol{B}_i}||_p)}{k}$$

To assess the quality of a clustering obtained using exact methods against one obtained using sketches, we want to find ways of comparing the two k-clusterings. A commonly used construct in comparing clusterings is the *confusion matrix*. Given two k-clusterings of a large number of items, each object will be associated with two clusters: one from the first clustering, and one item from the second. A $k \times k$ matrix records how many items are placed in each possible pair of classifications. Let $\mathrm{confusion}(i, j)$ be the function that reports how many items are classified as being in cluster i in the first clustering and in cluster j in the other clustering. If the clusterings agreed completely, then only the main diagonal of this confusion matrix would be populated: all other entries would be zero. We therefore define the following measure for assessing a pair of clusterings:

Definition 5.4.4 *The* confusion matrix agreement *between two k-clusterings requires the use of a confusion matrix on the clusterings that records the number of items in each clustering that are allocated to the same cluster. The agreement is given by*

$$\frac{\sum_{i=1}^{k} \mathrm{confusion}(i, i)}{\sum_{i=1}^{k}\sum_{j=1}^{k} \mathrm{confusion}(i, j)}$$

However, it is quite possible that two clusterings with very different allocation of items to clusters could still be a good quality clustering. To remedy this, for any clustering, we can compute the total distance of each element in the clustering from the center of its cluster. Any clustering algorithm should attempt to minimise this amount. We can then evaluate this distance for the clustering obtained using sketches as a percentage of the clustering obtained by exact distance computations. Let $spread_{exact}(i)$ be the spread of the ith cluster following a clustering with exact comparisons, and $spread_{sketch}(i)$ be similarly defined when approximate comparisons are used.

Definition 5.4.5 *The* quality of sketched clustering *with k clusters is defined as*

$$\frac{\sum_{i=1}^{k} spread_{exact}(i)}{\sum_{i=1}^{k} spread_{sketch}(i)}$$

5.4.2 Assessing Quality and Efficiency of Sketching

To assess the performance of sketch construction, we conducted the following experiment. For a tabular call volume data set corresponding to a single day (approximately 34Mb), we measured the time to create sketches of various sizes, for both L_1 and L_2 norms. We considered objects (tiles) from size only 256 bytes up to 256 kilobytes. The tests evaluated the time to assess the distance between 20,000 random pairs of tiles in the data space. Figure 5.8 presents the results.

The exact method requires the whole tile to be examined, so the cost of this grows linearly with the size of the tile. For this experiment the sketch size, k, is 64. That is, we pick the dimensionality of the sketch vector to be 64, and so the sketch is formed by convolving each tile with 64 randomly created matrices. The sketch size is independent of the tile size. If increased accuracy is desired, then

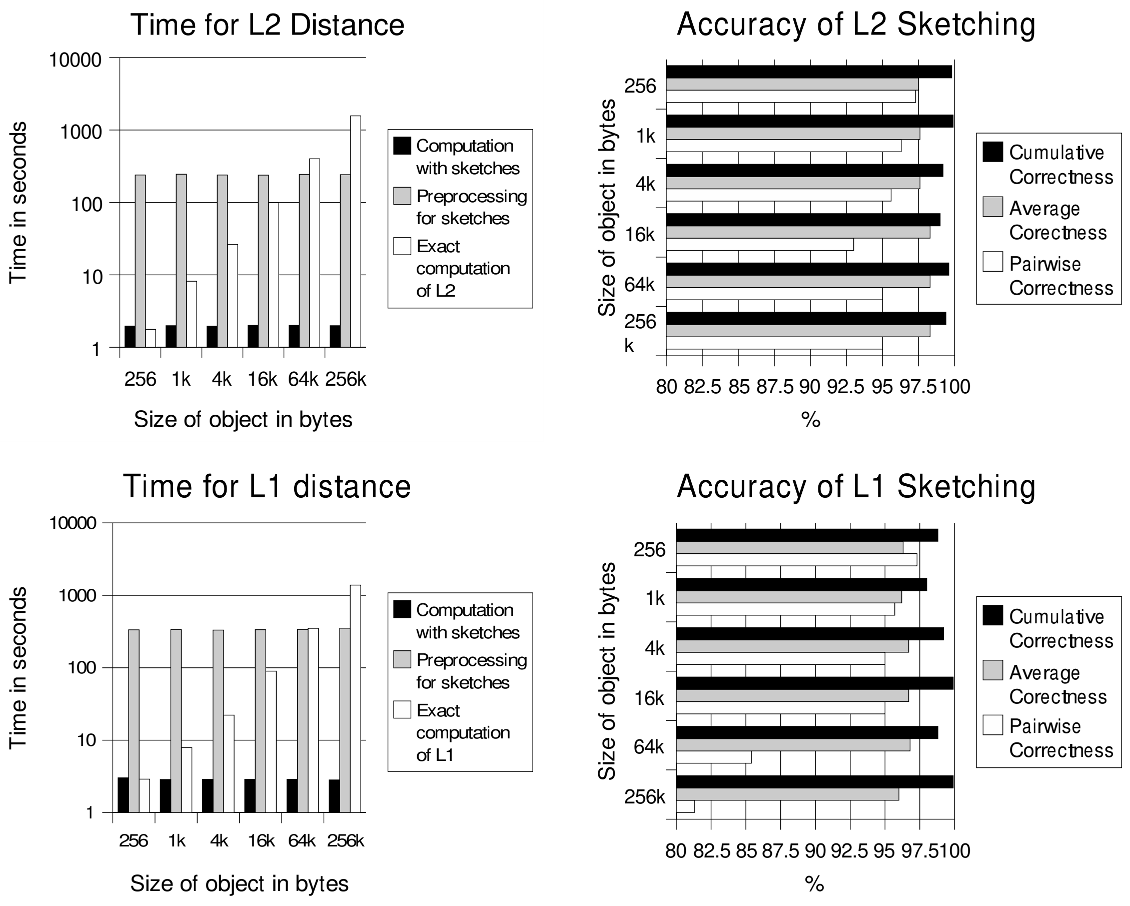
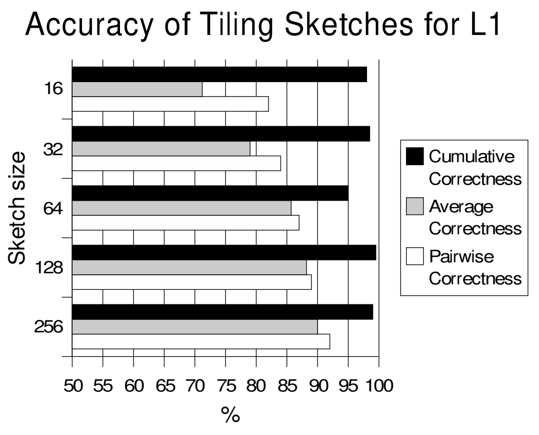


Figure 5.8: Assessing the distance between $20,000$ randomly chosen pairs

this size could easily be increased to reduce the error rate, as described in Theorem 2.2.3. Notice that the cost of assessing distance using sketches is also independent of the size of the tile: this is because we make a sketch vector based only on the parameters δ and ϵ. Hence, when we fix these parameters, the comparison time is more or less constant. When we create the sketches in advance, we consider all possible subtables of the data in square tiles (of size 8×8, 16×16 and so on up to 256×256). By using the Fast Fourier Transform method described in Theorem 5.2.1, the processing cost is largely independent of the tile size, depending mainly on the data size. Since we use the same sized data set each time, the pre-processing time varies little. The time to assess the L_p distance using sketches is much faster than the exact method when sketches are pre-computed in almost every case. It is clear that for a particular object size there will be a number of comparisons after which sketching is always beneficial for assessing the L_p distance. Our experiments reported in Figure 5.8 show that for more than 20,000 comparisons between objects of size greater than 64k then sketching is always beneficial.

To evaluate sketching accuracy, we computed the measures introduced. We tested the accuracy by comparing the distance computed using sketches with that for the exact value. We computed Average Correctness and Cumulative Correctness (Definitions 5.4.2 and 5.4.1); in most cases these was within a few percent of the actual value, for relatively small sized sketches (recall that the accuracy of sketching can be improved by using larger sized sketches). Figure 5.8 presents the results. Note that the times are shown on a logarithmic scale, since the cost for exact evaluation becomes much higher



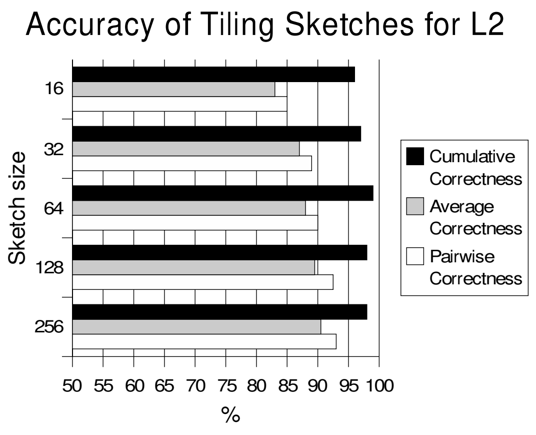


Figure 5.9: Accuracy results when tiling four sketches to cover a subrectangle

for larger tile sizes. On the average, the cumulative distance found by sketching agrees with very high accuracy with the actual answer.

Since in clustering it is not the absolute value that matters, but the results of comparisons, we ran tests by picking a pair of random points in data space and a third point. We determined which of the pair was closest to the test point using both sketching and exact methods, and recorded how frequently the result of the comparison was erroneous. This is the Pairwise Comparison Correctness of Definition 5.4.3. Under all measures of accuracy, the results of Figure 5.8 show that sketching preserves distance computations very effectively.

Observe that the quality of the pairwise comparisons decreases slightly under L_1 distance for large tile sizes. We claim that this is explained by our data set: for large enough tiles, the L_1 distance between any pair is quite similar. Hence, with the variation introduced by our approximation, it becomes harder to get these comparisons correct. However, in this situation, we claim that such errors will not affect the quality of any derived clustering detrimentally, because following such a comparison, it does not make much difference which cluster the tile is allocated to, since the tile is approximately the same distance from both. We shall see that this claim is vindicated when we examine the quality of the clusterings derived using approximate distance comparisons.

Tiling Sketches

In Definition 5.2.1, we saw how four sketches can be tiled to cover a rectangular area to make a sketch that is up to a factor of four worse than a normal sketch. In this section we go on to see how well this technique works in practice, and whether pairwise comparisons work well with tiled sketches.

It turns out that although tiling sketches is not as accurate as having sketches of the right size, this method of constructing sketches is much more accurate than our worst case analysis in Theorem 5.2.2 would lead us to expect. Figure 5.9 shows the result of experiments with the L_1 and L_2 metrics. As before, we took 10,000 subrectangles from our data at random, and found the exact and sketched distance between them. This time, the size of the subrectangles was also a random parameter. The sketched results were scaled in accordance with the ratio of the size of the rectangle being covered to the size of the rectangles covered by sketches. The cumulative correctness is in the high ninety percentiles, as before, but now the average correctness for both metrics only approaches 90% as the sketch size increases to 256 entries (recall that increasing the sketch size should improve the accuracy). The pairwise accuracy of using tiled sketches also improves beyond 90% as the sketch size is increased. These results are interesting, since they suggest that in practice tiling sketches can be almost as effective

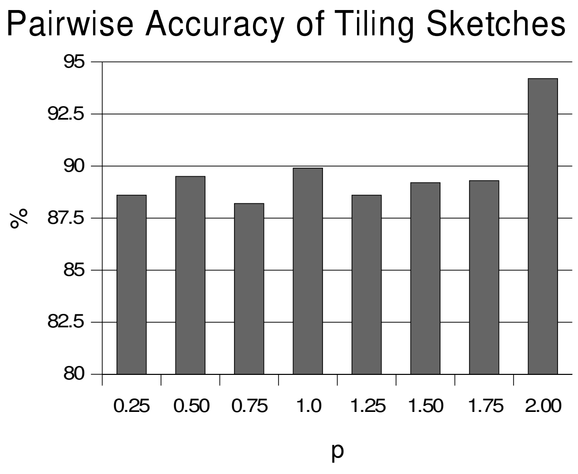


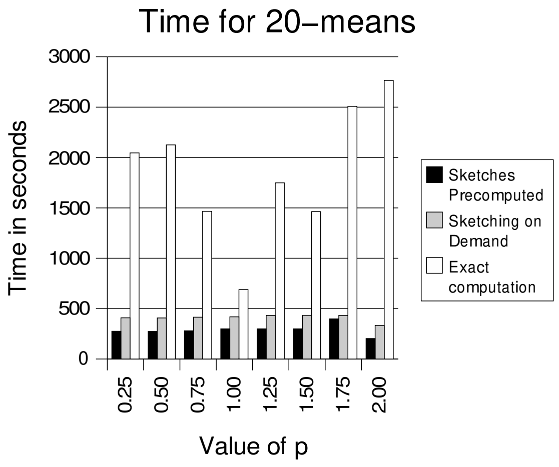
Figure 5.10: Quality of pairwise comparisons with tiled sketches

as using sketches of the exact size. This could in part be explained by the data: real life data might tend to have similar properties throughout the sub-rectangle being sketched. So despite the fact that some parts of the L_p distance between rectangles are "over-counted" by tiling sketches (some parts of the difference are counted twice, some four times, and the rest only once, as shown in Figure 5.2), we can still get an answer that is "good enough" in most cases following the scaling we do to adjust for the areas of overlap.

When we look at the accuracy of pairwise comparisons for other L_p norms, we see that the quality is not so good for norms other than L_1 and L_2, but still in the high eighty percentiles (Figure 5.10). Here, the parts of the rectangle that are over-counted by the overlapping rectangles of the tiled sketches have more of an impact, since where the distances between one rectangle and the two it is compared against are quite similar, this over-counting can easily bias the comparison in the wrong direction. Still, this approximation of the distance is still sufficiently accurate for many applications, such as clustering, where as we will go on to show, a small fraction of erroneous comparisons do not affect the overall quality of the result.

5.4.3 Clustering Using Sketches

The second set of experiments aims to evaluate the utility of sketching when applied to clustering with k-means, a popular data mining algorithm. In this experiment, we stitch 18 days of data together, constructing a data set of over 600Mb of raw data. With our experiments we wish to evaluate the performance of sketching under the following possible scenarios: (1) Sketches have been pre-computed, so no time is devoted to sketch computation, just to running the clustering algorithm on sketches (2) Sketches are not available and so they have to be computed "on demand" (3) Sketching is not used; instead the exact distance is computed. In each case, we divided the data up into tiles of a meaningful size, such as a day, or a few hours, and ran the k-means clustering algorithm on these tiles. To ensure that the methods were comparable, the only difference between the three types of experiments was the routines to calculate the distance between tiles: everything else was held constant.



Results for clustering the data using k-means (k=20), with data divided into tiles of size 9Kb.

Figure 5.11: Timing results for different L_p distances

Varying the value of p

Figures 5.11 and 5.12 present the results for a tile of size 9Kb. This tile represents a day's data for groups of 16 neighbouring stations. We experimented with a variety of settings for p: the traditional 2.0 and 1.0, and fractional values in between. The first set of results show that sketching is dramatically faster than using exact distance computations, while giving a clustering that is as good as that found using exact computation. To assess clustering quality we used two approaches. First, by creating a confusion matrix between the clustering using sketches and the clustering with exact computations. The percentage of tiles that are classified as being in the same cluster by both methods indicates how close sketching gets to the benchmark method, that is, the confusion matrix agreement of Definition 5.4.4. An alternative measure of the quality of two clusterings comes by comparing the spread of each cluster; the better the clustering, the smaller this spread will be. The spread is the sum of the divergence of each cluster from the centroid of that cluster. This gives the objective way to test the quality of the clusterings described in Definition 5.4.5.

When sketches are pre-computed the time to perform the clustering is many times faster, in some cases, an order of magnitude faster, than using exact distance computations. This is because the tiles being compared in this test are 9Kb in size, whereas the sketches are less than 1Kb, and so can be processed faster. By Theorem 2.2.3 the size of sketches is independent of the data size, so we know that this difference will grow more pronounced as the size of the objects being compared increases.

Perhaps less expected is the result that even when sketches are not available in advance, computing a sketch when it is first needed, and storing this for future comparisons is worthwhile. In fact we obtain major time savings: the speed-up is of the order of 3 to 5 times. Although creating a sketch can be costly (it requires the data tile to be convolved repeatedly with randomised matrices), in the application of clustering, one data tile will be compared with others many times. So the cost of making the sketch is recouped over the course of the clustering since each subsequent comparison has a much lower cost than the corresponding exact computation. Observing Figure 5.11, it is evident that there exists little variation in the cost of the algorithms using sketches, whereas the timings for the exact computations are much more variable, but consistently significantly more costly than with approximate comparisons. The variation in time for the exact computation is for two reasons: firstly, L_1 distance is much faster to compute, since it requires only computing sums of absolute differences, whereas

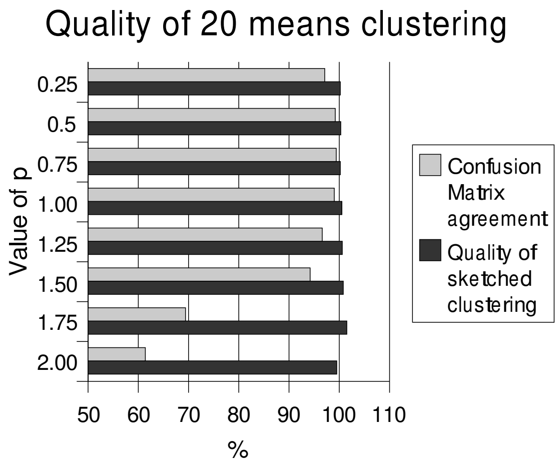


Figure 5.12: Quality results for different L_p distances

the other methods need to compute sums of powers of differences. Secondly, the k-means clustering procedure stops when a stable clustering is found, and for each value of p a different clustering will be reached. So some computations are finishing earlier than others, whereas it appears that computations with sketches display less variability. In each case, creating sketches adds the same cost (about 130s of compute time) since the number of sketches that are required, and the cost of creating each sketch is independent of the data values and the value of p. Note that since a slightly different method is used for $p = 2$ compared to $p < 2$ (see Section 2.2) — this means that L_2 distance is faster to estimate with sketches in this case, since the approximate distance is found by computing the L_2 distance between the sketches, rather than by running a median algorithm, which is slower.

Another positive result regards the accuracy of the sketching approach shown in Figure 5.12. By analysing the confusion matrix between computations using sketches and computations using exact distances, we see that for several of our experiments there is a high degree of correlation, indicating that tiles were allocated into the same cluster. We observe that the agreement is less good for higher values of p — for L_2 distance, it reduces to around 60%. But although the clustering we get is quite different from that obtained with the exact distance, the quality of the clustering in all cases is as good as the one found by exact methods. In fact, in many cases using sketches produces a clustering that is better than that with exact comparisons (where the quality rating is greater than 100%). This at first surprising result is partially explained as follows: even when we compute exact distances, the k-means algorithm does not guarantee finding the best clustering. Evaluating distances using sketches introduces small distortions in the distance computation that cause a different clustering to be reached (some objects will be placed in a different cluster). This clustering can be a small improvement over the one obtained by exact computations. There is no obvious reason for this, except to note that there is no guarantee that k-means will give the best result, and it appears that comparisons with a small probability of erring do not adversely affect it.

Varying the number of clusters

Figure 5.13 shows a series of experiments on the same set of data with k-means, as k is increased. The difference between having pre-computed sketches, and sketching on demand, remains mostly constant,

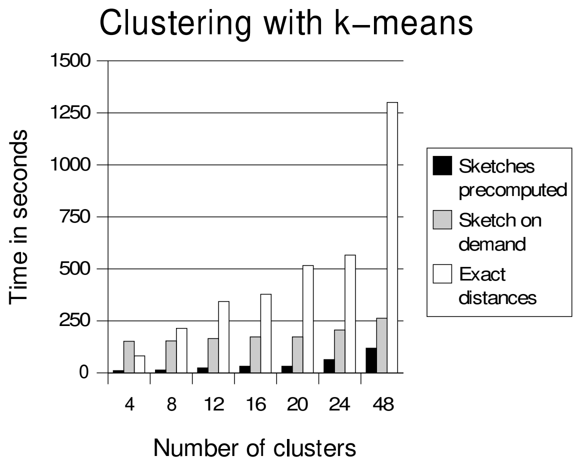


Figure 5.13: Clustering with different numbers of means

at around 140s. The cost of using exact computations is significantly more expensive in all but one case[3], and without sketches the time cost appears to rise linearly with k. This is achieved using relatively large sketches with 256 entries. This time benefit could be made even more pronounced by reducing the size of the sketches at the expense of a loss in accuracy.

We performed additional experiments, varying the size of the tile and other parameters. The results obtained were consistent with those reported; we omit these for brevity. In summary, the use of approximate comparisons does not significantly affect the quality of the clustering found. Indeed, we have shown cases where the quality of the clustering is improved. The k-means algorithm is already inexact: it depends on a heuristic to generate the clustering, and uses randomness to generate the initial k-means that are refined over the course of the program. We have shown that adding approximate distance comparisons to this clustering algorithm makes it run significantly faster, without noticeably affecting the quality of the output. This provides evidence that clustering algorithms or other procedures making use of L_p distance computations can be significantly speeded up by using approximate computations with sketches, and that this will yield results that are just as good as those made with exact computations.

5.4.4 Clustering Using Various L_p Norms

With our last experiments we wish to evaluate the utility of using various L_p norms in clustering for data mining tasks. We examine the output of the clustering procedure to determine the effects of varying the parameter p in the distance computations. We approach this in two ways: first, by examining how varying p can find a known clustering in synthetic data; and second, by presenting a case study analysing a single day's worth of data.

Synthetic Data

Recall that our synthetic data set is made by dividing the data into six pieces, and filling each with a distinct pattern, then adding "errors" of high and low readings. We divided this data set into square tiles of size 64k, and ran clustering using sketching on them for many values of p in the range $[0, 2]$. Since we know which cluster each tile should belong to, we can accurately measure how well the

[3]This occurs when the number of comparisons made in the course of clustering is not enough to "buy back" the cost of making the sketch.

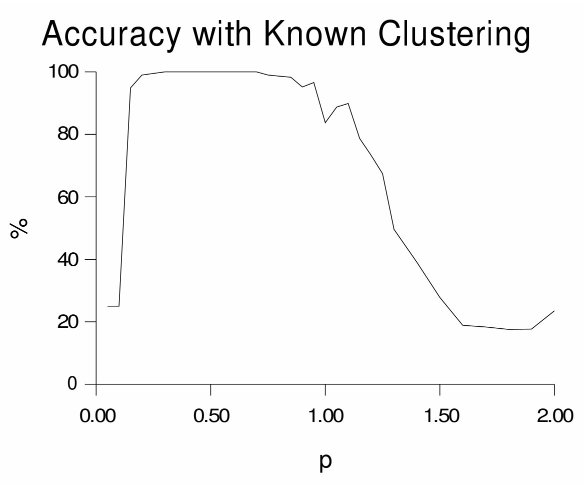


Figure 5.14: Varying p to find a known clustering

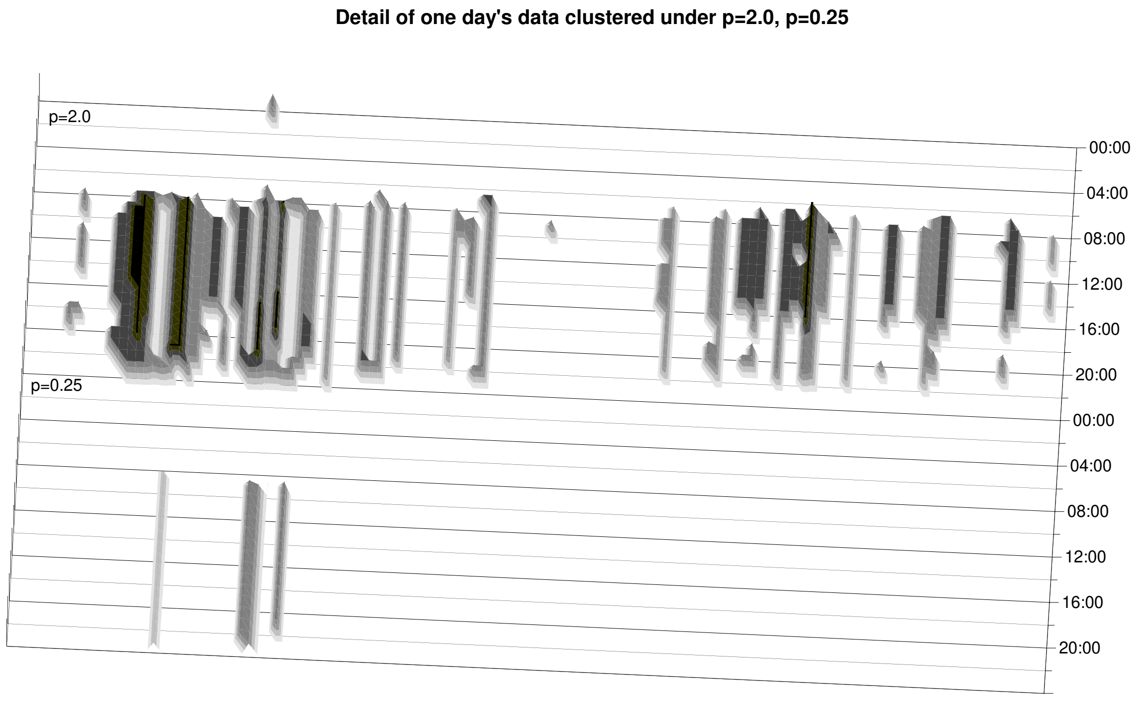
clustering does in the presence of the artificial errors and the approximation caused by using sketches. We measure the percentage of the 2000 tiles allocated to their correct cluster (as per Definition 5.4.4). Figure 5.14 shows the results as we vary p.

We first observe that traditional measures, L_1 and L_2 especially perform very badly on this dataset. But if we set p between 0.25 and 0.8, then we get the answer with 100% accuracy. The explanation for this is that the larger p is, the more emphasis it puts on "outlier" values: so if a single value is unusually high, then this will contribute a huge amount to the L_2 distance — it adds the square of the difference. With distances this high, it is impossible to decide which of the clusters a tile best belongs to, and the clustering obtained is very poor (if we allocated every tile to the same cluster, then this data set under this measure would score 25%). On the other hand, as p gets smaller, then the L_p distance gives a lower penalty to outliers, as we would wish in this case. If we keep decreasing p closer to zero, then we have seen that the measure approaches the Hamming distance, that is, counting how many values are different. Since here almost all values are different, the quality of the clustering is also poor when p is too small. However, we suggest that $p = 0.5$ gives a good compromise here: the results are not overly affected by the presence of outliers, and even with the approximation of sketching, we manage to find the intended clustering with 100% accuracy.

Real Data Case Study

For the case study, the geographic data we used was linearised, and grouped into sets of 75 neighbouring stations to facilitate visual presentation. A subsection of the results of the clustering is shown in Figure 5.15 (the full results are displayed in Figure 5.16). Each point represents a tile of the data, an hour in height. Each of the clusters is represented by a different shade of grey. The results shown in Figure 5.16 are for three different distance measures, L_2, L_1 and $L_{0.25}$. The largest cluster is represented with a blank space, since this effectively represents a low volume of calls, and it is only the higher call volumes that show interesting patterns.

Visual analysis of this clustering immediately yields information about the calling patterns. Firstly, it is generally true that access patterns in any area are almost identical from 9am till 9pm. We can see very pronounced similarities throughout this time — long, vertical lines of the same hue indicating that an area retains the same attributes throughout the day. Call volume is negligible before 9am, but drops off gradually towards midnight. It is also clear how different values of p bring out different



The top figure shows the clustering graphically when $p = 2.0$; the lower when $p = 0.25$. Each shade of grey denotes a different cluster (the largest cluster is denoted by a blank space to aid visibility). For higher p, more detail can be seen; for lower p, the most important sections are brought to the fore.

Figure 5.15: Detail of a clustering for a single day (subsection of Figure 5.16)

features of the data in this data set. For $p = 2$, perhaps the most common distance metric, a large fraction is allocated to non-trivial clusters. These clearly correspond to centers of population (such as New York, Los Angeles and so on) represented by clusters of darker shades. On either side of the centre of these areas (the greater metropolitan areas), patterns are less strong, and access is only emphasised during peak hours. We see how the clusters represented by darker shades are often flanked by those of lighter hue, corresponding to this phenomenon. For $p = 1$, there is less detail, but equally, the results are less cluttered (in Figure 5.16). Most of the clusters are indistinguishable from the background, leaving only a few places which can be seen to differ under the L_1 measure. It is significant that these also show strong vertical similarity of these clusters, suggesting that these areas show particular points of interest. When studying the full data set, we noticed that while some areas figure throughout the day, some sections on one side of the data are only active from 9am till 6pm; at the other extreme of the data diagram there are regions that are active from 6am to 3pm. This is explained by business hours and the three hour time difference between East and West coasts of the USA (this is visible on Figure 5.16). This difference is not so clearly visible on the more populated clustering with $p = 2$. As we further decrease p to 0.25, only a few regions are distinct from the default cluster. These are clearly areas worthy of closer inspection to determine what special features they exhibit. It therefore seems that p can be used as a useful parameter of the clustering algorithm: set p higher to show full details of the data set, or reduce p to bring out unusually strong clusters in the data.

This example demonstrates how clusterings of the tabular data set can highlight important

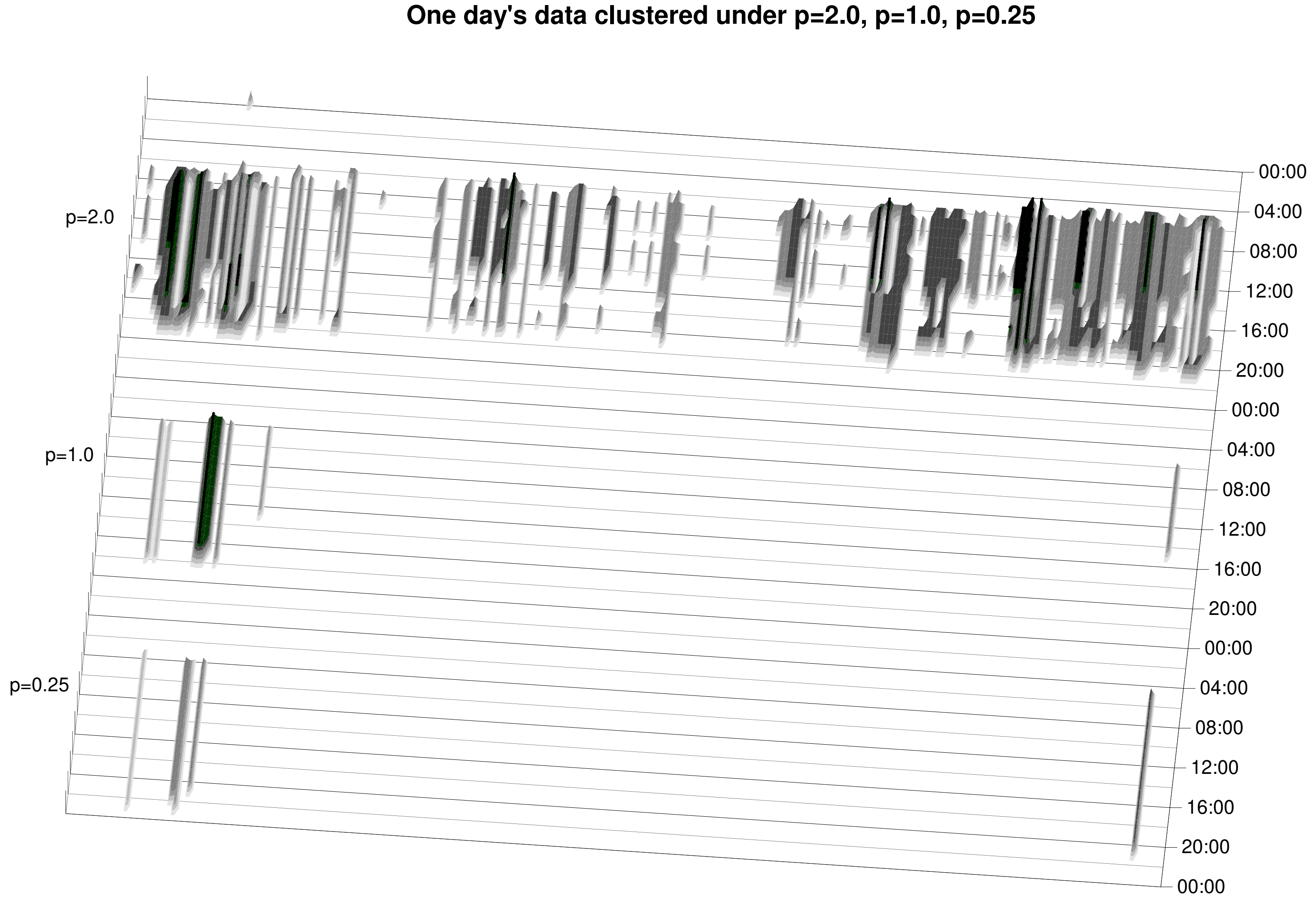


Figure 5.16: Clustering of one day's call data for the whole United States.

features. It has been confirmed by our knowledge of this data set; in novel data sets, we would expect to be able to find the important features of the data, without the foreknowledge of what these might be. It also demonstrates the importance of different values of p, in contrast with previous attention focusing on the traditional distance measures L_1 and L_2. The whole continuum of L_p distances can give different and useful information about the data.

5.5 Discussion

This chapter has focussed on studying the application of sketch based dimensionality reduction techniques to real world problems. The main results shown here are:

- Application of sketching to solve real world problems of comparing streams based on Hamming distance, and clustering in large tabular data based on L_p distances.
- Empirical proof that this style of sketching is practical, fast, space-efficient and useful in competition with other methods to tackle these problems.
- A study of using L_p norms for small p to successfully compare massive data streams and accurately compute the number of distinct elements.
- Experiments on using sketches in place of exact comparisons in clustering applications, and demonstration that this gives significant speed improvements while sacrificing very little quality of the final result

Comparisons based on L_p distances — either L_1, L_2 or Hamming distance — occur as a fundamental operation throughout Computer Science, especially in applied areas such as Database Theory. These sketching techniques could be applied to almost any procedure which uses this kind of comparison at its core to speed it up, or to reduce the working space that it uses. But further to this, the study of L_p norms for fractional values of p has also shown significant results. We have seen that values of p less than 1 sometimes give results more akin to those we want than traditional metrics such as L_1 and L_2. We have also seen that by setting p close enough to 0 we obtain a good approximation of the Hamming distance on vectors. Thus the parameter p is an important variable, and letting p take values in the full range $[0 \ldots 2]$ can give different and interesting results depending on our application scenario.

Chapter 6

Sending and Swapping

...let's assume that an author drafts the first version of his text, and let's call this version A. To simplify things, let's assume that the author writes it directly on a computer, or that if he had made any handwritten notes, they have disappeared.

This version A is printed, and at that point, the author begins correcting it by hand. In this way, we get version B, which in turn is transferred to computer, where again it is cleaned up and printed anew and becomes version C. In turn, this version is altered by hand and again recopied as version D on the computer, from which a new version will be created: version E. Since computers encourage corrections and reconstructions, this is how the process can give rise to — if the author does not throw the intermediate steps in the wastepaper basket — a series of versions, let's say from A to Z...

And here — when we print out the version again — we do not have that version C, which was supposed to faithfully reproduce version B. Instead, out comes a version that we will call X, but between B and X there are "ghost" versions, each one different from the other.

It could be the case, though rare, that the author — narcissistic and fanatical about his own changes, and using some kind of special computer program — has kept somewhere, inside the memory of the machine, all these intermediate changes. But usually this does not happen. Those "ghost" copies have vanished; they are erased as soon as the work is finished.

— Umberto Eco, 2002

6.1 Introduction

This chapter focuses on the communication issues related to managing documents shared by physically separated users. Suppose that two users each have a different version of what was at some unknown point of time in the past the same document. Each version was perhaps obtained by a series of updates to that original document (perhaps by a series of transmissions on the Internet and updates along the way). In this situation it would be reasonable to expect that many parts are similar if not identical. The challenge addressed here is how to let one user know the other version and save transmitting redundant information as much as possible. What makes the problem more challenging is that we do not assume that the updates themselves, or any logs of them, are available; the original document is also not assumed to be available — each user has only their current version.

Formally, let A and B be two such users holding documents a and b respectively. Neither A nor B has any information about the document held by the other. Their goal is to exchange messages so that B computes a; it may also be required for A to compute b as well. Our objective is to design efficient communication protocols to achieve this goal. We measure the efficiency of a protocol in terms of the total number of bits communicated, as per Yao's model of communication complexity (described in [KN97]). We are also interested in minimising the number of rounds (changes of direction in communication) as well as minimising the running time of the internal computations performed by A and B.

A trivial but inefficient way of exchanging documents is for A to send a to B in full. On the other hand, if A knew b (in addition to a), A could communicate a to B more efficiently by sending a sequence of operations for converting b to a. In this chapter we consider protocols in which the set of operations to convert one string into another is pre-determined between the users. Hence we shall be dealing with the same distances that we have already discussed, and using some of the embedding results already seen to build these protocols. The distance between a and b will allow us to find a lower bound on the length of the message that A must send to B.

In most common cases, a and b will be strings, and we shall be interested in analysing protocols that are efficient in terms of string distance. We will also discuss the case when a and b are other sequences — permutations or vectors. In this chapter, we will first study the body of work on this kind of problem in Section 6.1.1. We shall then find lower bounds on the amount of communication needed under different distance measures, described in Section 6.2. A general protocol for a large class of metrics is presented in Section 6.3 which gives a single round scheme that is within a factor of 2 of the optimal amount of communication. For individual distances, a series of divide-and-conquer schemes are presented in Section 6.4 which require a greater amount of communication, but are more computationally efficient to perform.

6.1.1 Prior Work

There have been several earlier works which take the following scenario: Two users each have a copy of a file. The file is divided into a sequence of pages, and certain pages differ between the two copies. The distance between two files is then the number of differing pages — it is assumed that the differences between pages are aligned with the page boundaries, so effectively this results in the Hamming distance between two strings. Each page corresponds to a character in the string, and two corresponding pages are either identical or disagree.

These protocols assume a fixed upper bound f on h(a,b) and they typically require the transmission of $\Omega(h \log n)$ bits to exchange files of size n with Hamming distance h. Metzner [Met83] first studied this problem, and described a communication scheme based on hash functions. It works in an obvious divide-and-conquer fashion: a hash for the whole file is exchanged; if there is a discrepancy then the file is divided into two parts, and the procedure continues on each half. Metzner asserts that by choosing a sufficiently large basis, then the probability of failure of this protocol becomes vanishingly small; in fact, as we shall see in our detailed analysis of this protocol (Section 6.4.1), the hash size must be a function of the file size and the distance between the files for this protocol to succeed with constant probability. We shall later show that the communication cost of this protocol is $O(h \log n \log f)$, and uses $O(\log n)$ rounds of communication.

In a later paper, Metzner presents an alternative procedure for the Hamming distance [Met91]. This uses the techniques of Reed-Muller codes to identify the location of any single differing page between the files. For files which differ in more than one place, the same divide and conquer technique is suggested, but using Reed-Muller codes in place of hash functions. In this way, if a section of the file under consideration has only one difference within it, then only a single exchange is required, rather than repeated division until the differing page is identified. In the worst case (when differences are clustered), this protocol still requires $O(h \log n \log f)$ bits of communication and $O(\log n)$ rounds.

Abdel-Ghaffar and El Abbadi [AGE94] and Barbará and Lipton [BL91] extend the approach of using techniques from error-correcting codes, with the aim of reducing the total amount of communication, and the number of rounds of communication. In particular [AGE94] describes a protocol based on Reed-Solomon codes; this protocol sends a single message of $O(f \log n)$ bits to correct up to f differences, which is within a constant factor of optimal if the bound f is tight. The same coding theory approach is taken in [MTZ01], which attempts to be computationally efficient in terms of the size of the sets (the number of non-zero entries in a vector), rather than in the size of the universe. This accounts for distances akin to the Hamming distance; there are somewhat fewer studies of the harder problems of general editing distances.

A simple scheme is described by Tridgell and Mackerras [TM96] in the design of a file exchange utility, *rsync*. The scheme has a single round of communication, and is initiated by B who wishes to receive a from A. B divides b into fixed length blocks, and computes fixed length hashes on each of these blocks; all these hashes are then sent to A. Starting from the first character in a, A computes a hash of the block starting at that location. If this hash matches a hash sent by B, then A adds the identifier of this hash to the output and advances by the block length; else A outputs the character under consideration, and advances by one character. In this way, A builds up a description of a in terms of blocks of b and single characters that B will be able to interpret. From our point of view, this approach is undesirable, since the total amount of communication is linear in the length of the files being exchanged, irrespective of whatever the true distance between the files is — even if a and b are identical. From a practical viewpoint though, this protocol is acceptable since it generally uses less communication to exchange related files than simply sending them directly, requires only two rounds of communication, and is computationally efficient. In fact, this utility is available to download and run on Unix systems [Tri].

Schwarz, Bowdidge and Burkhard [SBB90] take a model of file differences where not only can pages be changed, but they can also be moved, inserted or deleted. The proposed protocol is similar in nature to that originally given in [Met83]. One party sends hashes to the other, and if they are unable to reconcile them, then the substring under investigation is split into two and the process continues. Hence there are again $O(\log n)$ rounds of communication. The difference is that because insertions and deletions can alter the alignment of pages, upon receiving a hash the second party must try aligning it against all possible locations in the file. They claim that their protocol sends $d \log n$ hash values to exchange files with an edit distance of d. We shall later show that this can be improved to $O(d \log(n/d))$ and formally prove this claim. We shall also analyse how large the hash values must be to ensure that the probability of failure is bounded by a constant and hence find the total amount of communication necessary.

Evfimievski [Evf00] also considers block operations but again uses essentially the same divide and conquer technique to exchange files. However, whereas the earlier schemes performed the divisions in parallel — first considering all differing substrings of length $n/2$, then all of those of length $n/4$ and so on — Evfimievski's technique requires that the process be carried out by an inorder traversal of the binary parse tree. The bulk of the paper is to show that (the distance we call) compression distance with unconstrained deletes, du, is at most an LZ distance of $112du^2$. As we will prove in Section 6.4.4, the method described can exchange files using $O(l \log n)$ hashes for an LZ distance of l. Hence, applying this result, the files can be exchanged using $O(du^2 \log n)$ (hence this will not fit our definition of being efficient with respect to the distance measure, given in Definition 6.1.1). We shall later show how, using results based on the Edit Sensitive Parsing, this can be improved to $O(du \log^* n \log n)$. This is superior for all values of du.

Between them, the abovementioned papers consider Hamming distance or other particular editing distances. Orlitsky [Orl91] takes a more general approach to show how, given a suitable distance metric, documents can be exchanged using at most twice the optimal amount of communication in a single round. The proof of this is based on colouring hypergraphs. Later, we shall give an alternative proof of this fact, which is valid for all of the metric editing distances that we consider herein. Our proof is based on colouring graphs, and leads to a deterministic algorithmic solution. We shall see that the computational cost of a literal translation of this proof is exponential, rendering this approach of theoretical interest for the most part. However, by considering specifics of each distance it is sometimes possible to adopt this approach and make it practical.

6.1.2 Results

Many of the methods outlined above require an upper bound on the distance between two strings, but give no indication how such an upper bound could be obtained. By using the embeddings of string distances into vector spaces that we are able to make sketches for (Hamming and L_1), we can overcome this limitation by using approximate upper bounds on the distance. One party can build a sketch for their sequence and communicate it to the other in a single round of communication. In this chapter we shall describe communication efficient methods for exchanging documents using this upper bound. We begin by describing a single-round method that allows the files to be exchanged with an amount of communication that is at most twice the optimal amount. Here A deterministically computes and sends B an identifier for the string a which is unique among all strings that are within a distance of $2d(a,b)$ from b, where d is the distance of interest. By the use of known techniques in graph colouring or error-correcting codes, the size of this identifier can be bounded above by $c \log |b| \cdot d(a,b)$ bits, where c is a small constant.

We also provide simpler protocols that locate the differences between a and b in divide-and-conquer fashion (see Section 6.4). These protocols make extensive use of fingerprints for searching and comparing substrings of a and b. Some of these have already been described for Hamming distance, Edit distance and Compression distance with unconstrained deletes. We shall put these different methods into a common context and formally analyse their cost. We shall also give methods for distances, including Block Edit Distances, Tichy's Distance, and the Permutation distances studied in Chapter 3, where we know of no prior work on these problems. These methods are distinguished by the fact that they are communication efficient, and computationally tractable. In particular, we shall be interested in protocols that are efficient with respect to the relevant distance measure:

Definition 6.1.1 *A protocol which allows two parties to exchange objects a and b is* efficient with respect to a distance measure, *d, if the the communication cost of the protocol is guaranteed to be $O(d(a,b)\,\text{poly-log}(|a| + |b|))$.*

The important part is that for an efficient scheme, the dependency on the number of differences between the two objects must be linear. All the protocols that we describe will be efficient with respect to their corresponding distance measures. We also consider the following definition:

Definition 6.1.2 *A protocol between two communicating parties A and B is* non-trivial *if it enables A to transmit a string a to B, under a stated restriction on the distance between a and b, using fewer than $|a| \log |\sigma|$ bits of communication in total.*

Under this definition, the obvious scheme of sending a string a directly to B is considered trivial. We shall be analysing our protocols to determine under what circumstances they are provably better than this naïve scheme — that is, when they are non-trivial.

6.2 Bounds on communication

To examine the performance of the proposed protocols, we need a lower bound on the amount of communication required. We can obtain an information theoretic lower bound by considering how many bits are necessary for B to be able to compute a, given that B already holds b. If A already knew b in addition to a, A could send a single message containing sufficient information for B to derive a from b. If, for any b, there are at most m strings that a could possibly be, then $\lceil \log m \rceil$ bits are necessary to distinguish between them. The number of such possible strings is determined by the distance between a and b. We shall denote by k the quantity $|\sigma| - 1$, where σ is the alphabet from which our strings are drawn.

Lemma 6.2.1 *The amount S of communication needed to correct h Hamming differences between strings a and b ($|a| = |b| = n$) is bounded in bits as: $h \log(nk/h) \leq S \leq h(\log(nk/h) + 2)$, provided $h \leq n/2$.*

Proof. The number of strings which satisfy $d(a, b) = h$ can be found exactly: we can choose h locations out of the n and make a change there. Each of these characters can be changed to any one of $|\sigma| - 1 = k$ different values. This gives precisely $\binom{n}{h} k^h$ different strings. We show $\binom{n}{h} k^h \geq \left(\frac{kn}{h}\right)^h$ by induction in h which follows since $(1 + 1/h)^h (n - h) \geq 2(n - h) \geq n$ for all $2 \leq h \leq n/2$. The base case $h = 1$ gives equality. For the upper bound, we can similarly show that $\binom{n}{h} k^h \leq \left(\frac{ken}{h}\right)^h$ also by induction in h.[1] □

Lemma 6.2.2 *The amount S of communication needed to correct e edit operations is bounded as: $e \log(2(|\sigma| - 1)n/e) \leq S \leq e(\log(2|\sigma|(n + 1)/e) + \lceil \log \log n \rceil)$, where n denotes the length of the string a.*

Proof. The lower bound follows by considering strings generated by only changes or insertions. The number of such strings that can be formed from a with e insertions or changes can be found by choosing i locations to make an insert and $(e - i)$ locations to make a change. This is at least $\sum_{i=0}^{e} \binom{n}{i} k^i \binom{n}{e-i} k^{e-i}$. This expression simplifies to $k^e \sum_{i=0}^{e} \binom{n}{i} \binom{n}{e-i}$, which is just $k^e \binom{2n}{e}$. Using the methods of Lemma 6.2.1 bounds this quantity from below with $(\frac{2kn}{e})^e$. Taking the logarithm of this expression gives the required lower bound.

For the upper bound, we instead consider a bitwise encoding of the operations. Clearly, if we can encode all possible sequences of e edit operations in some number of bits, this upper bounds the amount of communication needed. We consider the operations in order of the occurrence (left to right). For the i'th difference, we encode the distance d_i from the last difference with $\log d_i$ bits, plus $\lceil \log \log n \rceil$ bits to encode the length of the encoding of d_i. The total cost of this is $\sum_i (\log d_i + \lceil \log \log n \rceil) = e \lceil \log \log n \rceil + e \sum_i (\log d_i)/e$. Using Jensen's inequality, this is no more than $e \lceil \log \log n \rceil + e \log \sum_i (d_i/e) \leq e(\log((n + 1)/e) + \lceil \log \log n \rceil)$

It is possible to use a bit flag to denote whether each edit is an insertion or a change, requiring a further e bits. We then use $\lceil \log |\sigma| \rceil$ bits to code the character concerned. In the case of a deletion, this is represented as a 'change', but using the code of the existing character at that location. The total cost

[1]Here, e is the base of the natural logarithm as normal.

of this encoding is given by $e(\log(|\sigma|(n+1)/e) + \lceil\log\log n\rceil) + e$. Summing these two costs gives the upper bound, as required. $\square$

For each of the other distances, we can take this idea of encoding each edit operation using a certain number of bits to get a bound on the amount of communication needed.

Lemma 6.2.3 *The amount S of communication needed to exchange strings or permutations such that the relevant distance between the two items is d is bounded in the following ways for the following distance measures ($|a| = n$)*
i) **Compression distances:** $S \leq 9d\lceil\log|a| + |b|\rceil$
ii) **Tichy's distance:** $S \leq 2d\lceil\log n\rceil$
iii) **LZ distance:** $S \leq 2d\lceil\log|a| + |b|\rceil$
iv) **Edit Distance with moves:** $S \leq 3d\lceil\log 2n\rceil$
v) **Reversal distance:** $S \leq 2d\lceil\log n\rceil$
vi) **Transposition distance:** $S \leq 3d\lceil\log n\rceil$
vii) **Swap distance:** $S \leq d\lceil\log n\rceil$
viii) **RITE distance:** $S \leq 3d\lceil\log 2|a| + |b|\rceil$
ix) **Permutation Edit distance:** $S \leq 2d\lceil\log n\rceil$

Proof. In each case we show a simple bitwise encoding of the relevant operations to give the corresponding upper bound.
i) **Compression distance:** We show a bitwise encoding of the allowed operations to give the upper bound. A copy or move operation can use $\lceil\log|a|\rceil$ bits to specify each of the start, length of substring and destination. Uncopies, or other operations can be encoded using fewer bits. Provided $|a| + |b|$ is more than $|\sigma|$ we will require no more than $3d\lceil\log^3(|a| + |b|)\rceil$ bits for d operations. In order to show that each operation can be speicified using $O(\log n)$ bits, we claim that any intermediate string consists solely of substrings of a or b. Clearly, this is true at the start and the end of the editing operation. Then note that any block operation (copying, moving or uncopying) which operates on a string containing only substrings of a and b results in a substrings consisting of substrings of a and b. Creating a substring that does not belong to either the source or destination string can be accomplished by character operations, but we must remove all such substrings at the end, and so these operations are superfluous, and can be removed from any optimal transformation. Hence, intermediate strings can be parsed into substrings of a and b. There is no need for any intermediate string to contain more than one copy of any substring of a or b. The total length of such substrings is $O(n^3)$, and so the length of any intermediate substring never exceeds this. Hence locations in each intermediate string can be specified using $O(\log n)$ bits.
ii) **Tichy's distance:** Each operation consists of copying a block from the original string. The cost of describing a copy is at most $2\lceil\log n\rceil$ bits.
iii) **LZ distance** As with Tichy's distance, each operation is a copy of a block from one or other of the strings, totalling at most $2\lceil\log n\rceil$ bits.
iv) **Edit distance with moves** A substring move operation requires $3\lceil\log n\rceil$ bits to describe the block being moved and its new location. The other operations require fewer bits to encode since $|\sigma| \leq n$. An additional 2 bits are required to flag whether the operation is an insertion, deletion, change or move.
v) **Reversal Distance:** There are $n(n-1)/2$ distinct reversals, so any one can be specified in $2\lceil\log n\rceil$ bits.
vi) **Transposition Distance** There are $\binom{n+1}{3}$ distinct transpositions, so any one can be specified in $3\lceil\log n\rceil$ bits.
vii) **Swap Distance:** There are $(n-1)$ distinct swaps, so any one can be specified in $\lceil\log n\rceil$ bits.
viii) **RITE distance:** The sequence need never exceed $\max(|a|, |b|)$ in length, given a suitable organisation of the operations. This is because if the sequence does exceed this length, then there is a char-

acter being inserted that is subsequently deleted. We will use n to denote $\max(a, b)$. Each of the operations can be specified in at most $3\lceil\log n\rceil$ bits, to describe its location (start, end and destination), plus an additional 3 bits to describe which of the five possible operations it is. This gives a cost of $3(\lceil\log n\rceil + 1) = 3\lceil\log 2n\rceil$ bits.
ix) **Permutation edit distance:** Each operation moves one element to a new position, and so can be described using $2\lceil\log n\rceil$ bits. □

6.3 Near Optimal Document Exchange

As we have already seen, many of the distances of interest are metrics. Much study has been made of metric spaces, and if we know that a distance is a metric this enables us to tell a lot about its structure. In fact, all of the metrics that we study here are defined in a particular way: we define a series of unit cost operations (such as the edit operations), and set the distance between two objects as the minimum cost sequence of operations to turn one into the other. They are therefore editing distances in the sense of Definition 1.1.1. In this situation, we can further exploit properties of the metric, and come up with a communication scheme that allows document exchange in at most twice the optimal amount of communication. We begin this section with a description of how this can be achieved with a known bound on the editing distance, and then go on to describe a multi-round adaptation that does not require an *a priori* bound on the distance.

6.3.1 Single Round Protocol

Let R be a symmetric relation on a set X. Let $d(x, y)$ be an editing distance on this set defined as usual from the transitive closure of R — that is, $d(x, y)$ is the minimum n such that $R^n(x, y)$. Suppose that there are two individuals, A who holds $x \in X$ and B who holds $y \in X$, such that $d(x, y) \leq l$ for some known l. The range of d is the non-negative integers, and we will refer to this as the distance. Let G_i be the undirected graph whose vertices are X and whose edges are $\{(x, y)|d(x, y) \leq i\}$.

Lemma 6.3.1 $\deg(G_{2l}) \leq (\deg(G_l))^2$.

Proof. $\deg(G_{2l})$ corresponds to the greatest number of vertices at a distance at most $2l$ (in the graph corresponding to R) from any particular vertex, x. As the metric is defined by unit cost operations, these vertices will be those which are at most l from some vertex at most l from x. Since from each vertex there are at most $\deg(G_l)$ vertices at distance at most l, the lemma follows immediately. □

Following the model of Orlitsky, we shall temporarily employ a hypergraph H whose vertices are the members of X and whose hyperedges are $e_y = \{x : d(x, y) \leq l\}$. Both parties can calculate this hypergraph independently and without communication (if some parameter of the graph is not implicit, A can prepend this information to the message with small additive cost). They can then use a deterministic scheme to assign colours to the nodes of the hypergraph with some number of colours. We use $\chi(H)$ to denote the chromatic number of H, the smallest number of colours required to colour H in order to ensure that for every hyperedge, all nodes in that edge have a unique colour. A's message is the colour of x. By the colouring scheme and the specification of the problem, B knows that x must be amongst the vertices in e_y, and can use the colour sent by A to identify exactly which it is. We can place upper and lower bounds on the single message complexity by considering this scheme. Since we must be able to distinguish x among the vertices in the hyperedge x, at least $\log(\deg(H))$ bits must be sent overall. To send the colour of x, we need $\lceil\log\chi(H)\rceil$ bits [KN97]. Let S be the number of bits necessary in a single message to allow B to calculate x (the single message cost). So $\log(\deg(H)) \leq S \leq \lceil\log(\chi(H))\rceil$.

Lemma 6.3.2 $\chi(H) \leq \deg(G_{2l})$.

Proof. $\chi(H)$ is the number of colours required to colour the vertices of H such that any two vertices which are both at most distance l from any point x are coloured differently. Consider y and y' such that $d(x, y) \leq l, d(x, y') \leq l$. Since d is a metric, $d(y, y') \leq d(y, x) + d(x, y') = d(x, y) + d(x, y') \leq 2l$. Any colouring which ensures that any two points within distance $2l$ have different colours is therefore a colouring of the hypergraph H. A colouring of G_{2l} achieves this, showing that $\chi(H) \leq \chi(G_{2l})$. On the assumption that G_{2l} is not complete, is connected and has degree of three or more (which will be satisfied by the examples in which we are interested), then $\chi(G_{2l}) \leq \deg(G_{2l})$, [Gib85] and so the lemma follows. □

Theorem 6.3.1 $\log(\deg(G_l)) \leq S \leq \lceil 2 \log(\deg(G_l)) \rceil$.

Proof. It is clear that $\deg(G_l) = \deg(H)$; in both cases the degree of each vertex corresponds to the number of vertices a distance at most l away. We combine this with Lemma 6.3.2 and Lemma 6.3.1 to get the fact that $\log(\deg(H)) \leq S \leq \lceil \log(\chi(H)) \rceil$ to prove the theorem. □

Thus, under certain conditions, a single message can be at worst twice the lower bound on its length for any number of messages. A similar theoretical result to that shown here is given by Orlitsky [Orl91]. By considering a more restricted class of functions, we can construct a concrete protocol which achieves the above bound. Note that our upper bound is related to the degree of a graph which we can calculate, and algorithms exist which will colour a graph with this number of colours; hence it is achievable in practice. Tractable instances of this technique are discussed further in Section 6.3.3.

Multi-round versions of the Colouring Protocol

The above section assumes that we know a tight upper bound, l, on the distance between the objects in question. This is an assumption shared by all previous works on this subject. In earlier sections, we have shown how to arrive at an approximate upper bound on distance for most distances, but a few are not covered. Here, we briefly discuss how to adapt the above technique to work in the absence of an upper bound on the distance, by introducing a probabilistic test. The modified protocol works in $O(\log d)$ rounds, instead of a single message, where d is the true distance. In the first round the users run the protocol with the assumption that $d(a, b)$ is upper bounded by some small constant. B then uses a hash function on the value of a generated by this protocol, and sends it to A. In turn, A computes the same hash function on the true value of a, and compares the two. If they agree, then it is assumed that the two values of a are the same, and the protocol terminates. We choose our hash functions in order to ensure that the probability of this happening when a and b do not in fact agree is small. The number of bits to represent this hash function is small in comparison to the number of bits needed to represent the messages exchanged, which is $O(\log n/\delta)$. If not, they double the distance, and repeat the protocol, until they agree. This requires $2 \log d$ rounds, and in the worst case transmits four times as many bits as if an exact value for d was available in advance.

6.3.2 Application to distances of interest

We will now show how Theorem 6.3.1 can be applied to the distances of interest. We shall make use of the definition of a non-trivial communication scheme (Definition 6.1.2).

Hamming distance is a metric on the space of strings of length n over an alphabet σ, and is induced by defining a symmetric relation on strings which differ in a single character. It is therefore an editing distance. For $l < n/2$, G_{2l} is not complete, and for any non-trivial n, G_{2l} will have degree higher than two: $\deg(G_l) = \sum_{i=1}^{l} \binom{n}{i} (|\sigma| - 1)^i$. There is no simple closed form for this expression, so instead we consider the upper bound on this quantity for up to l Hamming differences described in Lemma 6.2.1. This uses $l\log((|\sigma|-1)(n/l))+2$ bits, hence we have the result that up to l Hamming errors can be corrected using a single message with asymptotic cost at most $2l(\log((|\sigma|-1)(n/l))+2)$, which is non-trivial for $l < n/2$.

Edit distance is defined by charging unit cost to insertions, deletions or alterations of a character. Again, for $l < n/2$ (n denotes the length of the longer string), G_{2l} is not complete and will have degree more than two. Also, there is no concise exact expression for the degree of G_l, but we can use the binary encoding (Lemma 6.2.2) for an upper bound on $\log(\deg(G_l))$. We can encode up to e edit operations using at most $e(\log 2(|\sigma|(n+1)/e)) + (e\log\log n)$ bits, so $2\log\deg(G_l) \leq 2l(\log(|\sigma|(n+1)/l) + \log\log n)$ which is non-trivial for $l < n/2$.

Edit Distance with moves is an editing distance. Using the bitwise encoding, this gives an upper bound on the single message cost of $6d\lceil\log n\rceil$.

Compression Edit Distance is an editing distance. Using the simple encoding from Lemma 6.2.3 we can achieve a single message cost of $18l\log|a|$ bits, non-trivial for $l \leq \frac{|a|\log|\sigma|}{18\log|a|+|b|}$.

Permutation Edit Distance has only a single operation, of moving a single element. We can use the bitwise encoding to get a single message cost of $4l\lceil\log n\rceil$ bits, non-trivial for $l < n/4$.

Reversal Distance is found from unit cost reversals. Following the same argument, we get a single message cost of $4l\lceil\log n\rceil$ bits, non-trivial for $l < n/4$.

Transposition Distance is similar to reversal distance, and yields a single message cost of $6l\lceil\log n\rceil$ bits, non-trivial for $l < n/6$.

Swap Distance is similar to Permutation Edit Distance, and can have a single message cost of $2l\lceil\log n\rceil$ bits, non-trivial for $l < n/2$.

RITE Distances allow combinations of the various operations, reversals, indels, transpositions and edits, each at unit cost. By using a bitwise coding, we find a cost of $6l\lceil\log 2(|P|+|Q|)\rceil$ bits, which is non-trivial for $l < |P|/12$ (if P and Q are approximately the same length).

So for all the metric distances described here, a single difference correcting message can be constructed which has size $O(l\log n)$, with low coefficients. They are all efficient with respect to their corresponding distances. Almost all of them are non-trivial schemes for up to a linear number of differences, which covers all practical situations. When there is a linear number of differences, then the two sequences will look very different indeed.

6.3.3 Computational Cost

While these protocols are communication-efficient, they are computationally expensive. This expense comes from the requirement to construct a graph of all possible objects. For example, for permutation edit distance, there are $n!$ possible permutations of length n, so constructing the nodes of this graph requires $O(n!)$ time and space. Once the graph has been constructed the other operations are polynomial in the size of the graph; however, since the graph is exponentially large, these operations take time exponential in n. Firstly, G_1 can be found as the graph with edges between each pair of

	a	b
Bit-string	1101 0100 0101 0011	1001 0100 0101 0011
Parity of whole string (locations 0-15)	0	1
Parity of locations 8,9,10,11,12,13,14,15	0	0
Parity of locations 4,5,6,7,12,13,14,15	1	1
Parity of locations 2,3,6,7,10,11,14,15	0	0
Parity of odd locations	0	1

A calculates the colour of the bitstring a using parities to get 00100, which is sent to B. In turn, B finds the colour of b as 10101. By comparing the colours in a bitwise fashion, B can immediately find the location of the difference and hence work out what the string a is. The exclusive-or of the colours is 10001. The first bit indicates that a is not identical to b; the next four bits can be read as a binary integer to show that the differing bit occurs at position 1.

Figure 6.1: Calculating a colour using Hamming codes

nodes that have a distance of one between their respective objects. In the permutation edit distance example, this can be done in time $O(n^2 n!)$, since each permutation is a distance one from $O(n^2)$ others. Constructing G_{2l} from G_1 can be done in time $O((n!)^2 \log l)$, and finally colouring this graph can be done greedily in time linear in the size of the graph [Gib85]. So overall, the cost will be bounded by the most expensive operation, computing G_{2l}, which is $O(\log l (n!)^2)$. This is clearly impractical for all but very small values of n.

However, this technique is not solely of theoretical interest. Firstly, observe that since this procedure generates a single message, this message can be thought of as an error-correcting code to correct errors of the corresponding distance model. This proof shows that error-correcting codes for these models of error (transpositions, insertions, deletion, block operations etc.) exist, since we have shown concrete methods to construct such codes, albeit in a fashion that is computationally intractable.

Secondly, note that the exponential cost of the method outlined is not an inherent part of constructing the message. Given a sequence a, we wish to give it a colour so that its colour is unique amongst all other objects within a distance d of another sequence b. To ensure that both parties agreed on the colouring, we arranged for them both to construct the same graph from scratch and colour it using the same algorithm. What is instead desirable is if we could directly use a deterministic function to find the colour of an object. We would like that both the function and its inverse be computable in polynomial time. Then the overall computational cost would be polynomial rather than exponential. In fact, in some cases this is possible. In particular, the similarity of our colourings to error-correcting codes means that we can use techniques from the field of error-correcting codes to get our desired colourings. We show how this is possible for several of the distances under consideration.

Hamming Distance

The majority of coding theory focuses exclusively on the problem of correcting errors that change characters in place. It was precisely this situation that originally gave rise to the concept of the Hamming distance [Ham80]. We can use techniques from coding theory to efficiently compute the kinds of colours that are required. An example of this is given in Figure 6.1 which uses a Hamming code to allow the exchange of documents with up to a single difference. In this special case, the colour is calculated in $O(n \log n)$ time, and requires $\log n + 1$ bits, which matches the lower bound in this case where $h = 1$ and so is optimal. For the general case, Reed-Solomon codes can be used to create a colour for a string using $2h \log n$ bits to correct up to h differences. This is essentially the technique used in [AGE94]). We can extend this approach to certain other distances by taking advantage of their embeddings into the Hamming distance.

Swap Distance

We saw in Section 3.2.1 that there is an exact, non-distortive embedding of swap distance into the Hamming distance. So for any given sequence, there is a corresponding binary matrix of size n^2, and the Hamming distance between two matrices is the swap distance between the corresponding sequence. So to find the colour for a sequence under swap distance, we can compute its binary matrix, and then treat it as a binary string of length n^2, and compute the colour accordingly. To correct up to h differences, a message of size $2h \log n^2 = 4h \log n$ bits will suffice. One point to note is that swap distance can be as high as $n(n-1)/2$, which occurs between a permutation and its reverse. This protocol is only non-trivial for $h < n/4$.

Reversal Distance, Transposition Distance, RITE distances

These distances can also be embedded into the Hamming distance, although with distortive factors of between 2 and 3. Each permutation generates a binary matrix of size n^2 bits. The same approach can be used as above: we treat the matrix as a bit-string of length n^2, and use an error-correcting code to generate a colour. If the distance (reversal, transposition) is d, then the Hamming distance between the bit-strings will be between d and $2d$. Hence we can use an error-correcting code to correct up to $2d$ differences. The cost of this will be $2 \cdot 2d \log n^2 = 8d \log n$ bits for transposition distances which is non-trivial for distances up to $n/8$. With a little care, the same can be achieved for RITE distances. The embedding of RITE distances is into the L_1 metric; we can use the trivial embedding of L_1 into Hamming distance (in Section 7), and so gain a single message cost of size $O(d \log n)$.

6.4 Computationally Efficient Protocols for String Distances

The techniques described in Section 6.3.1 are computationally intractable for several of our distances, and not applicable to some of the others. Even in the cases for which an efficient colouring procedure is available, the computational cost of this may become excessive for very large strings or permutations. We therefore seek alternative protocols for document exchange which are not only relatively efficient in terms of communication, but also in terms of computation required. We shall adopt a divide-and-conquer approach, and tackle the problem over a series of rounds. We will make use of hash functions for probabilistic matching of substrings. In some cases, the approaches that we describe have been described independently in the literature already; however, we will set these methods into a common framework and additionally analyse the communication cost and probability of success, which has not been rigorously performed before.

We first consider an abstract problem which will be the basis for the solution of the problems under consideration. We are given a string of length n, and h of its characters are 'distinguished'. We can use queries of the form "does this substring contain any distinguished characters?" which give a yes or no answer. Our goal is to locate all h such characters. A simple solution is to divide the string into two equal size substrings and query them. If a substring is free of distinguished characters, then we do not consider it further. Otherwise it is further divided into two halves each of which is queried inductively. Clearly, this approach will find all the distinguished characters; the question is, how many queries will it require. We first define a simple parsing technique to build a binary tree on a sequence.

Definition 6.4.1 *The* binary parse tree *of a sequence a is defined by repeatedly pairing adjacent entries. The result is a binary tree of height $\lceil \log |a| \rceil$. It can be made iteratively: if $a = a[1]a[2] \dots a[n]$ then the first level consists of the nodes $a'[i] = (a[2i-1], a[2i])$ for $i = 1 \dots n/2$ (we assume for simplicity that n is a power of* *two: a can be 'padded' with null characters if this is not the case). This pairing procedure can be repeated on the resulting sequences until the new sequence is of length 1. An example of a binary parse tree is given in Figure 6.2.*

Lemma 6.4.1 *An upper bound on the number of queries made in this protocol is $2h \log(2n/h)$.*

Proof. The simple divide and conquer procedure effectively searches the binary parse tree of the string (from Definition 6.4.1). We consider how the distinguished characters could be placed in the string given the fixed search procedure. At any given level of the binary parse tree of the string, h distinguished characters will necessitate $2h$ different queries if they are in separate subtrees. This is because we might know h different substrings that contain a single distinguished character each. The protocol would split each of these substrings into two and make a query on each. So to maximise the number of queries, distinguished characters must occupy as many distinct subtrees as possible. Starting from the root of the binary parse tree and working down, we can arrange for the distinguished characters to occupy the first $\log h$ levels completely, requiring a query for every node in those levels of the parse tree. At the next level, there are more than $2h$ substrings, so it is not possible to require a query of each of them; instead, to maximise the number of queries, we ensure that no two distinguished characters are placed in the same subtree. This requires $2h$ queries at each successive level of the tree until the leaves are reached. Following this pattern, we must query all $2h-2$ substrings in the first $\log h$ levels, and $2h$ substrings in each of the remaining $\log n - \log h$ levels. By construction, we could not have arranged the distinguished characters to have any more queries, and so this is an upper bound. □

Each query gives us one bit of information, so here we are gaining no more than $2h \log(n/h) + O(h)$ bits of information. This is asymptotically only twice the minimum number of bits required to locate all the distinguished characters. This follows from Lemma 6.2.1 if we treat the location of Hamming differences as distinguished characters. We will now apply this abstract problem to our various distance measures.

6.4.1 Hamming distance

The problem of locating the Hamming errors, given an upper bound on the number of errors, is similar to the problem of group testing (also observed by Madej [Mad89]). We wish to group samples and perform a test which will return either 'all the same' or 'at least one mismatch'. The differences are that we have an ordering of the samples, given by their location in the string, and that we have to contend with the problem of false negatives. We use negative to mean that the test indicates no errors; false negatives are an inevitable consequence of using fewer than n bits of communication to locate the Hamming differences, since we are checking whether substrings are identical. Deterministically comparing two strings for equality requires n bits of communication; hence we shall be using probabilistic tests to reduce the communication load, which have some probability of failure. Our tests will be based on fingerprinting with hash functions such as those from Lemma 2.1.1. If a substring in one string hashes to the same value as the corresponding substring in the other, then we take this as evidence that the two substrings match. Note that there is no danger of false positives; if a test leads us to believe that the substrings are different, then there is zero probability that they actually match.

The following scheme is a straightforward way to locate and correct differences, and is similar to that proposed in [Met83]. Mapping our terminology onto that of the above problem, the locations of Hamming differences between the two strings are the distinguished characters. We shall use the proposed simple divide and conquer algorithm to locate them. The algorithm is illustrated with an example in Figure 6.2. The test we perform involves communicating to make the queries: in each round, one party will send the value of a hash function for each of the substrings in question, in the order they occur in the string. The other party will calculate the value for the corresponding substrings of their

Algorithm 6.4.1 *Run by A who holds a*

```
  $range \leftarrow [0, n-1]$
  repeat
    for all $[l, r]$ in $range$ do
      append hash($a[l, (l+r)/2-1]$) to $output$
      append hash($a[(l+r)/2, r]$) to $output$
    send $output$
    receive $bitmap$
    $newrange \leftarrow empty$
    for all $[l, r]$ in $range$ do
      dequeue $bit_1, bit_2$ from $bitmap$
      if $bit_1 = 1$ then
        append $[l, (l+r)/2-1]$ to $newrange$
      if $bit_2 = 1$ then
        append $[(l+r)/2, r]$ to $newrange$
    $range \leftarrow newrange$
  until cheaper to send characters
  for all $[l, r]$ in $range$ do
    append $a[l, r]$ to $output$
  send $output$
```

Algorithm 6.4.2 *Run by B who holds b*

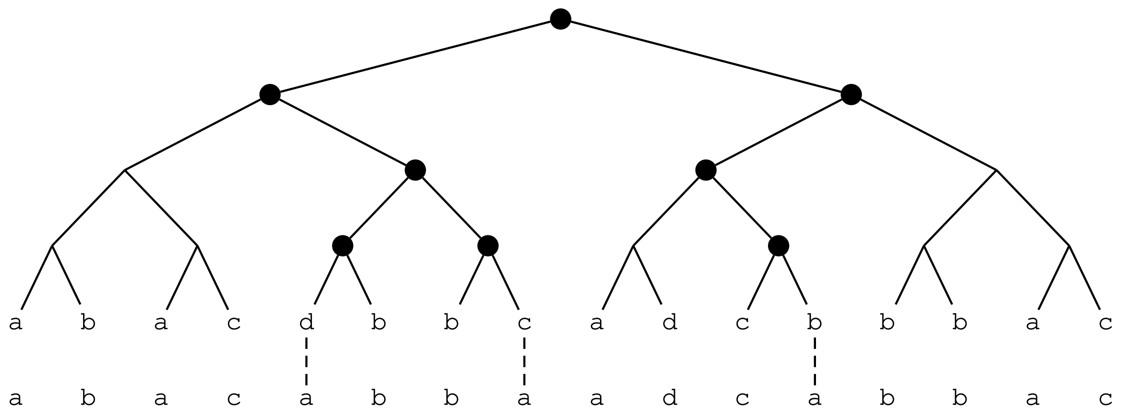
```
  $range \leftarrow [0, n-1]$
  repeat
    $newrange \leftarrow empty$
    receive $hashes$
    for all $[l, r]$ in $range$ do
      dequeue $hash_1, hash_2$ from $hashes$
      if hash($b[l, (l+r)/2-1]$) $\neq hash_1$ then
        append 1 to $bitmap$
        append $[l, (l+r)/2-1]$ to $newrange$
      else
        append 0 to $bitmap$
      if hash($b[(l+r)/2, r]$) $\neq hash_2$ then
        append 1 to $bitmap$
        append $[(l+r)/2, r]$ to $newrange$
      else
        append 0 to $bitmap$
    send $bitmap$
    $range \leftarrow newrange$
  until cheaper to send characters
  $a \leftarrow b$
  receive $characters$
  for all $[l, r]$ in $range$ do
    $a[l, r] \leftarrow$ next($l - r + 1$) $characters$
```

string. If they agree, then there is (with high probability) no difference between the two substrings; otherwise there is (with certainty) some discrepancy between them. The reply to the message is a bitmap indicating the success or failure of each test in order — say, using 1 to indicate that the hashes did not agree, 0 if they did. Every substring that was not identified is split into two halves, and the procedure continues on these new substrings until it is more expensive to send the hashes than to send the unidentified substrings. There are clearly no more than $2 \log n$ rounds, since A can only split substrings of a in half $\log n$ times. This procedure is outlined in pseudocode — the algorithm run by A is given in Algorithm 6.4.1, that run by B in Algorithm 6.4.2. We must choose our hash functions so that the overall probability of success of the procedure is high.

We will make use of the family of Karp-Rabin hash functions described in Section 2.1.3. These take a paramter δ and ensure that, over all choices of hash functions from the family, that the probability of two arbitrarily chosen sequences having the same hash value is δ'. We would like that, after performing all the tests, the probability of them all succeeding is at least a constant. Using the union bound, if we perform t tests, then the probability of them all succeeding is at least $1 - t\delta'$. Therefore, if we choose $\delta' = \delta/t$, for some constant δ', then we guarantee that overall the procedure has probability at most δ of failing.

Theorem 6.4.1 *With an estimate of the Hamming distance, $\hat{h}$, if A runs Algorithm 6.4.1, and B runs Algorithm 6.4.2, this protocol is efficient with respect to the Hamming distance. It communicates no more than $2h \log 2n/h$ hash values of $O(\log n + \log 1/\delta)$ bits each, and succeeds with probability $1 - \delta$.*

Proof. Lemma 6.4.1 says that we must make at most $2h(\log 2n/h)$ queries (note that this is h, not $\hat{h}$). We fix δ and so we will choose δ' so that $\delta' = \delta/(2\hat{h}(\log 2n/\hat{h}))$. To represent the values of the hash function that are exchanged requires $\log(n/\delta' \log n/\delta')$ bits, which is $\log(2n\hat{h}/\delta \log(2n/\hat{h}) \log(2n\hat{h}/\delta \log 2n/\hat{h}))$. This is $O(\log n/\delta)$. In total, we communicate at most $2h(\log 2n/h)$ hash values. For each hash that is sent, the other party sends one bit in response. We must also communicate the prime, p, that is used in


a b a c d b b c a d c b b b a c
a b a c a b b a a d c a b b a c

The lower string is treated as a, the upper is b. The Hamming distance between the two strings is 3. Blobs mark the differing substrings which are discovered in the traversal of the search tree; all other substrings are unchanged. The rounds progress as follows:

- **Round 1** A sends hashes $[0, 7], [8, 15]$
- **Round 2** B replies with 11 (indicating that both hashes disagreed)
- **Round 3** A sends hashes $[0, 3], [4, 7], [8, 11], [12, 15]$
- **Round 4** B replies with 0110
- **Round 5** A sends hashes $[4, 5], [6, 7], [8, 9], [10, 11]$
- **Round 6** B replies with 1101
- **Round 7** A sends the characters `abbaca`
- **Finish** B now knows what the string a is.

Figure 6.2: Illustrating how Hamming differences affect the binary parse tree

the hash functions, which also has size $O(\log n/\delta)$. Combining all these gives the communication cost. □

This protocol is *non-trivial* for $h = O(n/(\log n \ \log\log n))$. The overall cost is a factor of $O(\log(\hat{h}/\delta \log n))$ above the optimal; however, this cost comes from the size of the hash values sent. With regard to the number of hash values sent, we have shown it is about twice the optimal of any possible scheme which uses hash functions to give a binary answer to a question. The number of rounds is precisely $2 \log n - 1$: we can progress one level of the binary parse tree every other round.

Theorem 6.4.2 *The computational complexity of Algorithms 6.4.1 and 6.4.2 is* $O(n \log h)$

Proof. We make use of some of the observations of Lemma 6.4.1. Considering the binary parse tree of the string, once we have passed the level $\log h$, in the worst case we have to deal with $2h$ strings whose total length is $O(n)$. In this level, the parties must do $O(n)$ work to compute the hashes. In each subsequent level, the substrings under consideration can be no more than half the maximum length of those of the level above. So the total cost of these must be $O(n)$. In the worst case, we need to find hashes for every substring in the binary parse tree above level $\log h$. To do this costs $O(n)$ for each level, giving the claimed cost. □

Note that although an estimate of the Hamming distance is required for choosing the size of the hash functions, the number of queries depends on the true Hamming distance. In the absence of a good bound on the Hamming distance, the trivial upper bound $h \leq n$ can be used.

Depending on the choice of the hash function, it may be possible to halve the amount of information sent: if linear hashes are sent then the hash value for the second half of a substring can be calculated from the hash value of the first half and the whole substring. Additionally, by pre-computing the hashes of every substring bottom-up, the computational complexity of the protocol can be reduced from $O(n \log h)$ to $O(n)$. Otherwise, if hashes are independent, then stored values can be used to check that the procedure has succeeded. This comment applies to all of the hierarchical schemes presented in this section.

6.4.2 Edit Distance.

We next translate the above approach to deal with the edit distance. A similar scheme is proposed but not analysed in [SBB90].

It is not immediately apparent how to map the edit distance problem onto the abstract problem presented above. However, observe that the way that differences were located was by a process of elimination: identical substrings were found, until all that remained were disparate substrings. We may use the same kind of hierarchical approach to identify common fragments. It is no longer the case that substrings are aligned between the strings; however, we know that matching substrings will be offset by at most the edit distance.

The party B receiving the hash values must therefore do more work to identify them with a substring. B has a bound $\hat{d}$ on the edit distance; if a substring is unaltered by the editing process, then it will be found at a displacement of no more than $\hat{d}$ from its location in A's string. So B calculates hashes of substrings of the appropriate length at all such displacements left and right from the corresponding position in its string, testing each one to see if it agrees with the sent hash. If they do agree, it is assumed that they match, and so B now knows the substring at this location in A. The outline pseudocode for this is given in Algorithm 6.4.3.

We consider an optimal edit sequence from a to b of length d and how it affects the binary parse tree representing the splitting of a. A replacement has the same effect as in the Hamming case — it affects every ancestor of the changed character in the binary tree, but nowhere else in the tree. As shown in Figure 6.2, each affected node in the binary parse tree can cause up to two hashes to be sent. The same idea can be applied to the other operations. A deletion also affects characters at the leaf nodes, and every ancestor of that node (it could be thought of as a change of the original character to a null character, '$-$'). An insertion of any number of consecutive characters between two adjacent characters is considered to occur at the internal node which is the lowest common ancestor of the pair. It therefore changes every ancestor of the affected node. The protocol must traverse this tree, computing hashes on every node whose subtree contains any of these edit operations. This is illustrated in Figure 6.3.

Theorem 6.4.3 *If A runs Algorithm 6.4.1 and B runs Algorithm 6.4.3 then this protocol is efficient with respect to the edit distance. It communicates no more than $2d \log(2n/d)$ hash values, each of which is of size $O(\log n/\delta)$ bits and succeeds with probability $1 - \delta$.*

Proof. Observe that the worst case is exactly as before — if (almost) all errors are deletions or alterations (since we have to descend further down the tree to discover these). Certainly, in the worst case, the number of hashes sent will be that given by Lemma 6.4.1, $2d \log(2n/d)$. We must compare each hash sent with up to $2\hat{d} + 1$ others in the worst case. As before, we want the probability of an error to be no more than a constant, which gives $\delta' = 2d \log(2n/d)(2\hat{d} + 1)\delta$. Following the same line of argument as in Theorem 6.4.1, we compute hashes of size $O(\log(\frac{2}{\delta} n\hat{d}(2\hat{d} + 1) \log(2n/d)))$ bits, which is $O(\log n/\delta)$ since $d \leq n$. $\square$

Corollary 6.4.1 *The computational complexity of this protocol is $O(n \log n)$ hash computations.*

Proof. The time complexity for A is clearly $O(n \log d)$, since A makes the same computations as before, by running Algorithm 6.4.1 — we have already shown that the worst case number of hashes sent is the

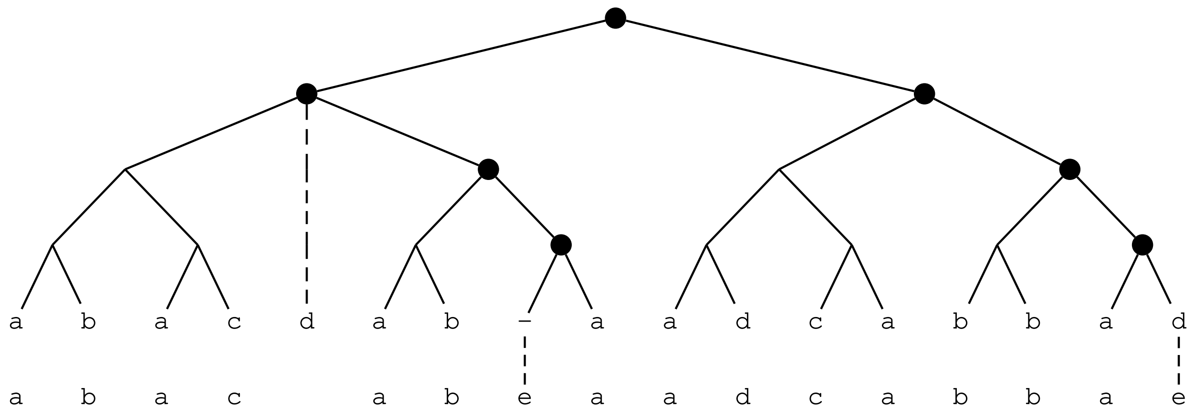
Algorithm 6.4.3 *Run by B who holds b*

```
range ← ([0, n − 1])
repeat
  receive hashes
  newrange ← empty
  for all [l, r] in range do
    w = (r − l)/2
    dequeue hash_1, hash_2 from hashes
    if ∄1 ≤ i ≤ n − w : hash(b[i, i + w]) = hash_1 then
      append 1 to bitmap
      enqueue [l, l + w] to newrange
    else
      append 0 to bitmap
      a[l, l + w] ← b[i, i + w]
    if ∄1 ≤ j ≤ n − w : hash(b[j, j + w]) = hash_2 then
      append 1 to bitmap
      enqueue [r − w, r] to newrange
    else
      append 0 to bitmap
      a[r − w, r] ← b[j, j + w]
    range ← newrange
  send bitmap
  range ← newrange
until cheaper to send remaining characters
receive characters
for all [l, r] in range do
  a[l, r] ← next(r − l + 1) characters
```

same. However, B has to do more work to align hash functions in Algorithm 6.4.3. Suppose at each level B computes the hash of every substring — there are $O(n)$ of these, and the hash values are $O(n)$ in length. We make the assumption here that given the hash for a substring $a[l : r]$, we can easily compute the hash for $a[l + 1 : r + 1]$. This is true of a large class of hash functions, including those described in Lemma 2.1.1. B can store each hash in a hash table (running a second hash function on the hash values if necessary). Working in the RAM model, in which manipulating hashes takes time $O(1)$, the time required to build the hash table is $O(n \log n)$: $O(n)$ work at each level. Searching the hash table takes time $O(1)$, the size of each hash. We follow the same procedure at each of the $\log n$ levels, giving a total cost of $O(n \log n)$. □

6.4.3 Tichy's Distance

Consider a variation of the above abstract problem. Instead of h distinguished characters, suppose that there are some number of "dividers" which are positioned between characters, such that there is at most one divider between any two adjacent characters. This problem is reducible to the distinguished character problem: there are $n - 1$ locations where dividers can be placed. If our queries are whether there are any dividers within a substring, then clearly the same protocol will suffice, with the same number of queries in the worst case. We consider two strings a and b whose Tichy distance is l. This means that a can be parsed into l pieces each of which is a substring of b. So we can use the same divide-and-conquer technique as before: for every hash value A sends, B searches to see whether there is any substring of b that hashes to the same value. If there is, then these substrings are identified; if not, the substring will be split in two and the procedure recurses on the two halves.


a b a c d a b – a a d c a b b a d
a b a c a b e a a d c a b b a e

The lower string is a, the upper is b. The edit distance between the two strings is 3. The parse tree of the string a is shown, with the editing operations to make b. Again, blobs mark the differing substrings which are discovered in the traversal of the search tree. A `d` is inserted after the fourth character — this affects the lowest common ancestor of its neighbouring nodes, but below this, their substrings are common to both a and b. The deletion of an `e`, the seventh character of a, and the change of the final character `c` to `e` both affect all their ancestors. The communication proceeds as follows

- **Round 1** A sends hashes on $[0, 7], [8, 15]$
- **Round 2** B replies with 11
- **Round 3** A sends hashes on $[0, 3], [4, 7], [8, 11], [12, 15]$
- **Round 4** B replies with 0101
- **Round 5** B sends hashes on $[4, 5], [6, 7], [12, 13], [14, 15]$
- **Round 6** B replies with 0101
- **Round 7** A sends the characters `eaae`

Figure 6.3: Illustrating how edit differences affect the binary parse tree of a

Theorem 6.4.4 *If A runs algorithm 6.4.1 and B runs Algorithm 6.4.3 then this protocol uses no more than $O(l \log(2n/l) \log n/\delta)$ bits of communication. It succeeds with probability $1 - \delta$.*

Proof. If A sends a hash of a substring that is also a substring of b, then B can identify the substring. Since a is formed from l substrings of b, it follows that a hash will only fail to be identified if the substring of a overlaps more than one substring of b. We can imagine a written out with $l - 1$ markers between characters indicating the start and end of each of the substrings of b. As noted above, this is equivalent to the original abstract problem: if a hashed substring of a contains no markers, then it can certainly be identified by B; if it contains markers then it may not be identified (if we are lucky, it might be, but we shall take a worst-case analysis). Therefore, we need to send no more than $2l \log(2n/l)$ hash values.

We now need to calculate the size of the hash value necessary to ensure that the probability of success is at least a constant. For each hash received from A, this has to be compared with up to n others as we try every offset within b. So we make at most $2ln \log(2n/l)$ comparisons, and following the same pattern as in the above sections, we need to choose δ' to be at least $2ln \log(2n/l)\delta$. If we have no *a priori* bound on l, we will have to use the fact that l is at most n. This gives the size in bits of the hash value as $O(\log n^3/\delta)$ which leads to the stated communication cost. As with the other protocols, the number of rounds is logarithmic in the length of a, since the size of substrings being handled halves in each round. □

The computational complexity of this protocol is identical to the protocol for edit distance, $O(|b| \log |b|)$, since the algorithms being run are the same, and we have shown that they obey the same bound on the number of hashes.

6.4.4 LZ Distance

Since we have shown how to deal with Tichy's distance, the LZ distance, which is similar in nature, follows fairly directly. We again take the same divide-and-conquer approach. The string a can be parsed into l substrings, which are either substrings of b or substrings which occur earlier in a. Rather than descending the tree in parallel, the protocol performs a left-to-right depth-first search for identifiable substrings of a. At each stage B tries to resolve the pair of hashes that have been sent by looking for a substring of b or the partially built a that hashes to the same value. B first considers the hash of the left substring, and attempts to resolve that. If B cannot resolve the hash, then this is indicated to A, who splits the substring into two, and sends hashes of each of these halves. If B can resolve the left substring, then B proceeds to the right substring, and follows the same procedure.

Theorem 6.4.5 *If A runs Algorithm 6.4.4 and B runs Algorithm 6.4.5, then this protocol uses $O(l \log(2n/l) \log n/\delta)$ bits of communication. It succeeds with probability $1 - \delta$ and the number of rounds involved is at most $2l \log(n/l)$.*

Proof. In an optimal parsing, a is parsed into l pieces, each piece of which is a substring present in b or earlier in a, or a single character. We can imagine that b has $l - 1$ "dividers" that separate the substrings in this parsing. If a hash falls between two dividers, then it can be resolved. The number of hashes necessary for B who holds b to identify a is that needed to locate the $l - 1$ dividers, which is given by Lemma 6.4.1. This is $2(l-1)\log(2\frac{n-1}{l-1})$, which is less than $2l \log(2n/l)$.

We are performing many more comparisons between hash values than before, so we require a larger base over which to compute hashes in order to ensure that the chance of a hash collision is still only constant for the whole process. We only compare hashes for substrings of the same length. If we have two strings each of length n then there are fewer than $2n$ substrings of any given length. So there are fewer than $2n^2$ possible pairwise comparisons. The lemma then follows. □

Algorithm 6.4.4 *Run by A who holds a*

```
push $[0, n-1]$ onto $rangestack$
repeat
  pop $[l, r]$ from $rangestack$
  $m \leftarrow (l+r)/2$
  if cheaper to send characters then
    send $a[l, r]$
  else
    send hash($a[l, m-1]$)
    send hash($a[m, r]$)
    receive $bit_1, bit_2$
    if $bit_2 = 1$ then
      push $[m, r]$ to $rangestack$
    if $bit_1 = 1$ then
      push $[l, m-1]$ to $rangestack$
until $rangestack$ is empty
```

Algorithm 6.4.5 *Run by B who holds b*

```
push $[0, n-1]$ onto $rangestack$
repeat
  pop $[l, r]$ from $rangestack$
  if cheaper to send characters then
    $a[l, r] \leftarrow$ receive $characters$
  else
    $m \leftarrow (l+r)/2$
    $o \leftarrow (r-l+1)/2$
    receive $hash_1$, $hash_2$
    if $/\ \exists j$ : hash($b[j, j+o]$) $=$ $hash_2$ or hash($a[j, j+o] = hash_2$ then
      push $[m, r]$ onto $rangestack$; $bit_2 \leftarrow 1$
    else
      $a[m, r] \leftarrow b[j, j+o]$; $bit_2 \leftarrow 0$
    if $/\ \exists i$ : hash($b[i, i+o]$) $=$ $hash_1$ or hash($a[i, i+o] = hash_2$ then
      push $[l, m-1]$ to $rangestack$; $bit_1 \leftarrow 1$
    else
      $a[l, m-1] \leftarrow b[i, i+o]$; $bit_1 \leftarrow 0$
    send $bit_1, bit_2$
until $rangestack$ is empty
```

Evfimievski [Evf00] considers what turns out to be the same distance measure, and claims that it requires no more than $3l \log n$ hashes to be sent. With a more rigorous analysis, we have improved this to $2l \log n/l$. The time complexity of our protocol is $O((|a| + |b|) \log(|a| + |b|))$ hash function manipulations. The procedure to achieve this complexity is essentially the same as described in Section 6.4.2, except that in addition to keeping tables of hashes of b, B must additionally add hashes of the received parts of a, yielding the slightly higher time complexity.

6.4.5 Compression Distances and Edit Distance with Moves

We develop the above protocol for LZ distance further to cope with Block Edit Distances. Suppose that the Compression Distance between a and b is c. Instead of using the simple divide-and-conquer approach based on a binary parse tree, we shall make use of the more structured parsing given by the ESP tree of the strings. Initially A forms the ESP parsing of the string a, and begins by sending the hash of the substring represented by the top level node in the parsing. B also finds the ESP parse tree of the string b. On receiving hashes of nodes, B attempts to find substrings represented by nodes of b that hash to the same value, and indicates whether they could be found with a bitmap sent to A. A then examines the node that occurs earliest in a which could not be resolved by B, and descends a level in the parse tree, splitting the node into up to three child nodes. A then sends hashes on the substrings represented by each of these nodes, and proceeds in this fashion until B has resolved the whole string.

Lemma 6.4.2 *If A runs Algorithm 6.4.6, and B runs Algorithm 6.4.7 then the number of hashes sent under this scheme is at most* $24c \log n(\log^* n + 10)$

Proof. The first time that B encounters a hash that cannot be resolved is when A sends the hash of a substring that is in the parsing of a but not in the parsing of b. Once this node has been identified, whenever it occurs elsewhere in a it can now be identified by B. Hence each node that is in $ET(a)$ but not in $ET(b)$ generates at most 3 hashes. This is very similar to the proof of Theorem 4.4.1, and in

Algorithm 6.4.6 *Run by A who holds a*

```
send $n$, $height$
push ESProot($a$) onto $nodestack$
repeat
  pop $node$ from $nodestack$
  while cheaper to send characters do
    send characters($node$)
    pop $node$ from $nodestack$
  send hash($node$)
  receive bit
  if $bit = 1$ then
    for all children of $node$ do
      push child($node$) onto $nodestack$
until $nodestack$ is empty
```

Algorithm 6.4.7 *Run by B who holds b*

```
receive $n$, $height$
$a \leftarrow null$
repeat
  while cheaper to send characters do
    receive $characters$
    append $characters$ to $a$
  receive $hash_x$
  if $\nexists node : \text{hash}(node) = hash_x$
   then
    send 1
  else
    send 0
    append $node$ to $a$
until length($a$) = $n$
```

total $3|T(a) \backslash T(b)|$ hashes are exchanged (recall the definition of the transformation T from Section 4.2). $3|T(a)\backslash T(b)| \leq 3|T(a)\Delta T(b)| \leq 3 \cdot 8\, c(a,b) \log n(\log^* n + 10)$, as shown in Theorem 4.4.1. □

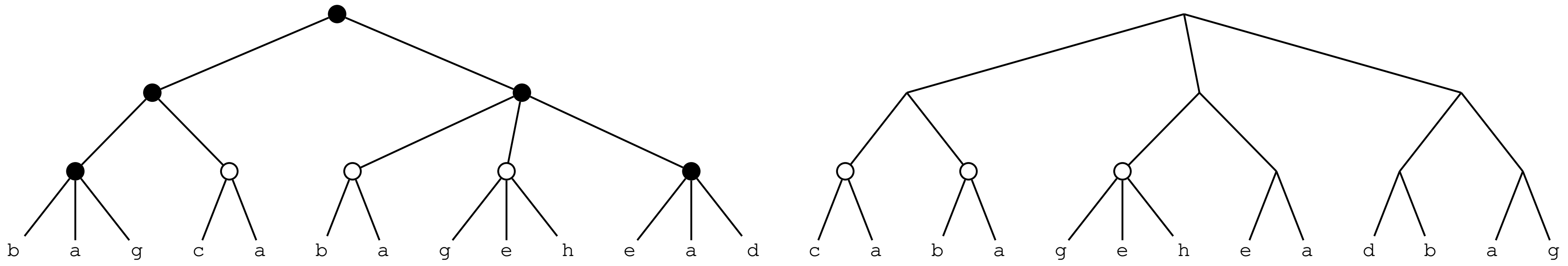
Theorem 6.4.6 *If A runs Algorithm 6.4.6, and B runs Algorithm 6.4.7, this protocol is efficient with respect to the Compression Distance, using $O(c(a,b)\log^2 n \log^* n)$ bits of communication.*

Proof. As observed in Lemma 6.4.2, this protocol exchanges $O(c(a,b)\log n \log^* n)$ hashes. Each hash needs to be compared to at most $O(n)$ hashes of substrings, and a trivial bound on the block edit distance is n. Putting all this together, we choose an appropriate hash size of $O(\log n)$ bits, and follow the usual line of argument to reach the desired result. □

Lemma 6.4.3 *The time complexity of this protocol is $O(n \log^* n)$.*

Proof. The time complexity of this protocol is slightly lower than those of others we have considered, since we can take advantage of properties of linear hash functions and the tree-structure of the ESP tree. It is dominated by the $O(n\log^* n)$ cost of computing the parse tree. Hashes need to be computed for each level of the parse tree. However, we only compare hashes of substrings corresponding to nodes in a parse tree, hence the hash for one node can be formed by a linear combination of the hashes of its children. Since each hash can be manipulated in time $O(1)$ in the RAM model, and the time cost is $O(n)$ hash manipulations, the total cost of making the hashes is $O(n)$. To allow the searching, these can be stored in a hash table, so that the total cost of the whole operation is $O(n)$. □

As can be seen in Figure 6.4, it might be advantageous to use a slightly different protocol: a hash is sent on the substring `bag` which B is unable to identify. However, `bag` is a substring of b and so could be identified if B searched all substrings, not just those that are nodes. So some communication could be saved, at the expense of some extra computation. Finally, we comment that the number of rounds can be improved to $\log n$ with some minor modifications to the protocol: instead of a depth-first approach, we can proceed down one level of the parse trees in each round. Some extra bookkeeping is necessary, but from the prior results on the nature of the parsing, the number of hashes exchanged is the same.



On the left is the ESP tree for A’s string, a; on the right is the parse tree for B’s string b. The protocol performs a depth first search of the tree of a, sending hashes on substrings from A to B. Nodes marked with a dark blob are present in a but not in b and so require a hash to be exchanged; nodes marked with a light blob are present in both a and b and so can be identified by B.

- A sends the length of a, 13, and a hash on the whole string, [0,12]
- B replies with 1
- A sends a hash on [0,4]
- B replies with 1
- A sends a hash on [0,2]
- B replies with 1
- A sends `bag`, then sends a hash on [3,4]
- B replies with 0
- A sends a hash on [5,12]
- B replies with 1
- A sends a hash on [5,6]
- B replies with 0
- A sends a hash on [7,9]
- B replies with 0
- A sends a hash on [10,12]
- B replies with 1
- A sends `ead` and the protocol terminates

Figure 6.4: The protocol for exchanging strings based on Compression Distance

Edit Distance with Moves

We observe that since Compression distance includes all the operations of Edit Distance with moves, it follows that the Compression distance between any pair of strings is no more than their Edit Distance with Moves. So since $c(a, b) \leq d(a, b)$, it follows that any scheme that is efficient with respect to Compression distance is also efficient with respect to Edit Distance with moves. This yields a corollary to Theorem 6.4.6.

Corollary 6.4.2 *If A runs Algorithm 6.4.6, and B runs Algorithm 6.4.7, this protocol is efficient with respect to the Edit Distance with Moves, d, using $O(d \log^2 n \log^* n)$ bits of communication.*

6.4.6 Compression Distance with Unconstrained Deletes

The protocol described above for Compression distance also suffices for the case when deletes are unconstrained. Let the unconstrained Compression Distance between a and b be du.

Theorem 6.4.7 *If A runs Algorithm 6.4.6 and B runs Algorithm 6.4.7 then this protocol uses $O(du \log^2 n \log^* n)$ bits of communication.*

Proof. As shown in Theorem 4.4.2, $du(a, b) \leq |T(b)\backslash T(a)| \cdot O(\log n \log^* n)$. If we follow the same argument as before, irrespective of which distance measure we are using, a can be constructed given b using at most $3|T(b)\backslash T(a)|$ hashes, which is at most $3|T(a)\Delta T(b)|$. Putting these together shows that the number of hashes exchanged is no more than $O(du \log n \log^* n)$. Since the protocol is identical in operation to that outlined above, it follows that the total amount of communication is as before, with du substituting for d. □

Evfimievski [Evf00] discusses this distance, and gives a protocol which is shown to require at most $336du^2 \log n$ hashes. The protocol is the same binary divide and conquer applied to LZ distance. The main result of the paper is in a proof which relates the LZ distance, l, and the Compression Distance with unconstrained deletes, du. He shows that $l \leq 112du^2$, which leads to the above bound. Our protocol, based on the ESP tree, exchanges $24du \log n(\log^* n + 10)$ hashes. It therefore uses a lesser amount of communication than Evfimievski's for the case where $14du > \log^* n + 10$. Note that if n represents the number of particles in the universe then $\log^* n$ is 5. We therefore claim that the above protocol uses fewer bits of communication in all practical cases. Further, the computation cost of our protocol for Block Edit Distance with unconstrained deletes is $O(n \log^* n)$, compared to the cost of $O(n \log^2 n)$ for the LZ distance which is used in [Evf00], and the number of founds can be made $\log n$ instead of $\Omega(du^2 \log n)$.

6.5 Computationally Efficient Protocols for Permutation Distances

Having shown efficient protocols based on divide and conquer for the string distances, we now show how the same approach can be applied to the permutation distances as well.

Swap Distance

We shall use the same divide-and-conquer approach as we did for the Hamming distance. A simple lemma suffices to show that this will be efficient in terms of the swap distance.

Lemma 6.5.1 $h(a, b) \leq 2\,\mathrm{swap}(a, b)$

Proof. Each swap interchanges two elements and leaves the rest as they were. Therefore, any single swap can alter the Hamming distance between the two strings by at most 2. So if the swap distance between a and b is $\mathrm{swap}(a, b)$, then the total change in Hamming distance can be at most twice this. □

Therefore, any document exchange protocol which is efficient with respect to the Hamming distance will also be efficient with respect to the swap distance. This immediately leads to a corollary to Theorem 6.4.1, substituting twice the swap distance s for the Hamming distance h.

Corollary 6.5.1 *The protocol for exchanging documents for Hamming distance can be run on permutations, with A running Algorithm 6.4.1 and B running Algorithm 6.4.2 on their respective sequences. This is efficient for the swap distance, and has a communication cost of no more than $O(s(\log n/s) \log n/\delta)$ bits. The computation cost is $O(n \log n)$ hash operations.*

Permutation Edit Distance

From the point of view of exchanging documents, Permutation Edit Distance is virtually identical to String Edit Distance. Each move operation has the same effect on the binary parse tree as a string edit operation — it affects leaf and internal nodes and hence causes a limited number of differences at various levels in the tree.

Theorem 6.5.1 *If A runs Algorithm 6.4.1 on permutation P, and B runs Algorithm 6.4.3 on permutation Q then this divide and conquer protocol for string edit distance is efficient with respect to permutation edit distance. It succeeds with probability $1 - \delta$ and communicates no more than $4d \log(n/d)$ hash values (where $d = d(P, Q)$).*

Proof. Each permutation can be treated as a string drawn from an alphabet of size n. Each move operation can be treated as a deletion followed by a re-insertion on a string. It therefore follows that the cost of exchanging permutations with a distance of d is no more than exchanging strings with a distance of $2d$. The proof is almost identical to that of Theorem 6.4.3. It has the same computation cost, $O(n \log n)$. □

Transposition Distance

We firstly give a lemma relating the Transposition distance between two permutations, P and Q ($t(P, Q)$), and the Tichy distance of the permutations, $\mathrm{tichy}(P, Q)$.

Lemma 6.5.2 $\mathrm{tichy}(P, Q) \leq 3t(P, Q) + 1$

Proof. Recall from Theorem 3.2.3 that the Transposition Distance between two sequences is bounded by 3 times the number of transposition breakpoints of P relative to Q. Between two consecutive breakpoints in P, the subsequence is identical to a subsequence of Q. In other words, if the transposition distance between P and Q is t, then P can be parsed into at most $3t+1$ substrings of Q. If this is the case, then the Tichy distance between P and Q cannot be any more than $3t$ for $t \geq 1$, since we have shown a way to parse P into at most $3t$ substrings of Q. Also, $\mathrm{tichy}(P, Q) = 0 \iff t(P, Q) = 0 \iff P = Q$. □

Since the Transposition distance is so closely related to the Tichy distance, the multi-round protocol for the Tichy distance, outlined above, is sufficient to exchange permutations in a fashion that is efficient in terms of the transposition distance. Applying the above lemma with Theorem 6.4.4 yields the following corollary.

Corollary 6.5.2 *The protocol for exchanging two documents for Tichy distance, Algorithm 6.4.1 and Algorithm 6.4.3, is also efficient with respect to their transposition distance, $t(P, Q)$. It has a communication cost of no more than $6t(P, Q) \log(n/2t(P, Q))$ hash values, and a computation cost of $O(n \log n)$.*

Reversal Distance

The same approach used to deal with transposition distance will also work for Reversal distance, with a small alteration. Transposition breakpoints denoted the junctions between substrings of the original sequence; reversal breakpoints mark the junction between substrings of the original sequence that may have been reversed. Thus the string can still be parsed in substrings of the original sequence — in this case, into at most $2r + 1$ such substrings — but some of these may be reversed. So we can use the same protocol again for exchange, that for Tichy's distance, but here we make an alteration: when B is searching for substrings that match the received hashes, b will be considered both forwards and reversed. Algorithm 6.4.3 must be modified so that when hash matches are being searched for, the reversed string must also be searched; but other than that, the algorithm is unchanged. It follows that this scheme will use the same number of hashes as a string whose Tichy distance is at most $2r$, though we will have to make twice as many hash comparisons. We therefore gain another corollary to Theorem 6.4.4.

Corollary 6.5.3 *The modified protocol for exchanging documents for Tichy distance using Algorithm 6.4.1 and the modified Algorithm 6.4.3 is efficient with respect to the reversal distance. It has a communication cost of no more than $4r \log(2n/3r)$ hash values, and a computation cost of $O(n \log n)$.*

Compound Permutation Distances

Lemma A.1.1 showed that if we allow combinations of permutation operations — reversals, transpositions and editing operations — then this induced distance between permutations can be approximated by counting the number of reversal breakpoints, and that this gives a 3-approximation. The above protocol exchanges sequences on the basis of the number of reversal breakpoints, and so will solve this problem. We gain a further corollary to Theorem 6.4.4.

Corollary 6.5.4 *If A runs Algorithm 6.4.1 and B runs the modified Algorithm 6.4.3 on their respective permutations then this modified protocol is efficient with respect to the compound permutation distance, τ. It has a communication cost of no more than $6\tau \log(n/2\tau)$ hash values and computation cost of $O(n \log n)$ hash operations.*

Allowing Indels

We have seen in Lemma A.1.2 that permutation distances with insertions and deletions, where one of the sequences is allowed to be a string (denoted τ'') can be embedded into the L_1 distance with a distortion of at most 3. Each reversal breakpoint in a relative to b can be thought of as the junction in a between two (possibly reversed) substrings of b. It therefore follows that the same modified protocol will suffice, giving one final corollary to Theorem 6.4.4.

Corollary 6.5.5 *The protocol of Algorithm 6.4.1 and the modified Algorithm 6.4.3 is efficient with respect to the compound permutation distances with indels. It has a communication cost of no more than $6\tau'' \log(2n/3\tau'')$ hash values and a computation cost of $O(n \log^2 n)$.*

6.6 Discussion

We have seen how two parties can communicate to exchange similar documents in a way that is much more efficient in terms of communication than sending the documents in full.

String Distance	**Metric**	**Lower bound**	**Single Round**	**Multi round hashes**	**Rounds**
Hamming Distance	Yes	$h \log(\|\sigma - 1\|)n/h$	$2h \log n(\|\sigma\| - 1)$	$2h \log 2n/h$	$\log n$
Levenshtein Edit Distance	Yes	$e \log 2(\|\sigma\| - 1)n/e$	$2e \log \|\sigma\|(n+1)/e$	$2e \log 2n/e$	$\log n$
LZ Distance	No	$2l \log n$	—	$2l \log 2n/l$	$l \log n/l$
Compression Distance	Yes	$9c \log n$	$18c \log 2n$	$24c \log n \log^* n$	$\log n$
Edit Distance with Moves	Yes	$3d \log 2n$	$6d \log 2n$	$24d \log n \log^* n$	$\log n$
Unconstrained Delete	No	$9du \log 2n$	—	$24du \log n \log^* n$	$\log n$
Tichy's Distance	No	$2l \log n$	—	$2l \log 2n/l$	$\log n$

Permutation Distance	**Metric**	**Lower bound**	**Single Round**	**Multi round hashes**	**Rounds**
Permutation Edit Distance	Yes	$2d \log n$	$4d \log n$	$4d \log n/d$	$\log n$
Reversal Distance	Yes	$2r \log n$	$4r \log n$	$4r \log 2n/3r$	$\log n$
Transposition Distance	Yes	$3t \log n$	$6t \log n$	$6t \log n/t$	$\log n$
Swap Distance	Yes	$\text{swap} \log n$	$2\,\text{swap} \log n$	$4\,\text{swap} \log n/\,\text{swap}$	$\log n$
RITE Distances	Yes	$3\tau'' \log 2n$	$6\tau'' \log 2n$	$6\tau'' \log 2n/3\tau''$	$\log n$

For each distance, we give a lower bound on the number of bits to exchange sequences; if the measure is a metric, then we give the single round cost based on the colouring protocols of Section 6.3. For the multi-round protocols based on hashing, we give the leading terms in the number of bits exchanged, and the number of rounds required.

Figure 6.5: Main results on document exchange

- We have seen how arguments based on graph colouring can achieve an amount of communication that is a factor of two above the lower bound for a large class of metric distances.
- For several of our distances, we have seen how computationally efficient protocols can achieve the single round cost owing to the structure of these metrics.
- We have described a number of protocols which sacrifice some efficiency in communication for computational tractability. These are all based on ideas of divide-and-conquer techniques using hash functions, some of which have been described before in the literature. For the first time we analyse the cost of these and give tight bounds on the exact number of bits communicated by these protocols.
- For a number of important distances, such as Tichy's distance, the LZ distance and the Block Editing distances, we give the first protocols or improved protocols to allow the efficient exchange of documents in terms of their distance. These draw on the analysis of these metrics and their embeddings from earlier chapters.

All the protocols described are *efficient with respect to* the distance measures employed — that is, the cost depends only linearly on the distance. The main results for this section in terms of the cost of communication are summarised in Table 6.5.

Chapter 7

Stopping

Goodbye to you, my trusted friend,
We've known each other since we were nine or ten.
Together we climbed hills and trees,
Learned of love and ABCs,
Skinned our hearts and skinned our knees.

[Jac73]

7.1 Discussion

By studying the use of embeddings we have made use of, and shown new results in, many areas of Computer Science, including Pattern Matching, Computational Biology, Data Compression, Information Theory, Databases, Coding Theory, Computational Geometry, Sublinear Algorithms, Graph Theory, Communication Complexity and Computational Statistics. In this final chapter we draw together some of the main themes of this thesis and discuss some possible extensions.

7.1.1 Nature of Embeddings

The majority of the novel embeddings presented in this thesis in Chapters 3 and 4 have been formed in a particular way: by recording the presence of certain features, or by counting certain features of sequences, and relating these to the distances of interest. In Chapter 3, looking at pairs of symbols was often sufficient to approximate permutation distances; in Chapter 4 a much more involved approach was necessary to perform the embedding based on recording the quantity of certain substrings of the sequence. But both methods yield what we will informally term "combinatorial" embeddings: they rely on combinatorial properties and give distortion factors which are essentially constants (in some cases, the constant depends on the size of the sequence). This is to be contrasted with the "geometric" embeddings described in Chapter 2, where the distortion of the embedding is a parameter, and determines the dimensionality of the target space.

The huge variety of different embeddings and techniques is hard to survey in any comprehensive way, but we can further distinguish the approaches here from other embedding works. A major result of embedding theory is that of Bourgain [Bou88] and reported in [Mat]: any metric space containing d points can be embedded into L_2 with a distortion of at most $O(\log d)$. For our approach to sequence distances, such a result is not as helpful as it may seem: our embeddings for strings and permutations are valid for all possible sequences, so the metric space contains as many points as there are sequences of length n. For binary strings of length n, there are 2^n possible strings, and so $\log d = n$. Since for our metrics the distance between any pair of sequences is at most n, a distortion factor of $O(n)$ is trivial (report the distance between every non-identical pair as n). Our embeddings (including those in Chapter 2) are computable without foreknowledge of all of the sequences that will be seen; Bourgain-style embeddings require full knowledge of the d points to compute the embedding. An in-depth survey of distance embeddings is given by Indyk [Ind01].

Ideally, we would want to discover efficiently computable embeddings of string and permutation distances into spaces such as Euclidean and Manhattan space that have distortion factors of $1 \pm \epsilon$. This seems a lot to ask for. Any such embedding would yield polynomial-time approximation schemes for distances, some of which are known to be MAX SNP-hard. Certainly, for reversal distance, constant distortion factors are the best that can be achieved, since this metric has been shown to be NP-hard to compute. This is also thought to be the case (but not proven) for transposition distance. Since permutations are a special case of strings, and have so far given tighter approximation factors than equivalent distances on strings, we would intuitively argue that this implies string distances are harder to deal with than permutations. However, the gap between small constant factors, and factors logarithmic in the length of the sequence is significant, and there is still cause to believe that there may be tighter approximation factors possible for string distances.

The negative results in this thesis do not entirely contradict this hope. The fact that permutation edit distance is hard to *estimate* says little about its hardness to *approximate* (Theorem 3.2.7). The fact that the Longest Common Subsequence requires $\Omega(|\sigma|)$ bits of communication to approximate (Theorem 3.2.8) says nothing about the hardness of the dual measure of edit distance. However, we argue that editing distances, and Levenshtein string edit distance in particular, are tough problems to deal with, and that any embedding to a vector distance or sketchable space is unlikely to have a small distortion factor. The result of Theorem 3.2.6 shows that there can exist no embedding of Permutation Edit distance (and also Levenshtein string edit distance [Mat02]) into L_1 or Hamming space with a distortion of less than $4/3$. With more advanced techniques, it seems likely that stronger lower bounds could be shown for these distances. It is still open whether constant factor embeddings are possible, or whether other (non-embedding) techniques could be used to address editing problems on permutations and strings. In the next section we discuss further the relation between strings and permutations.

7.1.2 Permutations and Strings

We have highlighted some of the gaps between the difficulty of dealing with permutations over strings: transposition and reversal distances can be found, sketched, and approximately matched against, up to a small constant factor for permutations. For strings, comparable distances based on the same fundamental operations — moving blocks around — have been approximated up to a logarithmic factor. This is a large difference, and a major question is whether this is inherent because of the difference in structures, or whether methods for strings are simply underdeveloped.

A concrete illustration of this is that String Edit Distance with Moves and Permutation Transposition Distance with Indels can be thought of as identical distance measures (they allow exactly the same operations: move a contiguous subsequence, insert and delete single symbols) but over different object domains (strings and permutations respectively). For permutations, we can make an embedding with a distortion factor of 2 (Section A.1.2), which holds when we allow one sequence to be a string. But as soon as we allow both sequences to be strings, then the best approximation factor we have for an embedding is $O(\log n \log^* n)$ with large suppressed constant factors. The question arises, is such a large factor a necessity when dealing with string distances?

We would claim that the power of permutations is in the fact that each symbol appears at most once and can be easily identified. As we saw in Section 3.2, the technique of looking at adjacent pairs which is powerful enough to uniquely define permutations is insufficient for strings. This occurs even if we allow one character to occur at most a small constant number of times. For example, if B, H, M, S are permutations with no common character between them, and a is some extra character, then the sequences $BaHaMaS$ and $BaMaHaS$ are indistinguishable based on looking at only symbol adjacencies, when one symbol occurs at most three times. Instead of symbol adjacencies, it seems that we need to adopt the method of building hierarchical structures on strings to be able to deal with editing operations. Any hierarchical structure with a constant branching degree at each level will have height at least $\log n$, and we would expect this factor to affect any method predicated on examining this many levels, since any operation will affect between one and $\log n$ entries in such a structure. This does not deny the possibility of an alternative approach that does not suffer from these weaknesses.

The gap between the difficulty of finding character based edit distance for strings and for permutations is a little easier to quantify. Traditional algorithms find the exact string edit distance between two strings (length n and m) in time $O(nm)$ using dynamic programming, or $O(nm/\log m)$ with the Four Russians technique [MP80]. The exact permutation edit distance can be found in time $O(n \log\log n)$ using an appropriate data structure. So both distances can be found in polynomial time, although string edit distance takes near quadratic time, whereas permutation edit distance is near linear. For embeddings of these distances, the results are weaker. We have seen how the permutation edit distance can be embedded into intersection size with a logarithmic distortion, whereas there are no known comparable embeddings for string edit distance. We discuss this further in Section 7.3 below.

In Section 3.2.6 we have also seen that problems of finding the distance between a string and a permutation seem no more difficult than between a pair of permutations, but are harder than between a pair of strings. This has useful implications for applications of string matching where the pattern string is known to have no repeated characters.

7.2 Extensions

Since this is a work on the distance between combinatorial structures, it is only reasonable to discuss problems of distance between structures other than those covered in detail.[1]. Many objects will degenerate into those we have considered already, as instances of sets, vectors, permutations or strings. Other important categories are trees and graphs, where there is already some body of work on measuring the distance between them.

7.2.1 Trees

Trees are a very natural extension of strings (with added structure), and are a good candidate for further investigation. It is possible to think of strings as a special case of trees: they can be represented as trees where each character is a child of the root. In fact, we have already made implicit use of edit distances on trees in the proof of Lemma 4.3.5, when we show how to convert one tree into another using leaf insert and delete operations, and subtree moves. In general, we will treat trees as having labels from some finite alphabet on each node. Alternatively, one might imagine that only leaf nodes are labelled and that internal nodes exist only to give structure. We distinguish tree editing paradigms by focusing on whether the trees are thought of as ordered (the children of each node occur in a specified sequence) or unordered (no organisation of the children). In many cases, documents will have tree structure: LaTeXdocuments are organised in chapters, sections and subsections, and this hierarchy can be mapped onto an ordered tree. Similarly, the organisation of files into directories under a computer operating system gives an unordered tree. XML documents can be treated as representing both ordered and unordered trees. In general, unordered tree problems will be more complex than their ordered equivalents: even testing whether two trees are equal, which is trivial in the ordered case, requires some ingenuity to design an algorithm to check in polynomial time for the unordered case.

We now briefly survey some of the different editing distances that have been studied for ordered and unordered trees.

Ordered Trees

The simplest editing distance between ordered trees is to take the standard string edit distance operations, and apply them to the tree. A change operation then corresponds to changing the label on a node. Insert and delete operations are a little more complicated: the simplest thing to do is to only allow inserts and deletes to leaf nodes (so a leaf node may be deleted or a new leaf node added as a child to an existing node). This metric is studied in [Cha99], and a dynamic programming solution given to find the distance. In fact, this is almost identical to the dynamic programming solution for string edit distance, with added restrictions to respect the tree structure. In [CRGMW96], a powerful subtree move operation is added to the repertoire. Here, any subtree may be detached from its parent and reattached to a new parent node at some specified point in its child list. An algorithm is given to find this editing distance. An earlier paper allows only inserts, deletes and changes of nodes, but permits inserts and deletes to affect internal nodes [ZS89]. Inserted nodes take a contiguous subsequence of their parent's children as their children, and the children of a deleted node become the children of its parent. A dynamic programming approach again finds the distance. A heuristic approach is taken in [BCD95], to allow a wide range of operations: inserts, deletes, splitting a node into two, merging two nodes and swapping two nodes.

[1]The cynic might choose to argue that these have not been covered because they do not begin with the letter 'S'

Unordered Trees

The basic editing distance on unordered trees, of deleting nodes, changing the label of nodes, or inserting nodes, has been shown to be NP-hard [ZSS92] and MAX SNP-Hard [ZJ94]. Here, deleting a node makes its children the children of its parent, and inserting a node allows a subset of the children of its parent to become its children (so every insertion can be reversed by a deletion and vice-versa). Hence heuristic approaches have been adopted to find editing distances without approximation guarantees [CGM97].

Tree Distance Embeddings

None of the approaches to finding tree distances mentioned above are amenable to an embedding approach. However, it is fairly straightforward to take the general approach of Section 4.2 to build approximate embeddings for trees of bounded degree (either ordered or unordered). The idea is to keep a histogram as before. This time, the histogram would record the number of copies of each node in the tree. Or, if copy and uncopy tree operations are allowed, then use bit flags to indicate whether a certain node is present or absent. It then follows that each permitted editing operation will affect only a limited number of entries in this histogram. The proof of Lemma 4.3.5 then says that one tree can be converted to another based on the size of the difference between their histograms. However the distortion factor of this embedding will be dh, where d is the bound on the degree, and h is the height of the tree. For balanced trees of n nodes, this will be $O(\log n)$, but in general this could be as large as $O(n)$, which would not be very satisfactory. A somewhat different approach would be needed to give tighter bounds on the approximation, and for trees whose degree is not bounded.

7.2.2 Graphs

The main problem in dealing with graphs is to define a meaningful distance measurement between pairs of graphs. We discuss some of the possibilities. Suppose that a graph with n vertices has each vertex labelled with a unique element from $\{1 \ldots n\}$. Then between two such graphs we can define a distance based on, say, inserting and deleting edges between vertex pairs. This distance is then exactly the Hamming distance between the adjacency matrix representation of the graphs, and can be dealt with using the methods developed already.

On the other hand, suppose graphs are unlabelled. Then determining whether a pair of graphs are actually the same is a non-trivial task: the problem of Graph Isomorphism, while widely conjectured to be solvable in polynomial time, has not been shown to be so. The problem of finding a matching of the nodes between the pair of graphs so that the editing distance between them is minimised is NP-Hard. This follows because the problem of Maximum Common Subgraph is NP-Hard [GJ79]. The smallest editing distance (based on inserting and deleting edges) is found by matching up the maximum common subgraphs, and then inserting and deleting edges as needed. A similar approach is motivated by Bunke and Shearer [BS98]. It is considered unlikely that there are efficient ways to solve this problem. Also, the existence of a polynomial time approximation algorithm for editing unlabelled graphs would imply the existence of a provably polynomial time test for isomorphism, which has so far eluded the algorithms community.

We might also think about block operations on graphs, but it is by no means clear what is reasonable to allow. For example, what effect would a block move of a subgraph have on the graph? In summary then, graph editing problems appear to be either trivial (the labelled case) or to pose significant challenges that have not yet been dealt with (the unlabelled case). A new problem definition should lead to 'good problems' that fall between these extremes. Alternatively, we might consider restricted kinds of graphs, such as trees (described above), or bipartite graphs [CMNR97].

Permutation Distance	Pairwise factor	Embedding Factor
Reversal	11/8	2
Transposition	3/2	2
Swap	1	1
Permutation edit	1	$\log n$

Figure 7.1: Approximation factors for permutation distances

7.3 Further Work

Throughout this work, there are some gaps in the stated results which it would be desirable to shore up either with algorithms to solve the problems, or else prove that these problems are computationally hard or insoluble in the way that we might desire. We now highlight the gaps, and outline the kind of results it may be feasible to prove.

Vectors and Sets

From the point of view of embeddings, it is reasonable to say that vector distances are virtually a closed problem: efficient solutions to embed vectors into smaller dimensional spaces are known for all L_p distances of interest ($0 < p \leq 2$), and hardness results preclude such embeddings for other measures such as dot product. Equally, problems for sets have either small upper bounds or large lower bounds (see the results of Figure 2.2). The problems we address, of approximate nearest and furthest neighbors, and clustering, have all received extensive study under Euclidean and Hamming distance, and satisfactory probabilistic solutions have been given. Future advances on such geometric problems would have the side benefit of implying efficient algorithms for the equivalent problems under the sequence distances we have studied, by making further use of our embeddings.

Permutations

The main open problem for the permutation distances is to produce embeddings that meet the best known approximation factors for pairwise comparisons. This has been achieved for the swap distance, and the gap is small for reversal and transposition based distances (see Figure 7.1 for a summary) but is large for Permutation Edit distance. This is the most interesting distance to deal with because of its close relationship to string edit distance and the length of the Longest Common Subsequence. The obvious open questions are whether the factor of $\log n$ can be improved or conversely whether the lower bound of $4/3$ on the factor can be raised and made more general. It is open whether any embedding of the Ulam metric can be made into a space which is sketchable, such as L_1 or L_2 space. Intuitively, it seems that improving on the $\log n$ distortion factor will be hard, but any embedding into a sketchable space would be of interest, even for a factor of order $\log n$ or more. Related to permutation edit distance, other permutation distances worthy of investigation include reversals and transpositions that are limited to moving blocks of at most some constant size [CS96, HV00].

Strings

From our point of view, the largest outstanding open problem for string distances is to find any embedding of the Levenshtein edit distance into some vector space — preferably a space that is sketchable. A more general but closely related question is to find some way to approximate the edit distance between two strings which is faster than the quadratic $O(mn)$ dynamic programming exact solution. The only bounds we currently have are one sided: the edit distance with moves is a lower bound on the edit distance, and we have various approximation results for this distance. However, this does not lead to a bounded approximation for edit distance. Other open problems for strings are to improve the factors of approximation from $O(\log n \log^* n)$, if possible to constants. It has already been remarked that equivalent problems on permutations are embeddable into vector spaces with only small constant distortions.

The problem of approximate pattern matching under string edit distances has received a lot of attention from the string matching community. For Hamming distance, many efficient solutions have been described. For Edit distance, several solutions have been given for the k-differences version of the problem (where only occurrences with an edit distance of less than some parameter k are reported). We have described the first efficient solutions for edit distance with moves. However, these have entailed the $O(\log n \log^* n)$ approximation factors inherent from the embedding approach. It is open to improve the approximation factors for these distances, and to provide better solutions for the edit distance case.

Other Variations

Throughout this work, we have concerned ourselves with unit cost operations: every editing operation is considered to have cost 1. Throughout biology and other matching situations, it is common to assign varying weights to different operations, or to give a different cost depending on the symbols involved (so transforming an `A` into a `T` may have a different cost compared to transforming into a `C`). To some extent, these can be absorbed into the current approaches, by claiming a reduced approximation factor, but in general a new approach needs to be taken to deal with weighted operation costs.

Appendix A

Supplemental Section on Sequence Similarity

"Eight weeks passed away like this, and I had written about Abbots and Archery and Armour and Architecture and Attica, and hoped with diligence that I might get on to the B's before very long."

[CD92]

A.1 Combined Permutation Distances

The following results describe embeddings for permutation metrics made by allowing combinations of the operations studied individually in Chapter 3: transpositions, reversals, edits, and symbol insertion and deletions.

A.1.1 Combining All Operations

We consider the compound distance allowing the combination of reversals, transpositions and permutation editing (moving a single symbol). Denote this distance as $\tau(P, Q)$. We make use of the transformation R from Section 3.2.2.

Lemma A.1.1 $\tau(P, Q) \leq \frac{1}{2}||R(P) - R(Q)||_H \leq 3\tau(P, Q)$.

Proof. Imagine transforming P into Q. Our suite of operations are Transpositions, Reversals and symbol Moves — however, a move can be described as a special case of a transposition, hence we need only consider two types of operation. We shall use again the notion of reversal breakpoints, recalling that they are defined as pairs of elements which are adjacent in one sequence but not the other. It is clear that a transposition could remove at most 3 reversal breakpoints, since a reversal breakpoint is a special case of a transposition breakpoint. And we know that a reversal can remove no more than 2 reversal breakpoints. So any operation can remove no more than 3 breakpoints, which gives a lower bound on the number of operations necessary. We have already seen how reversals alone can be used to transform a sequence using at most the number of reversal breakpoints in the proof of Theorem 3.2.2. Therefore, counting reversal breakpoints gives a 3-approximation to the distance. The number of breakpoints can be found using the $R(\cdot)$ matrices of Section 3.2.2. □

Example.

$$\begin{array}{cccccccccc} P & 0 & 2 & 4 & 3 & 1 & 7 & 6 & 5 & 8 \\ Q & 0 & 2 & 7 & 6 & 4 & 3 & 1 & 5 & 8 \end{array}$$

The number of reversal breakpoints between P and Q is 3. Note that a single transposition could transform P into Q. If we just use reversals, the transformation can be done in 3 operations:

$$0\ 2\ \underline{4\ 3\ 1\ 7\ 6}\ 5\ 8 \longrightarrow 0\ 2\ \underline{6\ 7}\ 1\ 3\ 4\ 5\ 8 \longrightarrow 0\ 2\ 7\ 6\ \underline{1\ 3\ 4}\ 5\ 8 \longrightarrow 0\ 2\ 7\ 6\ 4\ 3\ 1\ 5\ 8$$

Using just reversals gives a three approximation to the mixed distance.

A.1.2 Transpositions, Insertions, Reversals, Edits and Deletions

We describe a distance, $\tau'(P, Q)$, which is defined over sequences P, Q both drawn from a fixed universe of ℓ symbols, labelled 1 to ℓ. Not only does this allow transpositions, reversals and moves, but additionally symbol insertions and deletions, all charged at unit cost. These allow us to compare arbitrary permutations with non-identical sets of elements. Similarly, define $r'(P, Q)$ as the reversal distance between P and Q where inserts and deletes are allowed. Recall that we extend our sequences so that the first element is 0 and the last element is $\ell + 1$. Define $R'(P)$ as a binary matrix of size $(\ell + 2) \times (\ell + 2)$.

$$\begin{array}{rcl} (i \in P) \wedge (j \in P) \wedge (i > j) \wedge |P^{-1}[i] - P^{-1}[j]| = 1 & \Longrightarrow & R'(P)[i, j] = 1 \\ (i \in P) & \Longrightarrow & R'(P)[i, i] = 1 \\ \text{otherwise} & \Longrightarrow & R'(P)[i, j] = 0 \end{array}$$

Lemma A.1.2

$$\tau'(P, Q) \leq ||R'(P) - R'(Q)||_H \leq 3\tau'(P, Q)$$

$$r'(P, Q) \leq ||R'(P) - R'(Q)||_H \leq 2r'(P, Q)$$

Proof. We shall prove both of these claims together, since the important part is that each insert or delete has a limited effect on the number of breakpoints. Consider a canonical form of an optimal transformation in which all the deletions occur first, and all the insertions occur at the end. All characters in P but not in Q must be deleted, and all characters in Q but not in P must be inserted. Between these insertions and deletions, the start and end sequences are permutations of each other — call these P' and Q' — and can be transformed using the same techniques we have already seen. For τ', we use the results of Lemma A.1.1. The number of reversal breakpoints of these sequences gives a 3-approximation to the distance; we need to extend the concept of a reversal breakpoint to cope with the different alphabets used. For r' we use the results of Theorem 3.2.2, and we have a 2-approximation.

When calculating the number of reversal breakpoints, we shall count an additional breakpoint for every member of Q not in P: this modified count is $\phi'(P, Q)$. When we delete an element, this will always remove a reversal breakpoint (else, the operation is unnecessary and can be avoided), and can remove at most two breakpoints. When we insert an element i, this is paid for by the fact that i is in Q but not in P (this we count as a breakpoint). We may also remove an additional breakpoint if the symbol is inserted between both its neighbours in Q. Together, this means that every insertion and deletion removes at least one breakpoint and at most 2. We know that P' can be transformed into Q' using at most $\phi'(P', Q') = \phi(P', Q')$ operations. Therefore, counting the modified version of reversal breakpoints gives a 3-approximation to the combined distance and a 2-approximation to the reversal distance, with indels.

We now show that $\phi'(P, Q) + \phi'(Q, P) = ||R'(P) - R'(Q)||_H$. Clearly, since R' is identical to R except on the leading diagonal, this captures the pairwise reversal breakpoints. We use the main diagonal of this $R'(P)$ matrix to indicate the presence of element i with a 1 in entry $R'(P)[i, i]$, and only consider pairs i, j when $i \geq j$. So the Hamming distance of the main diagonals additionally counts all the elements of P not in Q and vice-versa, as required. Since $\tau'(P, Q) \leq \phi'(P, Q) \leq 3\tau'(P, Q)$ and $\tau'(Q, P) \leq \phi'(Q, P) \leq 3\tau'(Q, P)$ then $\tau'(P, Q) \leq \frac{1}{2}||R'(P) - R'(Q)||_H \leq 3\tau'(P, Q)$.

Similarly, for r', we know (from Theorem 3.2.2) that $r'(P', Q') \leq \phi(P', Q') \leq 2r'(P', Q')$ — that is, that we can find a 2-approximation to the reversal distance from the breakpoints of the permutations P' and Q'. By the same argument as above, we know that we can reach P' from P and Q from Q' using a number of insertions and deletions that is at least, and at most twice, the number of breakpoints generated by symbols in P but not in Q and vice-versa. Therefore $r'(P, Q) \leq \phi'(P, Q) \leq 2r'(P, Q)$ and $r'(Q, P) \leq \phi'(Q, P) \leq 2r'(Q, P)$. As noted above, $\phi'(P, Q) + \phi'(Q, P) = ||R'(P) - R'(Q)||_H$. Hence $r'(P, Q) \leq \frac{1}{2}||R'(P) - R'(Q)||_H \leq 2r'(P, Q)$. □

Example. Suppose $P = 0\ 7\ 6\ 5\ 3\ 4\ 2\ 8$ and $Q = 0\ 4\ 3\ 1\ 5\ 6\ 8$. Our transform matrices are as follows:

$R'(P)$	0	1	2	3	4	5	6	7	8
0	1	0	0	0	0	0	0	1	0
1	0	0	0	0	0	0	0	0	0
2	0	0	1	0	1	0	0	0	1
3	0	0	0	1	1	1	0	0	0
4	0	0	0	0	1	0	0	0	0
5	0	0	0	0	0	1	1	0	0
6	0	0	0	0	0	0	1	1	0
7	0	0	0	0	0	0	0	1	0
8	0	0	0	0	0	0	0	0	1

$R'(Q)$	0	1	2	3	4	5	6	7	8
0	1	0	0	0	1	0	0	0	0
1	0	1	0	1	0	1	0	0	0
2	0	0	0	0	0	0	0	0	0
3	0	0	0	1	1	0	0	0	0
4	0	0	0	0	1	0	0	0	0
5	0	0	0	0	0	1	1	0	0
6	0	0	0	0	0	0	1	0	1
7	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	1

Here $||R'(P) - R'(Q)||_H = 12 = \phi'(P, Q) + \phi'(Q, P) = 6 + 6$

To transform P into Q, we first remove elements of P not in Q, then apply reversals, then insert elements of Q not in P:

$$0\ \underline{7}\ 6\ 5\ 3\ 4\ 2\ 8 \xrightarrow{delete} 0\ 6\ 5\ 3\ 4\ \underline{2}\ 8 \xrightarrow{delete} 0\ \underline{6\ 5\ 3\ 4}\ 8 \xrightarrow{reverse} 0\ 4\ 3\ \_\ 5\ 6\ 8 \xrightarrow{insert} 0\ 4\ 3\ 1\ 5\ 6\ 8$$

Hence $\tau'(P,Q) = r'(P,Q) = 4$, and $r'(P,Q) \leq \frac{1}{2}||R'(P) - R'(Q)||_H \leq 2r'(P,Q)$.

Allowing one sequence to be a string

We relax the assumption that both sequences are permutations, and allow at most one of P, Q to be an arbitrary string of characters from the alphabet $\sigma = \{1, 2, \ldots \ell\}$. Denote these distances $r''(P,Q)$ for reversals and indels, and $t''(P,Q)$ for transpositions and indels. The following transformation is essentially the same as R' and T', but counts the number of adjacent pairs and occurences of each character, and we find the L_1 distance between the transforms. Here $*$ is used to denote an extra row of the matrix used to keep frequency counts.

$$\begin{aligned}
R''(P)[*,i] &= |\{k \mid P[k] = i\}| \\
R''(P)[i,j] &= |\{k \mid (i \geq j) \wedge (P[k] = i \wedge P[k+1] = j) \vee (P[k] = j \wedge P[k+1] = i)\}| \\
\\
T''(P)[*,i] &= |\{k \mid P[k] = i\}| \\
T''(P)[i,j] &= |\{k \mid (P[k] = i \wedge P[k+1] = j)\}|
\end{aligned}$$

If Q is the permutation, note that $R''(Q)$ and $T''(Q)$ will both be binary matrices under this modification.

Lemma A.1.3 *$r''(P,Q) \leq \frac{1}{2}||R''(P) - R''(Q)||_1 \leq \frac{2}{r}''(P,Q)$
and $t''(P,Q) \leq \frac{1}{2}||T''(P) - T''(Q)||_1 \leq \frac{2}{t}''(P,Q)$*

Proof. We shall give the proof for reversals in detail; the proof for transpositions is mostly identical. We define a modified notion of reversal breakpoints, denoted $\phi''(P,Q)$ which counts the number of pairs (i,j) that are adjacent in P less the number of times this pair occurs in Q, and additionally counts half the difference in the number of times a symbol i occurs in P against the number of times in Q. We observe that this is defined so that $\phi''(P,Q) + \phi''(Q,P) = ||R''(P) - R''(Q)||_1$. We will use ϕ'' to approximate the distance between P and Q.

At the core of the transformation we want to use reversals only to transform P into Q. We consider an optimal transformation, and observe some facts about it. Firstly, we can ensure that we only delete items i that occur more times in P than in Q: suppose this were not the case. Then, we must delete an item and then subsequently insert it. The net effect is to move the item to a new location. But this movement of a single symbol can be simulated using at most two reversals (one to move i, the second to restore the sequence). So there is an equivalent transformation that uses reversals only. A similar argument shows that we never need to insert an item and subsequently delete it, so we assume that we minimize the number of insertions and deletions. We can now reorder the operations so that all deletions occur first, then all reversals, then all insertions. This does not change the number of operations needed. Let the sequence at the start of the reversals section be P', and the sequence at the end of the reversals be Q'. P' and Q' are permutations of each other, and each symbol occurs at most once in P'. Let $d(P,P')$ denote the number of deletions used to transform P into P', and let $d(Q,Q')$ be the number of deletions used to transform Q into Q' (this is also the number of insertions used to transform Q' into Q). By the above observation, $d(P,P') + d(Q,Q') = ||R''(P)[*] - R''(Q)[*]||_1$.

We will consider the effect of performing the $d(P,P')$ deletes upon ϕ''. For any delete, we know that the number of breakpoints is reduced by at least one half: since we are removing a symbol i that occurs k times in P and at most once in Q, we have credit from $R'(P)[*,i]$ to do this. On the other hand, a deletion could remove at most two and a half breakpoints — this occurs when we have the sequence $a\ b\ c$ in P and $a\ c$ in Q: the breakpoints resulting from the pairs (a, b) and (b, c) are removed, as well as the half breakpoint from the removal of the extra b. A similar argument applies for insertions: every insertion removes at least half a breakpoint, symetrically with deletions, and can remove at most one and half breakpoints when the correct symbol is inserted between its two neighbors in Q. As usual, reversals can remove at most two breakpoints, since they do not insert or delete symbols and affect only two adjacaencies.

Next, we consider the central series of reversals to turn P' into Q'. Observe that $\phi''(P, Q) \leq \frac{5}{2}d(P, P') + \frac{3}{2}d(Q, Q') + 2r(P', Q')$, since this is the greatest effect each operation can have on the number of breakpoints. On the other hand, the number of (regular) breakpoints between P' and Q', $\phi(P', Q')$, is at most $\phi''(P, Q) - \frac{1}{2}d(P, P') - \frac{1}{2}d(Q, Q')$, since we know each deletion and insertion necessary to reach P' and Q' must remove at least one breakpoint. We know that we can transform P' into Q' using at most $\phi''(P, Q) - \frac{1}{2}d(Q, Q') - \frac{1}{2}d(P, P')$ reversals: since P' and Q' are permutations of each other then the breakpoints that remain correspond to the "regular" breakpoints and we can make the transformation using this many reversals (by Theorem 3.2.2). So $r(P', Q') \geq \phi''(P, Q) - d(Q, Q') - d(P, P')$. Now, $r''(P, Q) = d(P, P') + r(P', Q') + d(Q, Q')$, so

$$\begin{aligned}
2r''(P, Q) &\leq 2(r(P', Q') + d(P, P') + d(Q, Q')) \\
&\leq \phi''(P, Q) + \phi''(Q, P) \\
&\leq 3d(P, P') + 2d(Q, Q') + 2r(P', Q') + 3d(Q, Q') + 2d(P, P') + 2r(P', Q') \\
&\leq 5(d(P, P') + i(Q, Q') + r(P', Q')) \\
&= 5r''(P, Q)
\end{aligned}$$

The case for transpositions is identical, using breakpoints based on ordered pairs, because insertions and deletions have the same effect on these breakpoints. □

Bibliography

[Abi74] Walter Abish. *Alphabetical Africa.* New Directions, 1974.

[Abr87] K. Abrahamson. Generalized string matching. *SIAM Journal on Computing*, 16(6):1039–1051, 1987.

[ABR00] S. Alstrup, G. S. Brodal, and T. Rauhe. Pattern matching in dynamic texts. In *Proceedings of the 11th Annual Symposium on Discrete Algorithms*, pages 819–828, 2000.

[Ach01] D. Achlioptas. Database-friendly random projections. In *Proceedings of Principles of Database Systems*, pages 274–281, 2001.

[Ada79] D. N. Adams. *The Hitch-hiker's Guide to the Galaxy.* Pan, 1979.

[AFS93] R. Agrawal, C. Faloutsos, and A. N. Swami. Efficient similarity search in sequence databases. In *Proceedings of the 4th International Conference of Foundations of Data Organization and Algorithms*, volume 730, pages 69–84. Lecture Notes in Computer Science, Springer, 1993.

[AGE94] K. A. S. Abdel-Ghaffar and A. El Abbadi. An optimal strategy for comparing file copies. *IEEE Transactions on Parallel and Distributed Systems*, 5(1):87–93, 1994.

[AHK01] C. Aggarwal, A. Hinneburg, and D. Keim. On the surprising behavior of distance metrics in high dimensional space. In *Proceedings of the 8th International Conference on Database Theory*, pages 420–434, 2001.

[ALP00] A. Amir, M. Lewenstein, and E. Porat. Faster algorithms for string matching with k-mismatches. In *Proceedings of the 11th Annual Symposium on Discrete Algorithms*, pages 794–803, 2000.

[AM91] R. J. Anderson and G. L. Miller. Deterministic parallel list ranking. *Algorithmica*, 6:859–868, 1991.

[AMS96] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing*, pages 20–29, 1996.

[AMS99] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. *JCSS: Journal of Computer and System Sciences*, 58:137–147, 1999.

[Bal87] J. Ball. *Johnny Ball's Second Thinks.* Puffin Books, 1987.

[BCC$^+$00] A. L. Buchsbaum, D. F. Caldwell, K. W. Church, G. S. Fowler, and S. Muthukrishnan. Engineering the compression of massive tables: an experimental approach. In *Proceedings of the 11th Annual Symposium on Discrete Algorithms*, pages 175–184, 2000.

[BCD95] D. T. Barnard, G. Clarke, and N. Duncan. Tree-to-tree correction for document trees. Technical Report 95-372, Department of Computing and Information Science, Queen's University, Ontario, Canada, 1995.

[BCFM98] A. Broder, M. Charikar, A. Frieze, and M. Mitzenmacher. Min-wise independent permutations. In *Proceedings of the 30th Symposium on Theory of Computing*, pages 327–336. ACM Press, 1998.

[BCH99] D. Belanger, K. Church, and A. Hume. Virtual data warehousing, data publishing, and call details. In *Proceedings of Databases in Telecommunications*, volume 1819, pages 106–117. Lecture Notes in Computer Science, Springer, 1999.

[BCL02] D. Benedetto, E. Caglioti, and V. Loreto. Language trees and zipping. *Physical Review Letters*, 88(048702), 2002.

[BGR01] S. Babu, M. Garofalakis, and R. Rastogi. SPARTAN: a model-based semantic compression system for massive data tables. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 283–294, 2001.

[BHK01] P. Berman, S. Hannenhalli, and M. Karpinski. 1.375-approximation algorithm for sorting by reversals. Technical Report 2001-41, ECCCTR: Electronic Colloquium on Computational Complexity, 2001.

[BK98] P. Berman and M. Karpinski. On some tighter inapproximability results. Technical Report 85193-CS, Department of Computer Science, University of Bonn, 1998.

[BL91] D. Barbara and R. J. Lipton. A class of randomized strategies for low-cost comparison of file copies. *IEEE Transactions on Parallel and Distributed Systems*, 2(2):160–170, 1991.

[BMY01] D. A. Bader, B. M. E. Moret, and M. Yan. A linear-time algorithm for computing inversion distances between signed permutations with an experimental study. In *Proceedings of the 7th Workshop on Algorithms and Data Structures*, pages 365–376. Lecture Notes in Computer Science, Springer, 2001.

[Bou88] J. Bourgain. On Lipschitz embedding of finite metric spaces in Hilbert space. *Israel Journal Mathematics*, 52:46–52, 1988.

[BP93] V. Bafna and P. A. Pevzner. Genome rearrangements and sorting by reversals. In *Proceedings of the 34th Annual Symposium on Foundations of Computer Science*, pages 148–157, 1993.

[BP98] V. Bafna and P. A. Pevzner. Sorting by transpositions. *SIAM Journal on Discrete Mathematics*, 11(2):224–240, 1998.

[Bro98] A. Broder. On the resemblance and containment of documents. In *Proceedings of Compression and Complexity of Sequences (SEQUENCES'97)*, pages 21–29, 1998.

[BS98] H. Bunke and K. Shearer. A graph distance metric based on the maximal common subgraph. *Pattern Recognition Letters*, 19(2–3):255–259, 1998.

[BSW01] S. Babu, L. Subramanian, and J. Widom. A data stream management system for network traffic management. In *Proceedings of Workshop on Network-Related Data Management*, 2001.

[BYJK$^+$02] Z. Bar-Yossef, T.S. Jayram, R. Kumar, D. Sivakumar, and L. Trevisian. Counting distinct elements in a data stream. In *Proceedings of RANDOM 2002*, pages 1–10, 2002.

[Cap97] A. Caprara. Sorting by reversals is difficult. In *Proceedings of the First International Conference on Computational Molecular Biology*, pages 75–83, 1997.

[CCMN00] M. Charikar, S. Chaudhuri, R. Motwani, and V. R. Narasayya. Towards estimation error guarantees for distinct values. In *Proceedings of the Nineteenth Symposium on Principles of Database Systems*, pages 268–279, 2000.

[CD92] A. Conan-Doyle. The Redheaded League. *The Adventures of Sherlock Holmes*, 1892.

[CDIM02] G. Cormode, M. Datar, P. Indyk, and S. Muthukrishnan. Comparing data streams using Hamming norms. In *Proceedings of 28th International Conference on Very Large Data Bases*, pages 335–345, 2002.

[CGJS02] C. Cranor, L. Gao, T. Johnson, and O. Spatscheck. Gigascope: High performance network monitoring with an SQL interface. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, page 262, 2002.

[CGM97] S. S. Chawathe and H. Garcia-Molina. Meaningful change detection in structured data. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, volume 26(2), pages 26–37, 1997.

[CH98] R. Cole and R. Hariharan. Approximate string matching: A simpler, faster algorithm. In *Proceedings of the 9th Annual Symposium on Discrete Algorithms*, pages 463–472, 1998.

[Cha99] S. S. Chawathe. Comparing hierarchical data in external memory. In *Proceedings of the 25th International Conference on Very Large Databases*, pages 90–101, 1999.

[Chr98a] D. A. Christie. A 3/2-approximation algorithm for sorting by reversals. In *Proceedings of the 9th Annual Symposium on Discrete Algorithms*, pages 244–252, 1998.

[Chr98b] D. A. Christie. *Genome Rearrangement Problems*. PhD thesis, University of Glasgow, UK, 1998.

[CIKM02] G. Cormode, P. Indyk, N. Koudas, and S. Muthukrishnan. Fast mining of tabular data via approximate distance computations. In *Proceedings of the International Conference on Data Engineering*, pages 605–616, 2002.

[CL99] E. Cohen and D. D. Lewis. Approximating matrix multiplication for pattern recognition tasks. *Journal of Algorithms*, 30(2):211–252, 1999.

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[CM02] G. Cormode and S. Muthukrishnan. The string edit distance matching problem with moves. In *Proceedings of the 13th Annual Symposium on Discrete Algorithms*, pages 667–676, 2002.

[CMNR97] K. Cirino, S. Muthukrishnan, N. S. Narayanaswamy, and H. Ramesh. Graph editing to bipartite interval graphs: Exact and asymptotic bounds. *Lecture Notes in Computer Science*, 1346:37–53, 1997.

[CMȘ01] G. Cormode, S Muthukrishnan, and S. C. Șahinalp. Permutation editing and matching via embeddings. In *Proceedings of 28th International Colloquium on Automata, Languages and Programming*, volume 2076, pages 481–492, 2001.

[Cor02] G. Cormode. Crossword set by mugwump. *The Warwick Boar*, February 2002.

[CPṢV00] G. Cormode, M. Paterson, S. C. Ṣahinalp, and U. Vishkin. Communication complexity of document exchange. In *Proceedings of the 11th Symposium on Discrete Algorithms*, pages 197–206, 2000.

[CR94] M. Crochemore and W. Rytter. *Text Algorithms.* Oxford University Press, 1994.

[CRGMW96] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 493–504, 1996.

[CS96] T. Chen and S. Skiena. Sorting with fixed-length reversals. *DAMATH: Discrete Applied Mathematics and Combinatorial Operations Research and Computer Science*, 71:269–296, 1996.

[CV86] R. Cole and U. Vishkin. Deterministic coin tossing and accelerating cascades: micro and macro techniques for designing parallel algorithms. In *Proceedings of the 18th Symposium on Theory of Computing*, pages 206–219, 1986.

[Day] Daytona. AT&T research, Daytona Database Management System. Details at `http://www.research.att.com/projects/daytona/`.

[DG99] S. Dasgupta and A. Gupta. An elementary proof of the Johnson-Lindenstrauss lemma. Technical Report TR-99-006, International Computer Science Institute, Berkeley, 1999.

[DH98] M. Deza and T. Huang. Metrics on permutations, a survey. *Journal of Combinatorics, Information and System Sciences*, 23:173–185, 1998.

[DJK$^+$98] B. DasGupta, T. Jiang, S. Kannan, M. Li, and E. Sweedyk. The complexity and approximation of syntenic distance. *Discrete Applied Mathematics*, 88:59–82, 1998.

[DJMS02] P. Dasu, T. Johnson, S. Muthukrishnan, and V. Shkapenyuk. Mining database structures or how to build a data quality browser. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 240–251, 2002.

[EEK$^+$01] H. Eriksson, K. Eriksson, J. Karlander, L. Svensson, and J. Wästlund. Sorting a bridge hand. *Discrete Mathematics*, 241:289–301, 2001.

[EG81] S. Even and O. Goldreich. The minimum length generator sequence is NP-Hard. *Journal of Algorithms*, 2:311–313, 1981.

[EKSX96] M. Ester, H-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, page 226, 1996.

[Evf00] A. V. Evfimievski. A probabilistic algorithm for updating files over a communication link. *Theoretical Computer Science*, 233(1–2):191–199, 2000.

[Fal96] C. Faloutsos. *Indexing Multimedia Databases*. Kluwer, 1996.

[FGL$^+$00] A. Feldmann, A. Greenberg, C. Lund, N. Reingold, and J. Rexford. Netscope: Traffic engineering for IP networks. *IEEE Network Magazine*, pages 11–19, 2000.

[Fiv70] The Jackson Five. *ABC*. Motown Records, 1970.

[FKSV99] J. Feigenbaum, S. Kannan, M. Strauss, and M. Viswanathan. An approximate L_1-difference algorithm for massive data streams. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, pages 501–511, 1999.

[FM83] P. Flajolet and G. N. Martin. Probabilistic counting. In *24th Annual Symposium on Foundations of Computer Science*, pages 76–82, 1983.

[FM85] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for database applications. *Journal of Computer and System Sciences*, 31:182–209, 1985.

[FNS96] V. Ferretti, J. H. Nadeau, and D. Sankoff. Original synteny. In *Proceedings of the 7th Annual Symposium Combinatorial Pattern Matching*, volume 1075 of *Lecture Notes in Computer Science*, pages 159–167. Springer, 1996.

[FP74] M. Fischer and M. Paterson. String-matching and other products. In *Proceedings of SIAM-AMS: Complexity of Computation*, pages 113–125, 1974.

[FS00] J. Fong and M. Strauss. An approximate L^p-difference algorithm for massive data streams. In *Proceedings of the Annual Symposium on Theoretical Aspects of Computer Science*, pages 193–204, 2000.

[FY00] C. Faloutsos and B. Yi. Fast time sequence indexing for arbitrary L_p norms. In *Proceedings of the 26th International Conference on Very Large Databases*, pages 385–394, 2000.

[GG98] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[GGP$^+$99] L. A. Goldberg, P. W. Goldberg, M. Paterson, P. Pevzner, S. C. Ṣahinalp, and E. Sweedyk. The complexity of gene placement. In *Proceedings of the 10th Annual Symposium on Discrete Algorithms*, pages 386–395, 1999.

[GGR00] V. Ganti, J. Gehrke, and R. Ramakrishnan. DEMON: Mining and monitoring evolving data. In *Proceedings of 16th International Conference on Data Engineering*, pages 439–448, 2000.

[Gib85] A. Gibbons. *Algorithmic Graph Theory*. Cambridge University Press, 1985.

[Gib01] P. Gibbons. Distinct sampling for highly-accurate answers to distinct values queries and event reports. In *27th International Conference on Very Large Databases*, pages 541–550, 2001.

[GIM99] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *Proceedings of the 25th International Conference on Very Large Databases*, pages 518–529, 1999.

[GIV01] A. Goel, P. Indyk, and K. Varadarajan. Reductions among high dimensional proximity problems. In *Proceedings of the 12th Annual Symposium on Discrete Algorithms*, pages 769–778, 2001.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability, a Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[GKMS01a] A. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. QuickSAND: Quick summary and analysis of network data. Technical Report 2001-43, DIMACS, 2001.

[GKMS01b] A. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. Surfing wavelets on streams: One-pass summaries for approximate aggregate queries. In *Proceedings of 27th International Conference on Very Large Data Bases*, pages 79–88, 2001.

[GKPV01] M. Grossglauser, N. Koudas, Y. Park, and A. Varriot. Falcon: Fault management via alarm warehousing and mining. In *Proceedings of Workshop on Network-Related Data Management*, 2001.

[GM99] P. Gibbons and Y. Matias. Synopsis structures for massive data sets. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, A, 1999.

[Gon85] T. F. Gonzalez. Clustering to minimize the maximum intercluster distance. *Theoretical Computer Science*, 38(2-3):293–306, 1985.

[GP79] W.H. Gates and C. H. Papadimitriou. Bounds for sorting by prefix reversals. *Discrete Mathematics*, 27:47–57, 1979.

[GPS87] A. Goldberg, S. Plotkin, and G. Shannon. Parallel symmetry-breaking in sparse graphs. In *Proceedings of the 19th Symposium on Theory of Computing*, pages 315–324, 1987.

[GPS99] Q. Gu, S. Peng, and H. Sudborough. A 2-approximation algorithm for genome rearrangements by reversals and transpositions. *Theoretical Computer Science*, 210(2):327–339, 1999.

[GRS98] S. Guha, R. Rastogi, and K. Shim. CURE: An efficient clustering algorithm for large databases. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 73–84, 1998.

[GT01] P. Gibbons and S. Tirthapura. Estimating simple functions on the union of data streams. In *Proceedings of the 13th ACM Symposium on Parallel Algorithms and Architectures*, pages 281–290, 2001.

[Gus97] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology.* Cambridge University Press, 1997.

[Ham80] R. W. Hamming. *Coding and Information Theory*. Prentice-Hall, 1980.

[HNSS95] P. J. Haas, J. F. Naughton, S. Seshadri, and L. Stokes. Sampling-based estimation of the number of distinct values of an attribute. In *Proceedings of the 21st International Conference on Very Large Databases*, pages 311–322, 1995.

[HP95] S. Hannenhalli and P. A. Pevzner. Transforming men into mice (polynomial algorithm for genomic distance problem). In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, pages 581–592, 1995.

[HRR98] M. Henzinger, P. Raghavan, and S. Rajagopalan. Computing on data streams. Technical Report SRC 1998-011, DEC Systems Research Centre, 1998.

[HS77] J. W. Hunt and T. G. Szymanski. A fast algorithm for computing longest common subsequences. *Communications of the ACM*, 20(5):350–353, 1977.

[HV00] L. S. Heath and J. P. C. Vergara. Sorting by short block-moves. *Algorithmica*, 28(3):323–354, 2000.

[IKM00] P. Indyk, N. Koudas, and S. Muthukrishnan. Identifying representative trends in massive time series data sets using sketches. In *Proceedings of the 26th International Conference on Very Large Databases*, pages 363–372, 2000.

[IM98] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the 30th Symposium on Theory of Computing*, pages 604–613, 1998.

[Ind98] P. Indyk. On approximate nearest neighbors in non-Euclidean spaces. In *Proceedings of the 39th Annual Symposium on Foundations of Computer Science*, pages 148–155, 1998.

[Ind00] P. Indyk. Stable distributions, pseudorandom generators, embeddings and data stream computation. In *Proceedings of the 40th Symposium on Foundations of Computer Science*, pages 189–197, 2000.

[Ind01] P. Indyk. Algorithmic aspects of geometric embeddings (invited tutorial). In *Proceedings of 42nd Annual Symposium on Foundations of Computer Science*, pages 10–35, 2001.

[Jac73] T. Jacks. *Seasons in the Sun, translated from 'My Death' by Jacques Brel*. Goldfish, 1973.

[JD88] A. K. Jain and R. C. Dubes. *Algorithms for Clustering Data*. Prentice-Hall, 1988.

[Jer85] M. R. Jerrum. The complexity of finding minimum-length generator sequences. *Theoretical Computer Science*, 36(2-3):265–289, 1985.

[JL84] W.B. Johnson and J. Lindenstrauss. Extensions of Lipshitz mapping into Hilbert space. *Contemporary Mathematics*, 26:189–206, 1984.

[Kar93] H. Karloff. Fast algorithms for approximately counting mismatches. *Information Processing Letters*, 48(2):53–60, 1993.

[KMR72] R. M. Karp, R. E. Miller, and A. L. Rosenberg. Rapid identification of repeated patterns in strings, trees and arrays. In *Proceedings of the 4th Symposium on Theory of Computing*, pages 125–136, 1972.

[KN97] E. Kushilevitz and N. Nisan. *Communication Complexity*. Cambridge University Press, 1997.

[Knu98] D. E. Knuth. *The Art of Computer Programming, Vol 3, Sorting and Searching*. Addison-Wesley, 2nd edition, 1998.

[KOR98] E. Kushilevitz, R. Ostrovsky, and Y. Rabani. Efficient search for approximate nearest neighbor in high dimensional spaces. In *Proceedings of the 30th Symposium on Theory of Computing*, pages 614–623, 1998.

[KR87] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[KR95] J. D. Kececioglu and R. Ravi. Of mice and men: Algorithms for evolutionary distances between genomes with translocation. In *Proceedings of the 6th Annual Symposium on Discrete Algorithms*, pages 604–613. ACM Press, 1995.

[KS92] B. Kalyanasundaram and G. Schnitger. The probabilistic communication complexity of set intersection. *SIAM Journal on Discrete Mathematics*, 5(4):545–557, 1992.

[KS95] J. Kececioglu and D. Sankoff. Exact and approximation algorithms for sorting by reversals, with application to genome rearrangement. *Algorithmica*, 13(1/2):180–210, 1995.

[KSS99] H. Kaplan, M. Strauss, and M. Szegedy. Just the fax — differentiating voice and fax phone lines using call billing data. In *Proceedings of the 10th Annual Symposium on Discrete Algorithms*, pages 935–936, 1999.

[LCL$^+$03] M. Li, X. Chen, X. Li, B. Ma, and P. Vitanyi. The similarity metric. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 863–872, 2003.

[Lev66] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady.*, 10(8):707–710, 1966.

[LT97] D. Lopresti and A. Tomkins. Block edit models for approximate string matching. *Theoretical Computer Science*, 181(1):159–179, 1997.

[LV86] G. M. Landau and U. Vishkin. Introducing efficient parallelism into approximate string matching and a new serial algorithm. In *Proceedings of the 18th Symposium on Theory of Computing*, pages 220–230, 1986.

[Mad89] T. Madej. An application of group testing to the file comparison problem. In *Proceedings of the 9th International Conference on Distributed Computing Systems*, pages 237–245. IEEE, 1989.

[Mat] J. Matouǔsek. Embedding finite metric spaces into Euclidean spaces. Available from `http://www.ms.mff.cuni.cz/acad/kam/matousek/dg.html`.

[Mat02] J. Matoušek. Workshop in discrete metric spaces and their algorithmic applications — open problems. Available from `http://kam.mff.cuni.cz/~matousek/haifaop.ps`, 2002.

[McC76] E. M. McCreight. A space-economic suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

[Met83] J. J. Metzner. A parity structure for large remotely located replicated data files. *IEEE Transactions on Computers*, 32(8):727–730, 1983.

[Met91] J. J. Metzner. Efficient replicated remote file comparison. *IEEE Transactions on Computers*, 40(5):651–659, 1991.

[MF02] S. Madden and M. J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *Proceedings of 18th International Conference on Data Engineering*, pages 555–566, 2002.

[MP80] W. J. Masek and M. S. Paterson. A faster algorithm computing string edit distances. *Journal of Computer and System Sciences*, 20:18–31, 1980.

[MR95] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[MS77] F. J. MacWilliams and N. J. A. Sloane. *The Theory of Error-Correcting Codes*, volume 16 of *North-Holland Mathematical Library*. North-Holland, 1977.

[MṢ99] Y. Matias and S. C. Ṣahinalp. On the optimality of parsing in dynamic dictionary based data compression. In *Proceedings of the 10th Annual Symposium on Discrete Algorithms*, pages 943–944, 1999.

[MṢ00] S. Muthukrishnan and S. C. Ṣahinalp. Approximate nearest neighbors and sequence comparison with block operations. In *Proceedings of the 32nd Symposium on Theory of Computing*, pages 416–424, 2000.

[MṢ02] S. Muthukrishnan and S. C. Ṣahinalp. Simple and practical sequence nearest neighbors with block operations. In *Proceedings of 13th Annual Symposium on Combinatorial Pattern Matching*, Lecture Notes in Computer Science, pages 262–278, 2002.

[MSU97] K. Mehlhorn, R. Sundar, and C. Uhrig. Maintaining dynamic sequences under equality tests in polylogarithmic time. *Algorithmica*, 17(2):183–198, 1997.

[MTZ01] Y. Minsky, A. Trachtenberg, and R. Zippel. Set reconciliation with nearly optimal communication complexity. In *Proceedings of the IEEE International Symposium on Information Theory*, page 232, 2001.

[MVS01] D. Moore, G. M. Voelker, and S. Savage. Inferring internet denial of service activity. In *Proceedings of the Usenix Security Symposium*, pages 9–22, 2001.

[Mye86] E. W. Myers. An $O(ND)$ difference algorithm and its variations. *Algorithmica*, 1:251–256, 1986.

[Nav01] G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.

[Net] Cisco NetFlow. More details at `http://www.cisco.com/warp/public/732/Tech/netflow/`.

[NH94] R. T. Ng and J. Han. Efficient and effective clustering methods for spatial data mining. In *Proceedings of the 20th International Conference on Very Large Databases*, pages 144–155. Morgan Kaufmann, 1994.

[Nis92] N. Nisan. Pseudorandom generators for space-bounded computation. *Combinatorica*, 12:449–461, 1992.

[NOA] NOAA. National Oceanic and Atmospheric Administration, U.S. National Weather Service. `http://www.nws.noaa.gov/`.

[Nol] J. Nolan. Stable distributions. Available from `http://academic2.american.edu/~jpnolan/stable/chap1.ps`.

[NT84] J. H. Nadeau and B. A. Taylor. Lengths of chromosome segments conserved since divergence of man and mouse. *Proceedings of the National Academy of Science of the USA*, 81:814–818, 1984.

[OBCO00] G. Ozsoyoglu, N. H. Balkir, G. Cormode, and Z. M. Ozsoyoglu. Electronic books in digital libraries. In *Proceedings of IEEE Advances in Digital Libraries (ADL)*, pages 5–14, 2000.

[Orl91] A. Orlitsky. Interactive communication: Balanced distributions, correlated files, and average-case complexity. In *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science*, pages 228–238, 1991.

[PG86] K. F. Pang and A. El Gamal. Communication complexity of computing the Hamming distance. *SIAM Journal on Computing*, 15(4):932–947, 1986.

[PS85] F. P. Preparata and M.I. Shamos. *Computational Geometry : An Introduction*. Springer-Verlag, 2nd edition, 1985.

[PTVF92] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, 2nd edition, 1992.

[PW72] W. W. Peterson and E. J. Weldon, Jr. *Error-Correcting Codes*. M.I.T. Press, 2nd edition, 1972.

[Raz92] A. A. Razborov. On the distributional complexity of disjointness. *Theoretical Computer Science*, 106(2):385–390, 1992.

[SBB90] T. Schwarz, R. W. Bowdidge, and W. A. Burkhard. Low-cost comparisons of file copies. In *10th International Conference on Distributed Computing Systems*, pages 196–202. IEEE, 1990.

[SK83] D. Sankoff and J. B. Kruskal. *Time Warps, String Edits and Macromolecules: the Theory and Practice of Sequence Comparison.* Addison Wesley, 1983.

[Sma] California PATH Smart-AHS. More details at `http://path.berkeley.edu/SMART-AHS/index.html`.

[SN96] D. Sankoff and J. Nadeau. Conserved synteny as a measure of genomic distance. *DAMATH: Discrete Applied Mathematics and Combinatorial Operations Research and Computer Science*, 71:247–257, 1996.

[SS02] D. Shapira and J. A. Storer. Edit distance with move operations. In *Proceedings of the 13th Symposium on Combinatorial Pattern Matching*, volume 2373 of *Lecture Notes in Computer Science*, pages 85–98, 2002.

[Sto88] James A. Storer. *Data Compression: Methods and Theory.* Computer Science Press, 1988.

[SU98] J-R Sack and J. Urrutia, editors. *Handbook on Computational Geometry.* Elsevier Science, 1998.

[ṢV94] S. C. Ṣahinalp and U. Vishkin. Symmetry breaking for suffix tree construction. In *Proceedings of the 26th Symposium on Theory of Computing*, pages 300–309, 1994.

[ṢV96] S. C. Ṣahinalp and U. Vishkin. Efficient approximate and dynamic matching of patterns using a labeling paradigm. In *Proceedings of the 37th Symposium on Foundations of Computer Science*, pages 320–328, 1996.

[Tic84] W. F. Tichy. The string-to-string correction problem with block moves. *ACM Transactions on Computer Systems*, 2(4):309–321, 1984.

[TM96] A. Tridgell and P. Mackerras. The rsync algorithm. Technical Report TR-CS-96-05, Department of Computer Science, The Australian National University, 1996.

[Tri] A. Tridgell. rsync: File synchronisation utility. Available from `http://rsync.samba.org`.

[Tsa00] P. Tsaparas. Nearest neighbor search in multidimensional spaces. Technical Report 319-02, Department of Computer Science, University of Toronto, 2000.

[Ukk92] E. Ukkonen. Approximate string-matching with q-grams and maximal matches. *Theoretical Computer Science*, 92(1):191–211, 1992.

[Uni] Unidata. Atmospheric data repository. `http://www.unidata.ucar.edu/`.

[V^+01] J. C. Venter et al. The seqence of the human genome. *Science*, 291:1304–1351, 2001.

[Wag75] R. A. Wagner. On the complexity of extended string-to-string correction problem. In *Proceedings of the 7th ACM Symposium on the Theory of Computing*, pages 218–223. ACM Press, 1975.

[WDM00] M. Walter, Z. Dias, and J. Meidanis. An approximate algorithm for transposition distance. In *Proceedings of String Processing and Information Retrieval*, pages 199–208, 2000.

[Wei73] P. Weiner. Linear pattern matching algorithm. In *Proceedings of the 14th Annual IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[ZJ94] K. Zhang and T. Jiang. Some MAX SNP-hard results concerning unordered labeled trees. *Information Processing Letters*, 49(5):249–254, 1994.

[ZL77] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, IT-23:337–343, 1977.

[ZRL96] T. Zhang, R. Ramakrishnan, and M. Livny. BIRCH: an efficient data clustering method for very large databases. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 103–114, 1996.

[ZS89] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal of Computing*, 18:1245–1262, 1989.

[ZSS92] K. Zhang, R. Statman, and D. Shasha. On the editing distance between unordered labeled trees. *Information Processing Letters*, 42(3):133–139, 1992.