HIGHER COMPRESSION FROM THE BURROWS-WHEELER TRANSFORM WITH NEW ALGORITHMS FOR THE LIST UPDATE PROBLEM

Brenton Chapin

Thesis Prepared for the Degree of
DOCTOR OF PHILOSOPHY

UNIVERSITY OF NORTH TEXAS
May 2001

Chapin, Brenton. Higher Compression from the Burrows-Wheeler Transform with New Algorithms for the List Update Problem. Doctor of Philosophy, Computer Science, May 2001, 101 pp., 18 tables, 4 figures, 53 references.

Burrows-Wheeler compression is a three stage process in which the data is transformed with the Burrows-Wheeler Transform, then transformed with Move-To-Front, and finally encoded with an entropy coder. Move-To-Front, Transpose, and Frequency Count are some of the many algorithms used on the List Update problem. In 1985, Competitive Analysis first showed the superiority of Move-To-Front over Transpose and Frequency Count for the List Update problem with arbitrary data. Earlier studies due to Bitner assumed independent identically distributed data, and showed that while Move-To-Front adapts to a distribution faster, incurring less overwork, the asymptotic costs of Frequency Count and Transpose are less.

The improvements to Burrows-Wheeler compression this work covers are increases in the amount, not speed, of compression. Best x of $2x-1$ is a new family of algorithms created to improve on Move-To-Front's processing of the output of the Burrows-Wheeler Transform which is like piecewise independent identically distributed data. Other algorithms for both the middle stage of Burrows-Wheeler compression and the List Update problem for which overwork, asymptotic cost, and competitive ratios are also analyzed are several variations of Move One From Front and part of the randomized algorithm Timestamp. The Best x of $2x-1$ family includes Move-To-Front, the part of Timestamp of interest, and Frequency Count. Lastly, a greedy choosing scheme, Snake, switches back and forth as the amount of compression that two List Update algorithms achieves fluctuates, to increase overall compression.

The Burrows-Wheeler Transform is based on sorting of contexts. The other improvements are better sorting orders, such as "aeioubcdf..." instead of standard alphabetical "abcdefghi..." on English text data, and an algorithm for computing orders for any data, and Gray code sorting instead of standard sorting. Both techniques lessen the overwork incurred by whatever List Update algorithms are used by reducing the difference between adjacent sorted contexts.

ACKNOWLEDGEMENTS

For my ultimate scholarship providers, my parents.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

1.1 The Compression of Data

1.1.1 What is data?

Data conveys information. Data takes many forms. Some kinds of data are numerical values obtained by measurements of phenomena such as the digitized output of a microphone, or values generated by computation such as the digits of π. Other kinds consist of symbols which form a string from a language such as English, Fortran, or DNA.

The purpose of data is, ultimately, to convey information. Storage of data is not an end in itself. Data is stored because it may be needed in the future. Representation of data is usually tailored to the uses of that data.

Preferred representations of data evolved, or were designed, to ease usage of the conveyed information. Another consideration is simplicity of the representation. Early writing systems are generally difficult to learn and use, ambiguous, and limited compared to modern systems. Changing technology has reduced the need for systems suitable for clay tablets or stone (runic systems in which the characters contain only straight lines, since curves are difficult to carve), or signal towers with their firelight and shutters, and many others. Also, technology has inspired modifications suitable for new mediums such as the 7 segment displays on calculators and digital watches. Writing with pens in cursive script is becoming less common, displaced somewhat by typing on keyboards. Perhaps future systems will make current ones look equally awkward. Whatever the reason for a representation's form, compactness is often, but not always, a low priority in its design or evolution.

1.1.2 What is data compression?

Most representations of information do not optimize usage of resources. For example, one could replace every "qu" in English with a 'q', removing a lot of u's. Likely, the text will convey the same information as before, but take less space. Why is no information lost? Because those u's are redundant. Lossless compression of data is the removal of redundancies in the data while preserving the ability to return the data to its original form. Lossy compression involves the discarding of "unimportant" information from the data which then cannot be recovered. Although people have developed clever methods for determining what information is important, the ultimate judge continues to be subjective human evaluation.

"Removal" is an excellent argument against claims of infinite compression achieved through recursive methods. Ask the claimant how it is possible to remove something, such as redundancies or unimportant information, more than once. On the other hand, no compression method is perfect, so another pass or more over the data may be worthwhile.

Other methods commonly referred to as data compression are transformations to more compact representations (often followed by some simple entropy coding) which remove little of the repetition in the data. The popular MP3 lossy compressed sound format varies the amount of information thrown out to fit the same unit of time into the same space no matter how complicated the sound. An undesirable effect is that silence is not compressed well. A minute of total silence will take almost the same space as a minute of song. Also, repeats in the sound, common in music (the refrain of a song, for example), are redundancies that are not removed by MP3 [37]. Transformations can serve to model the data, improving the performance of classic universal data compression algorithms as in the PNG lossless image compression format which uses a dictionary based compression technique as its back end [52]. Lossy methods use transforms and models to segregate data into varying shades of important and less important parts. Trade-offs between amount of compression and quality of representation are made by varying the amount of the less important information that is thrown out.

1.1.3 Why compress?

Some resources used to convey and/or hold data are phone lines, books, and compact discs. No resource, however inexpensive, is cost free. Representing information as compactly as possible saves resources.

One may ask, why not just use the most compact representation instead of translating back and forth between compressed and uncompressed versions of the same information? A space efficent representation is usually not time efficient. For example, it is faster to look up a million digits of π than to compute them. But the most compact representation of π might be something like the following simple description: "area of a circle of radius r divided by area of a square of width r" translated into a program.

The above descripition of π is an example of Kolmogorov complexity, a measure of information [33]. The Kolmogorov complexity of a collection of data is the smallest program (in a reasonable language) that can generate that data. Finding the Kolmogorov complexity is an incomputable problem since an answer also answers the halting problem. For instance, possessing a "no" answer to the yes/no question "Is the Kolmogorov complexity of some data d less than c?" means that the halting question is answered for all programs of length less than c. But the Kolmogorov complexity of some data, such as the digits of π, is easily shown to be $O(\log n)$ where n is the number of digits to compute. One may generate infinitely many digits of π with a constant length program.

1.2 A short history of lossless data compression

1.2.1 Early work

The seminal paper on information theory is Claude Shannon's 1948 paper, "A mathematical theory of communications" [46] which gives the only function, log, that fits the natural properties of information. The first property is high probability events convey less information than low probability ones. For example, suppose one was working on a word in a crossword puzzle and one knew one letter of that word. If that letter is a ‘z’, one has more information (less words to try) than if that letter is an ‘e’. This and other axiomatic properties of information lead to a formula in which the probability p_s of a string s is inversely proportional to the amount of information I it conveys:

$$I(s) = -\log p_s$$

The expected value of the information, taken over all strings s generated by a source S over an alphabet A, is the *entropy*, H, of S, so

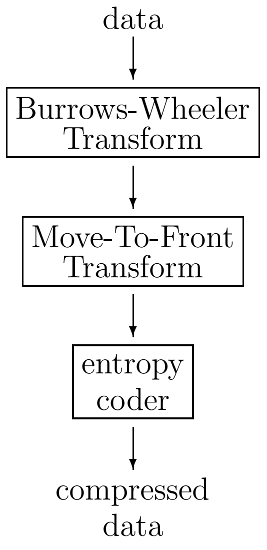
$$H(S) = -\sum_{s \in A} p_s \log p_s.$$

In 1952, Huffman published an algorithm for computing minimum redundancy codes, now called Huffman codes [24]. Huffman codes are optimal per symbol of data generated by an independent, identically distributed (i.i.d.) source. However, Huffman codes applied character-by-character are not optimal for strings. Arithmetic coding (Cleary and Witten, 1984 and Langdon, 1984) [16, 29] is optimal for strings from an i.i.d. source. Both reduce redundancy by using less resources for more common symbols at the expense of using more resources for less common symbols, rather than using the same amount for each symbol, for a net gain. The term “entropy coding” refers to the many variants of Huffman and arithmetic coding. One such, probability rank encoding [20], assigns codes to symbols as does Huffman, but can encode infinitely large alphabets (such as the set of whole numbers), whereas Huffman coding operates on an alphabet containing a finite number of symbols. But probability rank encoding is not efficient except on very specific distributions.

Much data has context and so is not modelled well by i.i.d. sources. Algorithms for compressing such data are mostly dictionary based or Prediction by Partial Match (PPM, 1984) [16]. In 1977 and 1978, Lempel and Ziv published two data compression algorithms: LZ77, the sliding window algorithm in which the dictionary is implicit [31], and LZ78 with its explicit dictionary [32]. In actuality, these algorithms model the data and then use entropy coding to compress the model. The conceptual separation of “model” and “coder” has helped clarify much of the subtle differences between algorithms. A further conceptual refinement is the separation of the statistics from the modeling and coding. The statistics are simply the counts of occurrences of items defined in the model.

LZ77 and LZ78 are fast ($O(m \log b)$ for a string of length m and a dictionary of size b) and compress optimally in the limit, but do not converge to optimality at the optimal rate [34]. PPM is optimal and converges at the optimal rate for many sources but is very slow. In 1994, Burrows and Wheeler introduced their block sorting algorithm [13] which is nearly as fast as LZ77 and compresses nearly as highly as PPM. Subsequent work has shown that context trees can produce the same models as both Burrows-Wheeler and PPM, making them roughly equivalent [30], and that PPM can be made to perform as fast as Burrows-Wheeler [19].

An important factor in the development, use, and refinement of all these techniques has been the legal issue. Many nations have patent and copyright laws which are intended to promote innovation by not allowing people to use or profit from innovations without arrangements with the inventors. In data compression, this has resulted in many more improvements to and widespread use of the unpatented LZ77 algorithm, embodied in the free `gzip` compression program, than to the patented extensions of the LZ78 algorithm, as seen in the gradual abandonment of the GIF compressed image format in favor of the newer PNG compressed image format. Although the patent on GIF expires soon, technology has left it behind. Current hardware supports millions of colors, so few will now want to use GIF with its 256 color limit. The Burrows-Wheeler Transform is unpatented. However, Julian Seward's free BWT based program `bzip` did not gain much acceptance in part because it used arithmetic coding, some variations of which are patented. In `bzip2` [44], error checking was added and the arithmetic coding was replaced with Huffman coding. Now `bzip2` is widely used, is included as a standard part of many distributions of the free operating system Linux, and is beginning to replace `gzip`, feats not managed by any of the hundreds of other compression programs written since `gzip`.



Burrows-Wheeler compression

1.2.2 Burrows-Wheeler Compression

The Burrows-Wheeler Transform (BWT) does not actually compress data. Burrows-Wheeler compression, as originally described, has 3 stages. First the data is transformed with the Burrows-Wheeler Transform, then further transformed with Move-To-Front, and finally compressed with an entropy coder.

Each of the 3 stages can be improved. Also, data can be better prepared for compression, much like the BWT prepares context sensitive data for entropy coding. Ways to encode the BWT output directly (combining the last 2 stages into 1) have been considered. These improvements came in part from a better understanding of the kinds of data involved.

1.3 Contributions of This Dissertation

This dissertation contains new material and results on the following:

- Improvements to Burrows-Wheeler compression
    - Better alphabet orders
    - Binary reflected Gray code sort
    - Encoding output of Burrows-Wheeler Transform by switching between a Move-To-Front improvement and Best x of $2x - 1$
- Design of a new family of algorithms called "Best x of $2x - 1$" encompassing Move-To-Front, Timestamp, and Frequency Count
- Analysis of overwork of Move-To-Front, Move 1 From Front, and Best x of $2x-1$
- Tight competitive ratios for two versions of Move 1 From Front

1.4 Organization of Dissertation

A familiar problem for writers of technical papers is how to organize subjects into a linear progression when a graph of their relationships forms a network. Maybe in the future hypertext will be employed. But for the present, the linear order of the stages of Burrows-Wheeler compression has been used to impose a linear order upon the material within this dissertation. Chapter 2 introduces *overwork*, the cost to an adaptive algorithm of making adaptations, used for much of the analysis in this thesis. Chapter 3 covers some of the preprocessing that can be applied before performing the BWT. A method for computing sort orders is presented. Chapter 4 considers modified sorting for the Burrows-Wheeler, specifically, Gray code sorting. Chapter 5 contains the analysis of Move-To-Front and variants from the viewpoint of steady state cost and overwork on pieces of i.i.d. data. Chapter 5 also has some of the modifications to the final stage of Burrows-Wheeler compression, the entropy coding. In conclusion, chapter 6 summarizes and points out directions for further work.

Parts of this dissertation have been previously published in the Proceedings of the Data Compression Conference as “Higher Compression from the Burrows-Wheeler Transform by Modified Sorting” by Brenton Chapin and Stephen R. Tate [15] and “Switching Between Two On-line List Update Algorithms for Higher Compression of Burrows-Wheeler Transformed Data” by Brenton Chapin [14].

CHAPTER 2

Asymptotic Cost and Overwork

2.1 Introduction

The proofs of optimality for dictionary compression (LZ78 and LZ77) and Prediction by Partial Match (PPM) assume the data is stationary and also ergotic, but there is very little real data that fits that criteria. For example, in a story, characters come and go, and the plot moves forward. The words are not random; the data is not stationary.

In addition to the measures of running time and space, algorithms are judged by the cost or quality of their solutions when a measure of cost is available and an optimal cost algorithm is not feasible. Many on-line problems such as cache updating or intractable problems such as the NP-hard set have a cost metric. In the case of data compression algorithms, the usual cost measure is the amount of compression achieved. Typically, interesting costs are the asymptotic cost, which is the limiting cost to an adaptive algorithm as the problem size grows arbitrarily large, or average and worst-case costs which are the costs on average instances and worst-case instances of data. Another is amortized cost, the cost of an operation taken as an average over a sequence of requests from any instance (even a worst-case one) of a problem.

If one discards the assumption of stationarity, another aspect of any adaptive data compression algorithm becomes important: the overwork. The overwork is the extra cost (beyond the asymptotic cost) incurred by an algorithm as it adapts to new data. Ordinarily, overwork can be “swept under the rug” since it is only the vanishingly small constant of initial adaptation to the data, after which the asymptotic cost dominates. But if the data changes from time to time, so that instead of just one initial adaptation many adaptations are needed, overwork becomes significant.

In fact, one could say overwork has been considered already in comparing LZ78 to PPM. Both are asymptotically optimal. Yet PPM achieves higher compression in practice. The difference lies in the speed of the two algorithms' adaptation to data. PPM adapts to data at the optimal rate. LZ78 does not [47].

In this chapter, terminology and definitions used throughout this work are listed next, followed by some analysis of overwork for various algorithms.

2.2 Definitions

2.2.1 Some common terms

symbol: The unit of language, such as a letter of the English alphabet.

alphabet: The set of symbols of which data may be comprised.

source: The source of the input data. The data may be characterized as a string belonging to a language (such as a sentence in English), or a sequence of unrelated requests for symbols from the alphabet, or in many other ways.

independent: Probabilities of symbols are not context dependent.

identically distributed: Probabilities of symbols do not change.

stationary: Probabilities are independent of time t

ergotic: The finite state machine of contexts contains no subset from which egress is impossible, and the greatest common denominator of the lengths of all the cycles is 1.

Zipf distribution: Similar to the informal "80-20" rule which states that 80% of the requests are for the 20% most common symbols. Formally, $p_i = 1/(iH_n)$ where $H_n = \sum_{j=1}^{n} 1/j$ is the nth harmonic number.

2.2.2 What the mathematical symbols represent

A: The source alphabet. A is an ordered set and is used for sorting in the BWT.

d: The input data. $d \in A^*$. $|d|$ is the length of d, denoted by m.

L: Ordered list maintained by an algorithm which may reorder the list. Contains the same symbols as A.

$\rho(l)$: rank of symbol $l \in L$. Symbol at front has rank of 1.

s: $= (s_1, s_2, ..., s_m)$. Sequence of requests for symbols in L.

s_i^j: A substring (piece) from the ith to the jth character of s, inclusive.

I: Intervals. Ordered indexes of dividing points of s. $I \subset \mathbf{N}$. Let I_0 denote 0 and not be counted as a member of I. Then the kth piece of s is $s_{I_{k-1}+1}^{I_k}$ for $1 \leq k \leq |I|$, abbreviated as $s_{\overline{k}}$.

$\tau(L)$: permutation of L.

c: Cost of serving a request in s. Two parts to the cost:

> c_1: Cost of accessing L.
>
> c_2: Cost of a permutation $\tau(L)$.

n: $= |A| = |L|$.

m: $= |d| = |s|$.

t: Time, or number of symbols processed. $0 \leq t \leq m$.

P, Q: Probability distributions.

E: Expected cost.

OV: Overwork is the expected cost minus the asymptotic cost. This amount is usually positive, hence the "over" in the term [11].

2.2.3 Important compression algorithms

entropy coding: Methods for homogenous data with no context dependencies

- **Huffman**: Optimal integer sized codes for each symbol
- **arithmetic**: Optimal fractional sized codes for each symbol
- **probability rank**: Integer sized codes for symbols from open-ended alphabets such as the set of whole numbers, ordered by probability of occurrence

universal compression: Methods for context sensitive data that are asymptotically optimal

- **LZ77**: Limpel and Ziv's sliding window (which implies a dictionary) compression algorithm, published in 1977, and the basis of the popular `zip` and `gzip` programs [31].
- **LZ78**: Limpel and Ziv's dictionary compression algorithm, published in 1978 [32].
- **PPM**: Prediction by Partial Match. Predict the next symbol by matching the last several symbols with past data. For instance, to predict a continuation for English "the", one might look back and count many "the ", "them", "then", "there", "these", "they" and a few "theater", "theocrat", and "thew" and assign probabilites accordingly [16].
- **BWT**: Burrows-Wheeler Transform. Based on sorting, for which fast and efficient methods are well known, the transform rearranges the data, grouping symbols with similar contexts near each other [13].

2.2.4 Identically distributed sources

i.i.d., independent: Each request in the sequence is independent of all other requests. $Prob(s^m) = \prod_{i=1}^{m} Prob(s_i)$.

p.i.i.d., piecewise independent: s, the sequence of requests is $|I|$ concatenated i.i.d. sequences $s_{\overline{k}}$, $1 \leq k \leq |I|$. Thus, $Prob(s_{\overline{k}}) = \prod_{I_{k-1}<j\leq I_k} Prob(s_j)$

i.p.i.d., independent piecewise: Same as piecewise independent plus the *pieces* are independent. Thus, $Prob(s) = \prod_{k=1}^{|I|} Prob(s_{\overline{k}})$.

o.i.i.d., ordered independent: $Prob(s^m) = \prod_{i=1}^{m} Prob(s_i)$ and $Prob(l_i) \geq Prob(l_j)$ for $i < j$. In other words, the list L is ordered by the probability of the symbols from most to least probable.

o.p.i.i.d., ordered piecewise independent: Same as piecewise independent plus $Prob(l_i) \geq Prob(l_j)$ for $i < j$ for all pieces $s_{\overline{k}}$.

b.i.p.i.d., binary i.p.i.d.: Same as i.p.i.d. plus $|A| = 2$ [51].

2.2.5 Competitive Analysis

deterministic: For a given input, a deterministic algorithm always makes the same decisions and produces the same results.

randomized: A randomized algorithm uses random values to make some decisions. Hence, an input may produce different results in separate trials.

on-line: Algorithm cannot use problem data (s) that has not yet been processed. At time t, algorithm is limited to s_1^t

off-line: Algorithm may use any data in s, including future requests, at any time t.

adversary: Source of data which attempts to maximize (hence, adversarial) the ratio of cost between some algorithm B and an optimal algorithm.

List Update: One of the first problems analysed competitively. Let L be an ordered list of items, and s^m be a sequence of m requests for these items. To satisfy a request for an item s_t, compare s_t to members of the list L, starting with the first member and proceeding in order until a match is found. One may pay proportionally to change the order of L between requests; most algorithms make simple changes for free. The problem is to minimize the number of comparisions needed to satisfy all the requests, plus the amounts paid to reorder L.

Algorithms for the List Update problem

Let l be the requested item. We give a short description of how each algorithm changes the list of items L after serving a request for item l.

MTF: Move-To-Front. Move l to front of list (rank 1).

TS: Timestamp. With probability p, move l to front of list, else move l in front of frontmost item requested at most once since last request for l [1].

FC: Frequency Count. Move l in front of all items requested less frequently.

TP: Transpose. Swap l with item in front of l.

M1FF: Move 1 From Front. Move l to rank 2 if it was at rank 3 or greater, else move l to the front [7].

Bx: Best x of $2x-1$. Move l in front of each item requested less than x times of the last $2x-1$ requests for l or that item [14].

2.3 The Burrows-Wheeler Transform

In this section we give a description of the Burrows-Wheeler Transform [13].

To perform the Burrows-Wheeler Transform on a string of length $m-1$, append a unique character. Generate all m suffixes of the appended string, sort the suffixes, and output the character preceding each of the sorted suffixes. (The unique character appended to the original string is the one that "precedes" the original string.) An example on the input string "alfalfa" is shown in tables 2.1 and 2.2. The output is "aff$llaa"

The inverse transform is even easier. The first column of the sorted strings may be obtained from the output of the BWT with a simple $O(m)$ bucket sort. The output column and the first column form all pairs that occurred in the original string. For all m symbols, point the ith occurrence of a symbol s in the first column to the ith occurrence of s in the output column. Then, starting at the unique symbol that was appended to the original input, the pointers form a chain of pairs that is the original

alfalfa$
lfalfa$
falfa$
alfa$
lfa$
fa$
a$
$

Table 2.1: All Substrings of “alfalfa$”

a	$
f	a$
f	alfa$
$	alfalfa$
l	fa$
l	falfa$
a	lfa$
a	lfalfa$

Table 2.2: Substrings of “alfalfa$”, sorted in alphabetic order “$afl”. Preceding symbols shown to left of sorted substrings.

a	$
f	a
f	a
$	a
l	f
l	f
a	l
a	l

Table 2.3: Output column and first column

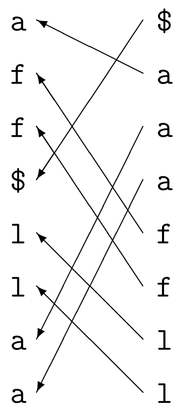


Figure 2.1: Pointers for inversion

input. Follow the pointers to recover the original data. The inverse BWT is shown in table 2.3 and figure 2.1.

There are some minor variations on the BWT that are essentially just different ways of describing the process. The unique character appended to the string may occur in an arbitrary place in the sort order but is usually at the low or high end. The "alfalfa" example arbitrarily set '$' at the low end of the sort order. The unique character is not necessary if instead a pointer to the starting character is included with the output. Rotations of the input string may be used in place of suffixes, in which case the output can be described as the last characters of the sorted rotations. The input string may be reversed so that the suffixes of the input are the prefixes of the original data, giving more traditional prefix based contexts. These alterations usually have little effect on the compression of BWT output.

An easily varied parameter of the BWT is the block size, which is the maximum amount of data to sort. If computing resources are insufficient to sort all the data in one large block, the data can be split into multiple blocks. This usually decreases compression efficiency. However, data that is different should not be lumped into the same block as the amount of compression will decrease. An example is the Calgary Corpus. Higher compression of the corpus is achieved by compressing each file separately rather than concatenated together in one large block.

By sorting, the BWT groups similar contexts near each other. As one might expect, characters preceding similar contexts are themselves similar. This transformed data can be highly compressed with fast and simple algorithms. An example of BWT output is shown in Table 2.4. The example shows the preceding characters of a contiguous group of sorted suffixes from Book1 of the Calgary Corpus. Where the context remains nearly the same, the set of preceding characters does likewise. For example, only 'p' or 'l' precedes "osed". The next context may be preceded by some of the same characters, as in the switch to "osely", or totally different characters as happens when switching from "osening" to "oseph". An encoder of BWT output should compress large quantities of symbols from sets of small cardinality well, and at the same time quickly adapt to new sets of symbols.

2.4 Approaches to Coding of BWT Output

In Burrows and Wheeler's seminal paper, BWT output was transformed with Move-To-Front. The MTF output was then encoded by run-length encoding of 0's and standard entropy coding. They noted that MTF followed by entropy coding gave more compression than the "superior" dictionary techniques. (The dictionary techniques are superior to MTF on most data, but not on output of the BWT.) Several refinements to MTF and entropy coding have been presented. Fenwick retained the run-length coding and improved the entropy coding by modeling the logarithms of the MTF output, thus allowing the coder to adapt more quickly to changes [21].

preceding character	context
l	osed, steps crossed the hall, an
p	osed, the Bath escapade/being qu
l	osed./ "Who is Mr. Boldwood?’ s
l	osed./" My life is a burden with
l	osed./<C xi>/<P 134>/OUTSIDE THE
l	osed./This night the buildings w
p	osed. However, it was too/late n
l	osed. My/uncle has a hut like th
l	osed.’/"Yes I suppose I should,’
o	osely/put together, and the flam
l	osely at the hot wax to discover
l	osely beheld. By degrees a/more
l	osely compressed was his mouth,
l	osely considered. what he would
l	osely drawn over the/accommodati
l	osely huddled, and outside these
o	osely thrown on, but not/buttone
l	osely, and his compressed lips m
h	osen being always on the side aw
h	osen by women as best. All/featu
h	osen it on/that account for his
h	osen my course. A runaway wife/i
o	osen my hand; I will, indeed I w
h	osen points, where they fed, hav
h	osen the front of her bodice as
o	osened/his woollen cravat, and c
o	osened his hand, saying, ’By Hea
o	osened tooth or/a cut finger to
o	osened, rose to the surface,/and
l	oseness of his cuts, that had it
o	osening/his neckerchief./On open
o	osening her tightly clasped arms
J	oseph/Poorgrass in a voice of th
J	oseph/Poorgrass o’ Weatherbury,

Table 2.4: Some sorted suffixes and preceding characters from Book1 of the Calgary Corpus

Balkenhol, Kurtz, and Shtarkov have further improved the basic technique by replacing MTF with several variations on a slightly different algorithm, called Move 1 From Front (M1FF) in this work, and the run-length coding of 0's with an order-3 Markov coder of 0's and 1's (2 or greater must still be encoded with the entropy coder), and other refinements [6, 7]. None of these methods are radical departures from Burrows and Wheeler's approach of having an algorithm for transforming the localized concentrations of similar symbols into a global preponderance of low values, followed by entropy coding.

The problem of how best to compress the output of the Burrows-Wheeler Transform is difficult. Before trying to answer hard questions, one should consider: Are they the "right" questions? Can they be answered? Will the answers help solve the problem? Perhaps the first questions concern the BWT: does the BWT help compress data and, if so, is the BWT worth using instead of other, possibly better, methods? Empirical data (obtained by such means as comparing amount and speed of compression of BWT algorithms with dictionary and other methods on suites of test files, performed on the same machine, with the same compiler and optimizations, etc.) suggest that, yes, BWT methods are viable.

Since empirical results were good, one might next wonder how BWT compares with dictionary and other algorithms on the theoretical front. Several works have proved, with limitations, that Burrows-Wheeler compression is optimal on stationary, ergotic data [18, 30, 35]. Thus, the theoretical work is about on par with the proofs for dictionary compression.

With the promising results on BWT based compression, an examination of the workings of the BWT, in particular, the kinds of input for which BWT performs best and good characterizations of BWT output, provides a starting point towards higher compression, the goal of this work. How should the output of the BWT be described? Piecewise independent identically distributed is one way. A similar view is as independent, but with a smoothly varying change between distributions, rather than as identically distributed pieces (no change in distribution within pieces and then abrupt changes at piece edges). A different approach is Inversion Coding, presented by Arnavut [4], an off-line technique to find and encode a representation of BWT output as an inversion of a permutation of a *multiset*. In the simplest form, each element of a multiset is paired with a frequency, indicating the number of occurrences of that element. BWT output can be characterized as a permutation of a multiset.

Another method of encoding BWT output is somewhat like run-length encoding or the interval coding of Elias [20]. Elias noted that Recency Ranking (his name for MTF) will always produce equal or smaller values than Interval Coding, which, for each symbol, outputs the number of characters (the interval) to the last occurrence of that symbol in the sequence. However, better encoding rules, such as only counting those symbols which precede the current symbol in the alphabet A (and in one variation, encoding only a count for the first symbol in A instead of a 0 value for each occurrence of that symbol), can make an interval coder compress in an MTF coder's league.

2.4.1 Inversion Coding

Experimental results show that Inversion Coding compresses slightly more than improvements on Move-To-Front techniques, but is a relatively slow $O(m \log m)$ time. What is not clear is how else Inversion Coding may be compared to Move-To-Front, say by formulating Inversion Coding in terms of asymptotic cost and overwork.

What follows is a brief description of inversion coding, with an example given in Table 2.5 (The example uses capital letters 'A' thru 'I' to represent numbers 10 thru 18):

1. Assign the numbers $1...m$ to each symbol in the sequence of m symbols as follows. Symbols preceding other symbols in alphabetical order have smaller numbers. All 'a's are less than all 'b's which are less than all 'c's and so on. Same symbols occurring sooner in the sequence have smaller numbers. The first 'a' is less than the second 'a' and so forth. This produces the linear index permutation.

2. Each entry in the inversion is the count of numbers in the linear index permutation that come after and are smaller than the corresponding entry of the linear index permutation.

3. Take the difference of the counts of successive occurrences of the same symbol in

sequence	acaabaaabcbaaccaaa
Move-To-Front (start list *abc*)	021021001212020100
Move 1 From Front	020020001202121100
interval coding (prepend *cba*)	031051003713030200
better interval coding	030040003700030000
better interval coding with count	.3..4...370..30...
linear index permutation	1F23C456DGE78HI9AB
inversion	0D0080005650033000
an inversion coding	0D0080003700030000

Table 2.5: Inversion Coding and dynamic List Update algorithms on a sample sequence

the inversion to obtain an inversion code. Reversing the encoding is possible because data on frequency and first and last position of each symbol is kept, adding a small amount of overhead.

Other inversions are possible, by using counts of numbers that are to the left and smaller, or to the right and larger, or to the left and larger.

Interestingly, the 2 approaches, inversion coding and interval coding with improvements, produce similar output. We show that, except for the initial symbols, the two methods produce identical output. Despite the similarity in outputs, this seems to be a previously unknown fact!

Theorem 1 *Except for the initial occurrence in the input string of each symbol in the alphabet, better interval coding and Inversion Coding produce identical output.*

Proof: Each entry in the inversion is the count of the number of values in the linear index permutation that are to the right of its position and smaller than its corresponding value in the permutation. A value to the right can only be smaller if it represents a symbol that occurs earlier in the alphabet, so each value is a count of all later occurrences of all symbols preceding the represented symbol in the alphabet. The inversion coding is the difference between successive values representing the same symbol, and is therefore the number of all symbols of earlier alphabetic order between the current and previous occurrence, which is precisely the way in which the better interval coding is computed. Only the initial values, where there is no previous occurrence of an individual symbol with which to take a difference, are different. ▮

Corollary 1 *Inversion Coding can be performed on-line in $O(m \log n)$ time.*

Thus Inversion Coding can match the speed of many List Update algorithms, including MTF.

2.4.2 Dynamic Update of a List of the Symbols in BWT Output

The List Update problem and Move-To-Front (MTF) algorithm has been studied since the 1960's, well before the advent of Competitive Analysis in the mid 1980's. Early work analysed the asymptotic cost of the MTF rule on arbitrary data and on very specific kinds of independent identically distributed (i.i.d.) data such as Zipf distributed i.i.d. data and data drawn from a 3 or 2 letter alphabet. Bitner analysed the overwork of MTF and several other dynamic list update algorithms [11]. In all cases, an access cost function $c = \rho(l)$ was used. The optimal worst-case cost for an on-line deterministic algorithm was shown to be 2 times the cost of an optimal off-line algorithm. Later work showed that the proofs of optimality held for any convex cost function such as $c = \log \rho(l)$ which is close to the cost to an entropy coder [48]. A probability rank encoder can encode with cost function $c = \log \rho(l) + 2 \log \log \rho(l)$.

Viewing the output of BWT as piecewise independent identically distributed (p.i.i.d.) makes for easy analysis in some cases. The analysis of some of the List Update algorithms, such as MTF, on p.i.i.d. data is fairly easy. Other algorithms, such as Transpose (TP), remain difficult to analyze. The p.i.i.d. model of BWT output has been the basis of some improvements in the amount of compression achieved. A point to make here is that, strictly speaking, the output of BWT is not p.i.i.d., though close enough to achieve practical improvements. A fuller explanation of this point is given in chapter 4.

Taking piecewise independent identically distributed as a reasonable model of BWT output, how should p.i.i.d. data be analysed? Again, there is more than one approach. Older methods focused on the asymptotic cost of a solution, which is fine for i.i.d. data but not p.i.i.d. Taking the overwork into account, ala Bitner, addresses this shortcoming, and explains why Move-To-Front (MTF) is, in practice, much better than the optimal asymptotic cost algorithms Transpose (TP) and Frequency Count. Competitive Analysis, which involves worst-case analysis under varying conditions, gives a theoretical basis for the observed differences on arbitrary data between on-line algorithms such as MTF and TP, and shows the power of randomization in avoiding worst-case situations. Essentially, if worst-case scenarios are a few random instances of a problem rather than a few highly structured and much more probable than average instances (for example, deterministic Quicksort in which the pivot is always the first element in the sub problems, on data that has already been sorted) then it is far less likely that worst-case problem instances will be picked. Discussions about how to refine or replace Competitive Analysis continue; meanwhile, useful results on many on-line problems, and randomized algorithms for solving them, have been obtained with the technique. One open problem in this area is the performance of randomized algorithms on the List Update problem. The optimal competitive ratio for deterministic algorithms is known (2 minus a small constant times optimum, for the usual cost function $c = \rho(l)$), and is met by Move-To-Front. But for randomized algorithms, the current lower limit is 1.5 times optimum [49] and the best known randomized algorithm achieves 1.618 (the golden ratio) times optimum [1]. The open problem may yet be solved, or superceded along with Competitive Analysis by some new paradigm. The main results in this work do not come from applying Competitive Analysis, but instead come from an extension of Bitner's overwork analysis applied to Effros' modifications of independent identically distributed data sources.

The possible characteristics of pieces of piecewise i.i.d. deserve some attention. P.i.i.d. data may contain many small pieces and/or a few large pieces. Specifically, there may be $O(1)$ pieces of size $\Theta(m)$ or $\Theta(m)$ pieces of size $O(1)$ or a mixture. And, of course, there may be any number or size in between such as $\Theta(m/\log m)$ pieces of size $\Theta(\log m)$. With this distinction, it is worth noting that while MTF is optimal for arbitrary data, MTF is not optimal for p.i.i.d. data in which the pieces are larger than $O(1)$ size. BWT on data generated from a stationary finite context model produces $O(1)$ pieces of size $O(m)$.

For encoding, where the cost of an individual item l is $c = \Theta(\log \rho(l))$ instead of $c = \rho(l)$ and for i.p.i.d. data rather than p.i.i.d. data, Merhav [36] showed a lower limit of

$$\sum_{k=1}^{|I|} |s_{\overline{k}}| H(s_{\overline{k}}) + (1-\epsilon)\left(\frac{1}{2}n|I| + |I| - 1\right)\log m$$

for the expected cost of encoding the data, for $\epsilon > 0$ and large m. We conjecture that on i.p.i.d. data, the MTF transform followed by probability rank encoding as in the Bentley, Sleator, Tarjan, and Wei analysis on i.i.d. sources [9], is within a constant amount of Merhav's limit if the pieces are of constant size.

Another aspect of the pieces is that they may be rearranged to put similar pieces closer to one another. An ability to order the pieces is one reason the BWT output is better modeled as p.i.i.d. rather than i.p.i.d. Given 3 pieces and a measure of distance between pieces, such as the overwork involved in switching from one to another, the pieces can be ordered so as to reduce the sum of the distances between adjacent pieces, except in the highly unlikely case that the distance between all 3 pieces is the same. Of course, the cost of describing piece boundaries and an arbitrary order for the pieces could easily negate any savings obtained by reordering, particularly if there are many small pieces. On the other hand, a small set of orders, obtained by following some low cost rule such as sorting by a few particular orders, can be cheaply encoded. This rearrangement of the pieces is what the preprocessing to find a sort order and the Gray code sort described in Chapters 3 and 4 do.

Be cautioned (or glad to have suggestions for future work!) that better models may be introduced and that this work does not prove any absolutes, such as that MTF adapts to new pieces at an optimal rate (which it perhaps does not) or that a List Update algorithm transforms p.i.i.d. data into ordered p.i.i.d. data. What is shown in this work (aside from the improvements applied to Burrows-Wheeler compression and empirical data supporting those improvements) are formulae for computing the asymptotic cost and overwork of some of the List Update algorithms, and comparisons between these algorithms. Move-To-Front, the archetypical List Update algorithm, is used in support of some observations about p.i.i.d. data and to show aspects of costs which may hold (but again, no ironclad proof) for all reasonable List Update algorithms. The assertions in the next section hold for MTF, and probably M1FF, Best x of $2x-1$, TP, and FC.

2.5 Overwork Formulae

2.5.1 Overwork and steady state cost of Move-to-Front

The Move-To-Front algorithm has been studied as far back as 1965. In 1976, Rivest [40] proved that on i.i.d. data, MTF has a steady state cost of $E(Cost_{MTF}) = 1 + \sum_{i=1}^{n} p_i \sum_{j \neq i} \frac{p_j}{p_i+p_j}$. Calculations using the above formula with sufficiently large n show that the cost of MTF is 1.386 times optimal on a Zipf distributed i.i.d. source.

In 1979, Bitner introduced the concept of overwork [11]. Overwork is the extra cost of serving requests when the list order has not stabilized. $E(Cost_{MTF}) = 1 + \sum_{i=1}^{n} p_i \sum_{j \neq i} \frac{p_j}{p_i+p_j} + OV_{MTF}$.

In 1985, Competitive Analysis showed MTF is 2-competitive on arbitrary sources [48]. As do the results of Rivest [40] and Bitner, this result uses the simple cost function $c = \rho(l)$ where the cost of a request for l is equal to the position of l within the list L at the time of the request.

Expected Cost from Uniform Distribution to P

We start with a review of Bitner's results [11], analyzing the overwork of MTF as an introduction to the analysis techniques that we will extend later to other settings and algorithms.

Theorem 2 (Theorem 2.1 from [11]) *If each initial list is equally likely, the expected cost of MTF after t requests is*

$$1 + 2 \sum_{1 \leq i < j \leq n} \frac{p_i p_j}{p_i + p_j} + \sum_{1 \leq i < j \leq n} \frac{(p_i - p_j)^2}{2(p_i + p_j)} (1 - p_i - p_j)^t$$

Proof: Given 2 symbols $l_i, l_j \in L, i \neq j$ with associated probabilities p_i, p_j in a list of n symbols that is maintained with MTF, the probability that l_j is in front of l_i at access t can be computed from the following observations:

Probability that $\rho(l_i) < \rho(l_j)$ initially, assuming all distributions are equally likely: $\frac{1}{2}$

Probability of neither l_i or l_j in t requests: $(1 - p_i - p_j)^t$

Probability of one or more l_i or l_j in t requests: $1 - (1 - p_i - p_j)^t$

Probability of l_j given that a request for l_i or l_j has occurred: $\frac{p_j}{p_i+p_j}$

Combining these observations in a straightforward way, we get

$$Prob(\rho(l_j) < \rho(l_i)) = \frac{p_j}{p_i + p_j}(1 - (1 - p_i - p_j)^t) + \frac{1}{2}(1 - p_i - p_j)^t$$

The expected cost of serving a request for l_i is the expected number of symbols in front of l_i, and we use this observation to show that the expected cost of Move-To-Front is

$$\begin{aligned}
E(Cost_{MTF}) &= \sum_{i=1}^{n} p_i E(\rho(l_i)) \\
&= \sum_{i=1}^{n} p_i \left(1 + \sum_{j \neq i} Prob(\rho(l_j) < \rho(l_i))\right) \\
&= 1 + \sum_{i=1}^{n} p_i \sum_{j \neq i} Prob(\rho(l_j) < \rho(l_i)) \\
&= 1 + \sum_{i=1}^{n} \sum_{j \neq i} p_i Prob(\rho(l_j) < \rho(l_i)) \\
&= 1 + \sum_{i=1}^{n} \sum_{j \neq i} \left(\frac{p_i p_j}{p_i + p_j}(1 - (1 - p_i - p_j)^t) + \frac{p_i}{2}(1 - p_i - p_j)^t\right) \\
&= 1 + \sum_{i=1}^{n} \sum_{j \neq i} \frac{p_i p_j}{p_i + p_j} + \sum_{i=1}^{n} \sum_{j \neq i} \left(\frac{p_i}{2} - \frac{p_i p_j}{p_i + p_j}\right)(1 - p_i - p_j)^t \\
&= 1 + 2 \sum_{1 \leq i < j \leq n} \frac{p_i p_j}{p_i + p_j} + \sum_{i=1}^{n} \sum_{j \neq i} \frac{p_i(p_i + p_j) - 2p_i p_j}{2(p_i + p_j)}(1 - p_i - p_j)^t \\
&= 1 + 2 \sum_{1 \leq i < j \leq n} \frac{p_i p_j}{p_i + p_j} + \sum_{i=1}^{n} \sum_{j \neq i} \frac{p_i^2 - p_i p_j}{2(p_i + p_j)}(1 - p_i - p_j)^t \\
&= 1 + 2 \sum_{1 \leq i < j \leq n} \frac{p_i p_j}{p_i + p_j} + \sum_{1 \leq i < j \leq n} \frac{(p_i - p_j)^2}{2(p_i + p_j)}(1 - p_i - p_j)^t,
\end{aligned}$$

which is the value claimed in the theorem statement. ▮

The steady-state cost and overwork can taken directly from the expected cost in the last theorem (the parts that do not and do depend on t, respectively). In other words, we write

$$E(Cost_{MTF}) = 1 + 2 \sum_{1 \leq i < j \leq n} \frac{p_i p_j}{p_i + p_j} + OV_{MTF},$$

where the overwork, OV_{MTF} is

$$OV_{MTF} = \sum_{1 \leq i < j \leq n} \frac{(p_i - p_j)^2}{2(p_i + p_j)} (1 - p_i - p_j)^t.$$

This is the result Bitner obtained.

Expected Cost from Distribution Q to P

In this section we generalize the results of Bitner to an arbitrary prior distribution, rather than the uniform case that he studied. Instead of starting the list in an arbitrary order, let the list be conditioned by the same algorithm on a prior i.i.d. probability distribution Q. An assumption made throughout is that enough symbols have been drawn from Q that the overwork from distributions preceding Q is negligible. A justification for not considering these hypothetical preceding distributions further is that they may not matter because Q can be the actual probabilities (generated from the probabilities and switch order of several distributions) at the instant of the switch to P, rather than the probabilities of the last distribution used to generate the symbols.

Theorem 3 *Let MTF condition list L with probability distribution Q. Then, t requests after the switch to probability distribution P, the expected cost of MTF is*

$$E(Cost_{MTF}) = 1 + 2 \sum_{1 \leq i < j \leq n} \frac{p_i p_j}{p_i + p_j} + OV_{MTF}$$

and the overwork is

$$OV_{MTF} = \sum_{1 \le i < j \le n} \frac{(q_j p_i - q_i p_j)(p_i - p_j)}{(q_i + q_j)(p_i + p_j)} (1 - p_i - p_j)^t$$

Proof: Combining the same observations made in the proof of the uniform to distribution P cost, we obtain

$$\begin{aligned}
&E(Cost_{MTF}) \\
&= 1 + 2 \sum_{1 \le i < j \le n} \frac{p_i p_j}{p_i + p_j} + \sum_{i=1}^{n} \sum_{j \ne i} \left(\frac{q_j p_i}{q_i + q_j} - \frac{p_i p_j}{p_i + p_j} \right) (1 - p_i - p_j)^t \\
&= 1 + 2 \sum_{1 \le i < j \le n} \frac{p_i p_j}{p_i + p_j} + \sum_{i=1}^{n} \sum_{j \ne i} \frac{q_j p_i (p_i + p_j) - p_i p_j (q_i + q_j)}{(q_i + q_j)(p_i + p_j)} (1 - p_i - p_j)^t \\
&= 1 + 2 \sum_{1 \le i < j \le n} \frac{p_i p_j}{p_i + p_j} + \sum_{i=1}^{n} \sum_{j \ne i} \frac{q_j p_i^2 + q_j p_i p_j - p_i p_j q_i - p_i p_j q_j}{(q_i + q_j)(p_i + p_j)} (1 - p_i - p_j)^t \\
&= 1 + 2 \sum_{1 \le i < j \le n} \frac{p_i p_j}{p_i + p_j} + \sum_{i=1}^{n} \sum_{j \ne i} \frac{q_j p_i^2 - p_i p_j q_i}{(q_i + q_j)(p_i + p_j)} (1 - p_i - p_j)^t \\
&= 1 + 2 \sum_{1 \le i < j \le n} \frac{p_i p_j}{p_i + p_j} + \sum_{1 \le i < j \le n} \frac{q_j p_i^2 - p_i p_j q_i + q_i p_j^2 - p_i p_j q_j}{(q_i + q_j)(p_i + p_j)} (1 - p_i - p_j)^t \\
&= 1 + 2 \sum_{1 \le i < j \le n} \frac{p_i p_j}{p_i + p_j} + \sum_{1 \le i < j \le n} \frac{(q_j p_i - q_i p_j)(p_i - p_j)}{(q_i + q_j)(p_i + p_j)} (1 - p_i - p_j)^t
\end{aligned}$$

Separate the parts dependent on t from the rest and we obtain the overwork

$$\begin{aligned}
E(Cost_{MTF}) &= 1 + 2 \sum_{1 \le i < j \le n} \frac{p_i p_j}{p_i + p_j} + OV_{MTF} \\
OV_{MTF} &= \sum_{1 \le i < j \le n} \frac{(q_j p_i - q_i p_j)(p_i - p_j)}{(q_i + q_j)(p_i + p_j)} (1 - p_i - p_j)^t
\end{aligned}$$

which completes the proof. ▮

If one sets all the q's to $1/n$, these equations reduce to Bitner's.

2.5.2 Overwork of MTF is usually positive

For the overwork to be negative, the list must start "closer" to the optimum order. The expected cost of the encoding the initial list must be less than the expected cost of encoding the list as t goes to infinity. That this can happen is easily demonstrated by starting a list in the optimum order for a distribution with high entropy (the probabilities of all the symbols are nearly equal) and then performing MTF on the sequence. The list might be in such an order if the previous distribution had the same ordering of probabilities but with lower entropy. For example, a switch from a probability distribution on 3 symbols a, b, c at probability 0.9, 0.09, and 0.01 respectively to 0.6, 0.24, and 0.16 respectively will produce negative overwork. The list is more likely to be in the correct order for the second distribution after processing requests drawn according to the first than after processing requests drawn according to the second.

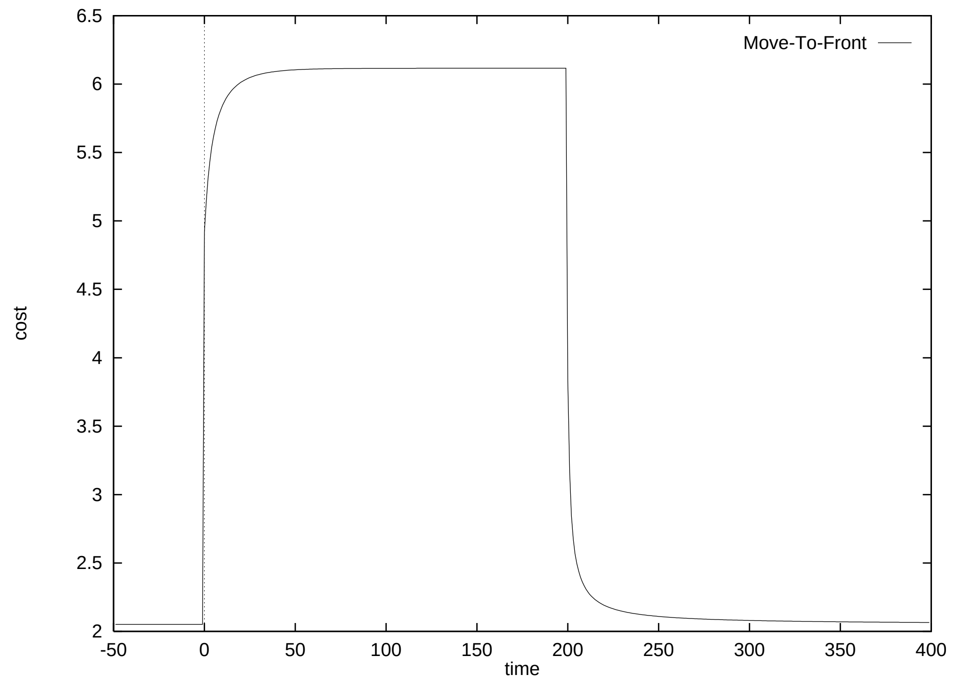


Figure 2.2: Negative overwork with alphabet of 16 symbols. Q is a geometric distribution 0.6, 0.24, 0.096, ... and P is a Zipf distribution 0.296, 0.148, 0.099, ... Switch from Q to P at time 0, and back to Q at time 200.

We show that overwork is usually positive with a few lemmas. We show that if overwork for a pair of symbols is negative in one direction, then it is positive in the other. We also show that overwork for a pair can be positive in both directions.

Lemma 1 *If the overwork from Q to P for a pair of symbols l_i and l_j is negative, then overwork from P to Q for that pair is positive.*

Proof: W.l.o.g., assume that $p_i \geq p_j$. The overwork is zero when $p_i = p_j$ so we concentrate below on $p_i > p_j$.

The overwork from Q to P for a pair of symbols l_i and l_j is $(p_iq_j - p_jq_i)(p_i - p_j)$ times the non-negative quantity $\frac{(1-p_i-p_j)^t}{(p_i+p_j)(q_i+q_j)}$. Negative overwork therefore means $(p_iq_j - p_jq_i)(p_i - p_j) < 0$. Since $p_i > p_j$, this reduces to $p_iq_j - p_jq_i < 0$. The part of overwork from P to Q that can be negative is $(q_ip_j - q_jp_i)(q_i - q_j)$. We show that if $p_iq_j - p_jq_i < 0$, each of $q_ip_j - q_jp_i$ and $q_i - q_j$ is positive, and so the product is positive.

For both parts of this proof, we start with the fact that $p_iq_j - p_jq_i < 0$ because overwork for l_i and l_j from Q to P is negative.

1. Show that $q_i - q_j > 0$.

$$\begin{aligned} 0 &> p_iq_j - p_jq_i \\ &= p_jq_j\left(\frac{p_i}{p_j} - \frac{q_i}{q_j}\right) \\ &> p_jq_j\left(1 - \frac{q_i}{q_j}\right) \qquad \text{since } \tfrac{p_i}{p_j} > 1 \\ &= p_j(q_j - q_i) \end{aligned}$$

Therefore $q_i - q_j > 0$.

2. $q_ip_j - q_jp_i > 0$ is obvious from a simple rearranging of terms, since $q_ip_j - q_jp_i = -(p_iq_j - p_jq_i)$.

Therefore $(q_ip_j - q_jp_i)(q_i - q_j) > 0$. ∎

Lemma 2 *If $p_i > p_j$, the overwork for 2 symbols l_i and l_j from Q to P is negative iff $q_i > \frac{p_i}{p_j} q_j$.*

Proof: Overwork is negative iff $\frac{(q_j p_i - q_i p_j)(p_i - p_j)}{(q_i + q_j)(p_i + p_j)} (1 - p_i - p_j)^t < 0$.

Since probabilities are non-negative and $p_i > p_j$ by conditions of the lemma, OV_{MTF} is negative iff $q_j p_i - q_i p_j < 0$. Therefore $OV_{MTF} < 0$ iff $q_i p_j > q_j p_i$, which is equvalent to $q_i > \frac{p_i}{p_j} q_j$. ▮

Lemma 3 *If $p_i > p_j$ and $q_i \leq q_j$ then the overwork for a pair of symbols l_i and l_j from Q to P and from P to Q is non-negative.*

Proof: As in previous lemmas, overwork for l_i and l_j from P to Q is negative only if $(q_i p_j - q_j p_i)(q_i - q_j) < 0$. Since $q_i \leq q_j$, $(q_i - q_j) \leq 0$. Since $q_i \leq q_j$ and $p_j \leq p_i$, $(q_i p_j - q_j p_i) \leq 0$. Therefore $(q_i p_j - q_j p_i)(q_i - q_j) \geq 0$. Furthermore, Lemma 2 directly implies that $(p_i q_j - p_j q_i)(p_i - p_j) \geq 0$. ▮

Theorem 4 *When switching from one probability distribution Q to another P, where P and Q are randomly chosen probability distributions and $P \neq Q$, overwork of MTF for any pair of symbols l_i and l_j is non-negative with probability at least 0.75.*

Proof: As before, assume w.l.o.g. that $p_i \geq p_j$.

First, if $p_i = p_j$, the overwork is zero. If the symbols are equally likely, it does not matter what order they are in.

This leaves the case $p_i > p_j$. If the overwork from Q to P is negative, then $q_i > \frac{p_i}{p_j} q_j$, by Lemma 2. Therefore negative overwork implies $q_i > q_j$ because $\frac{p_i}{p_j} > 1$. If all pairs of distributions Q and P are equally likely, $q_i \leq q_j$ represents 0.5 of the occurrences, in which the overwork is non-negative in both directions, by Lemma 3.

In the remaining 0.5, where $q_i > q_j$, one of the overworks from Q to P or from P to Q must be positive, by Lemma 1. The probability is at least 0.5 that the overwork from Q to P is positive, and at least 0.5 that the overwork from P to Q is positive. Thus, for MTF, the overwork of adapting from a distribution Q to a new distribution P for a pair of symbols l_i and l_j is non-negative with probability at least 0.75. ▮

2.5.3 Less overwork from smaller changes

We attempt to show that a switch to a "closer" distribution will, in general, produce less overwork. We show this for the MTF algorithm and conjecture that a probability distribution switch that is closer and less overwork for MTF will be similarly beneficial for all reasonable on-line algorithms. We concern ourselves with positive overwork only. Although overwork can be negative, it is more likely to be positive as was shown in the previous section.

We will define a probability distribution D that lies "between" Q and P. We will show that if MTF is adapted to Q, there will be less overwork in a switch to D than in a switch to P.

Theorem 5 *Let P, Q, and D be probability distributions and $0 \leq x \leq 1$ such that $D = xP + (1-x)Q$. Let $OV_{MTF}(Q,t)$ be the overwork of MTF at time t on some data assumed to be distributed according to Q. If $OV_{MTF}(Q,t) = 0$ and $OV_{MTF}(P,t) \geq 0$, then $OV_{MTF}(P,t) \geq OV_{MTF}(D,t)$.*

Proof: Compare the overwork from Q to P to the overwork from D to P for one pair of symbols l_i and l_j.

$$\begin{aligned}
0 &\leq \frac{(q_jp_i - q_ip_j)(p_i - p_j)}{(q_i+q_j)(p_i+p_j)}(1-p_i-p_j)^t - \frac{(d_jp_i - d_ip_j)(p_i - p_j)}{(d_i+d_j)(p_i+p_j)}(1-p_i-p_j)^t \\
&\leq \frac{(q_jp_i - q_ip_j)(p_i - p_j)}{(q_i+q_j)(p_i+p_j)} - \frac{(d_jp_i - d_ip_j)(p_i - p_j)}{(d_i+d_j)(p_i+p_j)} \\
&= \frac{p_i - p_j}{(d_i+d_j)(q_i+q_j)(p_i+p_j)}((q_jp_i - q_ip_j)(d_i+d_j) - (d_jp_i - d_ip_j)(q_i+q_j)) \\
&= \frac{p_i - p_j}{(d_i+d_j)(q_i+q_j)(p_i+p_j)}(d_iq_jp_i - d_jq_ip_j - d_jq_ip_i + d_iq_jp_j) \\
&= \frac{p_i - p_j}{(d_i+d_j)(q_i+q_j)(p_i+p_j)}(d_iq_j - d_jq_i)(p_i - p_j) \\
\Rightarrow 0 &\leq d_iq_j - d_jq_i
\end{aligned}$$

Substitute $x(P-Q)+Q$ for D

$$\begin{aligned}
0 &\leq (x(p_i - q_i) + q_i)q_j - (x(p_j - q_j) + q_j)q_i \\
&= x(p_i - q_i)q_j + q_iq_j - x(p_j - q_j)q_i - q_jq_i \\
&= xp_iq_j - xq_iq_j + q_iq_j - xp_jq_i + xq_jq_i - q_jq_i \\
&= xp_iq_j - xp_jq_i \\
&= x(p_iq_j - p_jq_i)
\end{aligned}$$

Sum up all pairs to obtain difference in total overwork

$$x \sum_{1 \leq i < j \leq n} (p_iq_j - p_jq_i) \geq 0$$

As x goes to 1, the distance from D to P goes to 0 and the difference in the overwork between Q to P and Q to D increases. Thus, as D gets closer to P, the overwork from Q to D increases. ∎

2.6 Summary

This chapter defined terms used throughout this work and gave a description of the Burrows-Wheeler Transform. We discussed ways of characterizing BWT output, then pointed out that BWT output is like piecewise independent identically distributed (p.i.i.d.) data. We also discussed the analysis techniques Competitive Analysis and overwork, the cost of adapting to a change. We analysed the asymptotic cost and overwork of the Move-To-Front algorithm on p.i.i.d. data, showing that overwork is usually positive for any pair of symbols, and that overwork is proportional to the "distance" between the distributions of the pieces.

CHAPTER 3

Preprocessing for Burrows-Wheeler Compression

3.1 Introduction

In this chapter we talk about some of the ways in which data can be modified before applying Burrows-Wheeler Compression. We refer to these modifications as "preprocessing"; another term is "filtering", which suggests lossiness. The goal of most of the preprocessing is to increase the amount of compression. But preprocessing can also increase the speed (sometimes at the cost of a small loss in the compression ratio) by, for instance, reducing the amount of data to process.

One description of the Burrows-Wheeler Transform (BWT) is that it takes a file of n bytes and creates n permutations of the data by moving the first 1 to n bytes of the data to the end of the remaining data, in effect rotating the data. The n strings of n bytes each are then sorted, which groups similar contexts together, and the last byte of each string is the output data. Since similar contexts are grouped together, this sequence of "last characters" is highly compressible. To compress the data, the output of the transform is run through a Move-To-Front encoder, and the output of that is compressed with arithmetic coding [13].

The choice of Move-To-Front (MTF) coding is important. While contexts are grouped together, there is no per-context statistical information kept, and so the encoder must rapidly adapt from the distribution of one context to the distribution of the next context. The MTF coder has precisely this rapidly adapting quality.

The order of sorting determines which contexts are close to each other in the resulting output, and so the sort order (including the ordering of the source alphabet) can be important in BWT-based compression. We are not aware of any investigation prior to our original DCC presentation of this work, and many people seem to consider sorting a fixed part of the algorithm. For example, in an extensive study of BWT-based compression [21], Fenwick states that "anything other than a standard sort upsets the detailed ordering and prevents recovery of the data" — however, this is not entirely true, as *any* reversible transformation (such as a modified sorting order) can be used for this first phase.

Even with the rapid adaptability of the MTF coder, placing contexts with similar probability distributions close together reduces the cost of switching from one context to another (as justified theoretically in Theorem 5), resulting in reduction in the final compressed size. This dependence on input alphabet encoding is a characteristic that is fairly unique among general-purpose compression schemes. Previous techniques, including statistical techniques (such as the PPM algorithms) and dictionary techniques (represented by LZ77, LZ78, and their descendants), are largely based on pattern matching which is entirely independent of the encoding used for the source alphabet.

It is easy to test a compression algorithm's dependence on alphabet ordering — simply run the source through a randomly chosen alphabet permutation and see how subsequent compression is affected. Using readily available programs `gzip` (version 1.2.4) as a representative of LZ77-based compression and `bzip` (version 0.21) as a representative of BWT-based compression, the following results were obtained in performing this simple experiment. The input file is a 24 bit color version of the popular image of Lena (original size 786,488 bytes).

	Order	
Algorithm	Original	Random
BWT	586,783	671,612
LZ77	730,980	731,170

Note that the random alphabet reordering has very little effect on the dictionary technique, but makes a huge difference to the BWT-based algorithm. Clearly, using an arbitrarily chosen order can have a significant negative impact on the size of the compressed output. In the case of images, like in this test, the natural intensity-level encoding is not arbitrary and is quite natural (and is taken advantage of by image coders such as DPMC), and so it seems unlikely that a modified sorting order would make a significant improvement. However, for text files and other files where the input coding is initially quite arbitrary, it is reasonable to ask whether reordering the sorting stage can produce better compression results. Our experiments show that finding other orders that improve the compression by small amounts is not difficult.

3.1.1 Previous Work

The paper of Burrows and Wheeler describing the BWT [13] is only 7 years old, and yet the technique has captured the interest of people in both the research community [21] and the popular computer press [38]. There are at least two publicly released programs based on this technique: `bzip`, an implementation by Julian Seward that includes coding improvements suggested by Fenwick [21], and `szip`, an implementation by Michael Schindler that concentrates on speed improvements.

Attempted improvements on the original algorithm (BWT followed by MTF coding) have shown very limited success. The most prominent improvements come from the extensive study done by Peter Fenwick [21] in which some improvements were made to the final coding stage of the Burrows-Wheeler algorithm. The work presented in this chapter also provides modest improvements in the compression performance, but we focus on the initial sorting phase of the Burrows-Wheeler algorithm. We are not aware of any prior published research into this particular aspect.

3.2 Improving the Sort Order

When the data rotations are ordered in the BWT, the sorting is done in standard lexicographical order[1]. If the initial alphabet encoding is assigned in an arbitrary manner (such as ASCII encoding or opcode encoding in an object file), then the resulting sorted order is also fairly arbitrary. As demonstrated above, the ordering of the characters can make a significant difference in the size of the BWT compressed output. An improvement to the ordering could center on finding a better arbitrary ordering, or perhaps by having a small library of 4 or so orderings and picking the best one based upon some test or user choice. The speed of the algorithm would be preserved and the ordering could be saved in a few bits.

[1] In this paper, "lexicographical order" always means the standard lexicographical ordering using the *original* alphabet encoding.

Another way to improve on the order would be to spend time analyzing the data to determine the best ordering. The order would then need to be saved with the data which would take about 211 bytes (actually $\lceil \log_2(256!) \rceil$ bits). The 211 byte overhead may be mitigated somewhat by using that order as an initial ordering for the MTF coder. This idea could in fact be applied to representative data from some large class (such as English text), and the resulting order could be one of the small number of available fixed orderings as described previously. This provides the reordering benefit to commonly encountered classes of data, and yet avoids the overhead of initially selecting the order for each compressed file. Our experiments show that this is indeed useful for classes such as English text.

These ideas are explored in detail in the next two sections.

3.2.1 Heuristic Orders

Our first attempt in reordering the input alphabet was simply hand-selecting orderings that seemed to make sense to us heuristically. One of the best heuristic orderings was the one that grouped the vowels together, but kept capital and lower case letters separate as in ASCII. Other seemingly sensible heuristic orderings, such as grouping capitals with their lower case equivalent: "AaBbCc...", did not perform as well. Ordering by frequency of occurrence, which has the advantage of no overhead since the decoder can determine the correct ordering from the data, turned out to be one of the worst orderings.

Much uncompressed data is English or some other human language (text data). A heuristic order optimized for text would be useful. In fact, the alphabetical order of ASCII is not the best order. Experiments show that an order which groups similar symbols near each other gets a small but noticeable (0.25% to 0.5%) improvement over the ASCII ordering. Symbols that are similar are what one would intuitively expect to be similar. The best heuristic orderings tried on text group vowels, similar consonants, and punctuation together. One hand coded ordering, "AEIOUBCDGFHRLSMNPQJKTWVXYZ" plus a few punctuation groupings ("?!" and "+-,."), does well on text, and, since it is close to the original ordering, it does

File	Original	"aeioubcdgf..."
bib	27,097	26,989
book1	230,247	229,558
book2	155,944	155,515
geo	57,358	57,369
news	118,112	117,734
obj1	10,409	10,402
obj2	76,017	76,062
paper1	16,360	16,221
paper2	24,826	24,705
pic	49,422	49,427
progc	12,379	12,331
progl	15,387	15,304
progp	10,533	10,503
trans	17,561	17,514
total	821,652	819,634
lena	586,783	589,913
lesms10	923,850	920,558

Table 3.1: Performance of hand-selected heuristic alphabet reordering.

not usually perform much worse on non-text data than the original order. Table 3.1 shows the performance of the selected ordering on files from the Calgary Compression Corpus, as well as on the 24-bit version of `lena` and the very large (3,334,517 byte) text file of Les Misérables (`lesm10`), obtained from Project Gutenberg [53]. For comparison, an order computed (with methods discussed in the next section and summarized in Table 3.3) from text data outside the Calgary Compression Corpus improved compression similarly for the text files of the corpus, but suffered a worse penalty for the non-text files.

3.2.2 Computed Orders

A good way to analyze the data to determine the best ordering uses the idea that a pair of different byte values that are likely to be followed (or preceded) by the same set of byte values should be close to each other in an ordering of the data. This reduces the “change in probability distribution” overhead (or overwork) encountered in the encoding phase. For example, when “?” and “!” are encountered, it is often at the end of a sentence. Sentences are very often followed by a space or a line feed/carriage return.

To utilize the above idea, consider an ordering of contexts as a walk through the various contexts. The overwork of going from one context to the next is related to the dissimilarity of the context probability distributions, and we would like to minimize this cost. For reordering the input alphabet, the problem can be viewed as an instance of the traveling salesperson problem (TSP) in which each vertex is a single character context, and the edge costs are computed based on some probability distribution distance measure. We then try to minimize the cost of the TSP tour. Since this is an NP-hard problem [22], we tried various approximation algorithms in order to select an alphabet reordering, and we also tried various distance measures for computing the edge costs.

For all distance measures, start by creating a histogram for each of the n possible characters. Each histogram contains counts of the characters immediately preceding each occurrence in the data of the character represented by that histogram. Characters are “close” to each other if their histograms are “close”. In the first distance measure, the “distance” between two histograms is captured by summing the squares of the differences of the logarithms of each of the 256 counts. The second distance measure uses a standard measure from probability theory and information theory, the Kullback-Leibler distance, or relative entropy [17]. The third and fourth distance measures are based on distance measures used in the algorithms literature when analyzing the move-to-front algorithm [48] (which is the basis for the coding phase of the BWT-based compressor): the histograms are sorted in decreasing order of frequency, and then the number of “inversions” between the two lists are counted. The third distance measure is precisely the number of inversions, and the fourth distance measure is the logarithm of the number of inversions.

For one approximate solution to the resulting TSP, we used a simple approximation algorithm based on minimum spanning trees due to Rosenkrantz, Stearns, and Lewis [41]. For distance measures satisfying the triangle inequality, the tour produced

	Orig order	MST tour	addition	farinsert	multifrag	loss
Orig metric	230,247 230,247 0 26,860	229,561 229,351 210 9,415	229,921 229,710 211 8,824	229,777 229,566 211 8,587	231,210 230,998 212 10,829	229,458 229,247 211 9,300
KL dist	230,247 230,247 0 111.71	230,216 230,007 209 44.73	230,017 229,801 216 39.42	229,868 229,652 216 37.59	230,079 229,867 212 45.91	230,424 230,212 212 37.62
Inv	230,247 230,247 0 274,756	229,712 229,500 212 147,632	229,455 229,244 211 132,702	230,446 230,230 216 131,756	229,496 229,281 215 133704	229,780 229,566 214 131,622
LogInv	230,247 230,247 0 682.1	229,712 229,500 212 613.5	229,569 229,358 211 571.7	229,808 229,597 211 572.2	229,496 229,281 215 571.8	229,832 229,620 212 584.8

Table 3.2: TSP reordering results using file book1. Numbers in each box, from top to bottom, are the total compressed size, the compressed size of just the reordered data, the size of encoding the reordering permutation, and finally the TSP tour length.

by this algorithm is guaranteed to be no worse than twice the optimal tour length, but in our case the distances do not necessarily satisfy the triangle inequality and so there is no guarantee on the performance of the algorithm. We also used several of the approximation algorithms included in the `TspSolve` package distributed by Chat Hurwitz [25] — specifically, we used the addition, farinsert, multifrag, and loss techniques from this package. The results of these tests are summarized in Table 3.2.

Trials indicate that the various TSP algorithms do find orderings that are better (in terms of TSP tour length) than the original alphabet encoding. In this particular text file, the improvement in TSP lengths is roughly reflected in improvements to the compressed output size. Even with the additional overhead of encoding the reordering permutation, the total compressed size is decreased in the better orderings.

For files in the Calgary Compression Corpus, gains were observed for all of the English text files, and substantial gains were obtained for the file `geo`. On the other hand, very little gain was observed for the `pic` file, and reordering resulted in significantly degraded performance for the `obj2` file. In the case of images such as `pic`, the pixel value ordering is very natural and seems to be the best possible.

For non-text files, the weight of the order found by TSP was almost always much less than the weight of the lexicographical order, yet the sizes of the compressed data for each ordering did not always correspond. Clearly, the correlation in such cases is weak. Perhaps reordering contexts with more than one character would result in a more direct correlation.

The largest improvement occurs in files that are not text or true color images but have some non-standard organization for which lexicographical order is not very good. Such files would include 256 color images done with a colormap. Perhaps a measure for determining when to apply TSP would be to compare the compressed size for lexicographical order with the compressed size for a random order. If the two are close, then TSP will likely produce a better order.

Table 3.3 shows the results of using a computed reordering for all the files in the Calgary Compression Corpus. Each file was run through the reordering process using our first distance measure and the farinsert TSP approximation algorithm, and the resulting output size (including the size required to encode the alphabet permutation) is given in the 2nd column. We also performed a test where we took a large amount of English text unrelated to the corpus (obtained from Project Gutenberg) and computed a good general English text ordering from this data. This fixed reordering was then used in encoding all the data files — the results benefit from the fact that you do not have to encode the fixed permutation of the input alphabet with the compressed data. In Table 3.3, the last column shows the best compression achieved on each file, and this value is marked in bold in each row. The most interesting thing about the ordering computed from the Project Gutenberg files is that it is an excellent selector of files written in English. With one exception, all of the files from the Calgary Compression Corpus that performed best under this ordering were in fact the English text files (even though they were not used in computing the ordering). The one exception is the LISP program, which when examined was in fact discovered to have large pieces of English text within it in the form of both comments and function names. The large

	Orig order	TSP (farinsert, 1st metric)	Fixed (text) reorder	Best
bib	27,097	27,199	**26,977**	26,977
book1	230,247	229,777	**229,071**	229,071
book2	155,944	156,077	**155,613**	155,613
geo	57,358	**55,897**	57,565	55,897
news	118,112	118,385	**117,978**	117,978
obj1	**10,409**	10,647	10,511	10,409
obj2	**76,017**	77,450	77,080	76,017
paper1	16,360	16,477	**16,264**	16,264
paper2	24,826	25,132	**24,654**	24,654
pic	**49,422**	49,682	49,518	49,422
progc	**12,379**	12,579	12,427	12,379
progl	15,387	15,571	**15,364**	15,364
progp	**10,533**	10,734	10,568	10,533
trans	**17,561**	17,887	17,663	17,561
Total size	821,652	823,494	821,253	818,139
lena	**586,783**	599,883	600,872	586,783
lesms10	923,850	**920,541**	921,392	920,541

Table 3.3: Computed alphabet reordering for all files in the Calgary Compression Corpus

Les Misérables text file also was not best with this fixed text reordering. For such a large file, a data-dependent ordering saves enough space to more than compensate for the overhead of including the ordering.

It is interesting to compare this table with the results for the hand-selected ordering of the previous section. The performance of the fixed, computed ordering is comparable to that of the hand-selected ordering on the text files. This is encouraging for other arbitrarily encoded input sources, suggesting that we do not have to examine and hand-tune orderings for each input source.

3.3 Other Preprocessing

The simplest preprocessing is run-length coding, which increases compression in 2 ways. Occasionally, runs can be more efficiently compressed before BWT rather than after. But mostly, compressing runs before performing a block sort allows more data to fit inside a fixed-size block. Run-length coding does not destroy the contexts, which the BWT needs. When compressing more data than will fit in a single block, Burrows-Wheeler compression is almost always more efficient as block size increases. `bzip2` uses this method of preprocessing.

A possible way to gain the advantages of increased block sizes without the simple expedient of using more memory for larger blocks, is coding the data with some kind of static Huffman coding. The use of static Huffman leaves the contexts sortable.

More sophisticated are the data dependent techniques. These techniques can help any general purpose compression method. Tailoring them specifically for Burrows-Wheeler compression yields even more gain. On English and other natural languages, significant increases in amount of compression comes through such ideas as tagging words and phrases with parts of speech identifiers, thus allowing the BWT to sort noun from verb rather than just letter from letter [28]. There are many image specific techniques, including such simple ideas as rotating the image 90 degrees, or taking the difference of each pixel with a neighboring pixel, or transforming from red, green, blue (RGB) colors to luminance, hue, and saturation values. More complex are the discrete cosine transform (DCT), used in JPEG [26], and wavelets [23].

3.4 Summary

Some of the preprocessing ideas require minor changes to the parameters of the Burrows-Wheeler Transform, such as bit-based BWT rather than byte based for compressing data that has been preprocessed with standard static Huffman coding. Alternatively, Huffman coding could be performed with bytes rather than bits. Variations on the BWT are covered in the next chapter.

Preprocessing is not independent of the data compression technique it aids, as is seen in the alphabet reordering. Dictionary methods are not significantly affected by the order of the alphabet, while Burrows-Wheeler can be. Other preprocessing methods work with more than Burrows-Wheeler based algorithms, but can be tailored to better fit the particular compression algorithm used.

	preprocessing			preprocessing	
	yes	no		yes	no
bib	26726	26870	barb	194759	194759
book1	205537	215353	bird	30963	30963
book2	143199	148049	boat	165914	165915
geo	52333	57740	bridge	51773	51777
news	111512	113016	camera	39460	39460
obj1	10668	10671	circles	1058	777
obj2	75632	75632	clegg	489938	889977
paper1	15636	15993	crosses	1052	997
paper2	23196	24077	france	15972	13178
pic	30158	46008	frog	131575	121163
progc	12075	12244	frymire	195602	190851
progl	14861	15160	goldhil1	47580	47580
progp	10720	10525	goldhil2	173923	173923
			horiz	212	226
bible.txt	729468	737387	lena1	46772	46772
cp.html	7723	7723	lena2	166149	166149
e.coli	1144042	1144042	lena3	513065	575445
kennedy.xls	25347	111812	library	88480	85394
sum	12690	12689	mandrill	209397	209397
world192.txt	417476	414948	monarch	642463	665097
xargs.1	1769	1787	montage	27681	27681
			mountain	184090	184084
h.influenza	444863	444863	peppers2	165852	165852
lena.bmp	526207	580755	peppers3	436420	565707
lesms10	748266	783536	sail	733767	733767
			serrano	81686	81686
			slope	13098	13098
			squares	211	196
			text	1544	1465
			trans	17811	17261
			tulips	727457	806005
			washsat	71552	71552
			zelda	159123	159123

Table 3.4: Value of Preprocessing: DC 1.24 on files from various corpi

Since real data seldom fits the simpler model of stationary and ergotic, transforming the data before applying a universal compression algorithm can give dramatic improvements in the amount of compression obtained. Temptations to dismiss preprocessing techniques as “cheating”, or unworthy of mention because most are not general to all data, should be curbed. Otherwise, one may as well ignore all the methods designed specifically for image or sound data, such as PNG. The more one knows about the data to be compressed, the more one can compress it.

CHAPTER 4

The Burrows-Wheeler Transform and Variations

4.1 Introduction

The Burrows-Wheeler Transform (BWT) is described in chapter 2 and again briefly at the start of chapter 3. In this chapter, we explore the BWT in more detail. We explain why BWT is fast and why it achieves high compression. We examine some not so trivial modifications to the BWT.

The key to the Burrows-Wheeler Transform is sorting, a well understood and thoroughly researched area of Computer Science. Sorting groups substrings with similar contexts close to each other, resulting in long runs of similar symbols in the output column. This output is easy to compress. A nice feature is the ability to handle contexts of unlimited length. Some versions of PPM cannot handle unlimited length contexts.

The speed of the transform is the speed with which the data can be sorted. It is well known that $\Omega(m \log m)$ is the lower bound time complexity for comparison based sorting of m items, where each comparison can be performed in $O(1)$ time. But in a variety of special cases, sorting can be performed in $O(m)$ time. The sorting required in the Burrows-Wheeler Transform on a block of size m is, in theory, such a special case because the strings to be sorted are all suffixes of the input data.

Practical algorithms for performing the sort have a running time greater than $O(m)$. Running times and memory requirements differ, and can depend on the data. Among the algorithms used to sort the suffixes are the original one presented in BW94, the suffix array method of Sadakane [42], and the Forward Radix Sort of Andersson and Nilsson [3]. The Bentley-Sedgewick algorithm [10], used in the original Burrows-Wheeler paper, completely sorts all suffixes of a particular context, that is, all suffixes beginning with the same small set of symbols, before moving on to other suffixes. Sorting of further contexts is accelerated by referring back to already sorted contexts where possible. The suffix array algorithm of Sadakane sorts all suffixes out to k

characters. The speed of the method comes from the fact that k can be doubled in each pass. Analysis of those two methods [45] and others [42, 30, 3] show that the time complexity of the various methods depends upon the data being sorted. Let A be the average match length, which is the average number of symbols that must be compared to find a difference between two adjacent sorted suffixes. Then the original sort method has time complexity of $O(Am \log m)$. Sadakane's suffix array method has time complexity of $O(m \log m)$ but with a higher constant than Bentley-Sedgewick. The Forward Radix Sort is $O(m \log A)$, which is the fastest, but takes more memory. To get the best of the original and Sadakane's methods, later versions of Seward's `bzip2` program, an implementation of Burrows-Wheeler compression, start with the Bentley-Sedgewick algorithm, then start over with Sadakane's if running estimates of A are too large.

The inverse BWT is easily done in linear time. Decompression is faster than compression. For many applications, decompression speed is more important than compression speed. For instance, decompression is performed each time compressed software is installed, and compression is done only once when the software is created. Data intended for broadcast, if compressed, will be compressed once and decompressed many times. Another instance is in compressed hard drives (Stacker, and `MS-DOS` `dblspace` and `drvspace` are some examples, also, so called executable compressors such as `pklite` and, on the Unix side, `gzexe`); hard drive accesses are more often reads than writes. On the other hand, data backup systems, video security systems, and other systems in which information that may not be needed is temporarily stored, and space probes, where resources available for compression of Earth bound data are extremely limited, are applications where speed and simplicity of compression are more important than decompression.

Memory requirements are linear in m which is a significant amount of memory on machines of the 1990's. The dictionary based algorithms can be implemented with a fairly small amount of memory for the dictionary. In environments where memory is very limited, Burrows-Wheeler compression may not be suitable.

The BWT is not well suited for on-line applications. Before the transform or inverse may be performed, all the data in a given block must be available. The block size can be reduced, of course, but this costs in compression efficiency.

Schindler created a limited context version of the Burrows-Wheeler Transform [43]. Each suffix is sorted out to some user specified (or hardcoded) k places. Ties are resolved with the unique position (t) of each suffix in the original string. As might be expected, in most cases compression improves as k increases. For most data therefore, the limited context transform does not compress as well. One exception is the concatenated files of the Calgary Corpus, where the position in the original string is significant because the data changes considerably from individual file to file. Limited context sorting is faster (replace the A value of the BWT algorithm times with $min(A, k)$), so the transform is faster, however, the inverse transform is slower.

Another variation, introduced here but not explored in detail, is "interleaved" context-position sort. Since data is usually not stationary, its approximate position within a file can be a better predictor than the kth symbol of its context. A sort can be done using the first x symbols of context followed by the most significant part of the position, then followed by more context and the next most significant part of the position, etc. Thus, limited context sort is just one way in which context and position can be mixed.

In the Roman alphabet based ASCII world, people are so accustomed to 8 bit codes that the existence of others is easily overlooked. The Burrows-Wheeler Transform can be adapted to any desired alphabet size, including 16 bit codes (Unicode) used for international alphabets. Alphabets of variable length symbols are also possible and some experiments have been performed with word based BWT on natural languages. Because the BWT sorts infinitely long contexts, it performs reasonably well on 2 symbol alphabets. A bit-based BWT can be performed on data encoded with a static Huffman code for reasonable results.

4.2 Gray Code Sort

Sorting is typically done in lexicographic order using a standard character encoding such as ASCII. For example, the phrases "ayz, aza, azz, baa, baz, bba, bbz, bza, bzz, caa, caz" are sorted in order. But typical sorting is not always the best way to organize the data for BWT, as has already been demonstrated. If the sorting method is changed to put similar strings closer to each other, an improvement of around 0.5% can be achieved, including an additional 0.5% when combined with one of the better orders discussed in the previous chapter. The binary reflected Gray code minimizes changes in bits between adjacent binary codes. The sorting of strings can be done in an analogous fashion.

Sorting is changed by inverting the sort order for alternating character positions. Let the jth column of the data in the $n \times n$ BWT matrix be the jth character of all the strings created by rotating the data. The first column is sorted as before. But all following columns will be sorted in forward and backward orders according to data from previous columns. Specifically, whenever a character in a column changes between string i and string $i + 1$, the sort order of all following columns is inverted. Only the leftmost change (the lowest column) is considered if more than one column changes. Sorting in this fashion will produce an ordered list reflected in a way analogous to the binary reflected Gray codes. Below is an example, using lexicographical order on the phrases given earlier.

		Inversion Point For Columns	
Normal	Reflected	2	3
ayz	ayz		√
aza	azz		
azz	aza	√	√
baa	bza		
baz	bzz		√
bba	bbz		
bbz	bba		√
bza	baa		
bzz	baz	√	√
caa	caz		
caz	caa		

File	Number of columns sorted in reflected order					
	0	1	2	4	8	max
bib	27,097	27,053	27,032	27,030	27,031	27,051
book1	230,247	230,158	230,130	230,134	230,069	230,074
book2	155,944	155,885	155,809	155,708	155,663	155,666
geo	57,358	56,924	56,850	56,850	56,855	56,859
news	118,112	117,996	117,944	117,899	117,932	117,897
obj1	10,409	10,388	10,384	10,382	10,381	10,381
obj2	76,017	75,978	75,922	75,851	75,817	75,829
paper1	16,360	16,347	16,350	16,347	16,334	16,341
paper2	24,826	24,808	24,819	24,810	24,818	24,828
pic	49,422	49,415	49,405	49,420	49,407	49,406
progc	12,379	12,364	12,357	12,348	12,341	12,340
progl	15,387	15,378	15,357	15,366	15,380	15,355
progp	10,533	10,529	10,536	10,523	10,529	10,527
trans	17,561	17,519	17,525	17,518	17,484	17,488
total	821,652	820,742	820,420	820,186	820,041	820,042
lena	586,783	584,060	582,742	582,657	582,664	582,664
lesms10	923,850	923,608	923,472	923,282	922,975	923,000

Table 4.1: The result of ordering data based on the reflected ordering described in Section 4.2.

Notice how the 2nd and 3rd character positions are more homogeneous in the reflected ordering. This greater homogeneity also exists in the last column of the BWT, which is the column used as input to the MTF coder. Unlike the simple alphabet reordering, reflection improved the compression of the BWT algorithm on every file tested.

One does not need to extend the reflection of sorting to all n columns to gain improvement in the compression size. As one might expect, experiments show that the greatest improvement occurred when reflection was done on the 2nd column. Reflecting the 2nd and 3rd columns further improved compression and reflecting the 2nd through jth columns where j ranged from 4 up to several hundred usually reaped further improvements, decreasing as j increased until very little change occurred for $j > 8$. This is summarized in Table 4.1.

file	aeiou bcdgf	TSP (MST, 1st metric)	TSP (farinsert, 1st metric)	Fixed (text) reorder
bib	26,960	27,406	27,160	26,935
book1	229,391	229,452	229,623	228,938
book2	155,207	155,809	155,957	155,399
geo	56,887	56,081	55,297	57,041
news	117,557	118,109	118,331	117,913
obj1	10,383	10,801	10,639	10,483
obj2	75,818	77,693	77,181	76,922
paper1	16,237	16,503	16,466	16,257
paper2	24,728	25,167	25,166	24,622
pic	49,390	49,011	49,594	49,531
progc	12,300	12,653	12,582	12,419
progl	15,283	15,684	15,555	15,304
progp	10,489	10,805	10,758	10,566
trans	17,463	17,793	17,837	17,631
total	818,093	822,967	822,146	819,961
lena	585,754	598,230	595,720	597,031
lesms10	920,352	919,424	920,067	921,149

Table 4.2: Combining reflected-order sorting with alphabet reordering.

The reflected ordering can of course be combined with the alphabet reordering as described in the preceding chapter. Table 4.2 shows the results of combining reflection (using maximum number of columns) with some of the sort orders from previous chapter.

4.3 Best overall results

Table 4.3 takes the best compression results from previous tables and shows the methods used on each file to achieve the result. Since much of the corpus is text, the hand coded order “aeiou” is best for most of the files. The TSP schemes did better than ASCII order on text files, but usually not as well as “aeiou”. On `geo`, the one non-text file for which lexicographical order was bad, all the TSP schemes found much better orders. For all files (except paper1 when reordered with “aeiou”),

File	Original BZIP	Our Best Size	method
bib	27,097	26,935	Fixed text + reflect
book1	230,247	228,938	Fixed text + reflect
book2	155,944	155,207	aeiou + reflect
geo	57,358	55,297	TSP(farinsert) + reflect
news	118,112	117,557	aeiou + reflect
obj1	10,409	10,381	reflect
obj2	76,017	75,818	aeiou + reflect
paper1	16,360	16,221	aeiou
paper2	24,826	24,622	Fixed text + reflect
pic	49,422	49,011	TSP(MST) + reflect
progc	12,379	12,300	aeiou + reflect
progl	15,387	15,283	aeiou + reflect
progp	10,533	10,489	aeiou + reflect
trans	17,561	17,463	aeiou + reflect
total	821,652	815,522	
lena	586,783	582,664	reflect
lesms10	923,850	919,424	TSP(MST) + reflect

Table 4.3: Overall best results

reflected-order sorting improved compression.

The savings due to the combination of the two techniques of this and the previous chapter (alphabet reordering and reflected order sorting) is actually greater than the sum of the savings due to the individual techniques. Considering the individual pieces of the compression algorithm, we estimate that the alphabet reordering is responsible for approximately 67% of the savings, the reflected sorting is responsible for approximately 26% of the savings, and the remaining 7% of savings comes from the combination of the two techniques.

Gains on image data can be as much as 1.5%. Tests of `bzip2` and `bzip2r`, a version modified for Gray code sort, on the Waterloo corpus yielded the usual 0.25% gain on most files but with up to 1.5% on several and in one case a loss of 12%.

		bzip2	bzip2r
barb	262274	200723	200268
bird	65666	33646	33544
boat	262274	174829	174509
bridge	65666	54193	54196
camera	65666	42051	41991
circles	65666	1008	997
clegg	2149096	910813	911696
crosses	65666	1137	1121
france	333442	16334	16382
frog	309388	133849	133776
frymire	3706306	205429	207574
goldhill1	65666	50185	50132
goldhill2	262274	183516	183294
horiz	65666	179	195
lena1	65666	48668	48593
lena2	262274	173865	173323
lena3	786568	584357	575918
library	163458	91482	91360
mandrill	262274	216484	216246
monarch	1179784	686317	676798
montage	65666	28826	28719
mountain	307330	197096	197095
peppers2	262274	174664	174343
peppers3	786568	567303	557424
sail	1179784	762464	750626
serrano	1498414	87066	98098
slope	65666	13527	13407
squares	65666	171	185
text	65666	1671	1659
tulips	1179784	837698	826471
washsat	262274	82762	82607
zelda	262274	168020	167733
total	16466106	6730333	6690280

Table 4.4: Standard BWT and Gray code sort BWT on image data from the Waterloo Corpus

4.4 Adding Gray Code Sort to BWT Algorithms

The “relative” Gray code sort described in the preceding section likely cannot be computed in $O(m)$ time. The sort direction of a column $k+1$ cannot be determined until column k is completely sorted. The same problem also slows the computation of the inverse BWT. A much more feasible Gray code sort is the “absolute” one where the symbols in the alphabet are assigned alternating sort directions. With one bit per symbol, the sort order of a suffix $d_i...d_m\$$ can be determined with exclusive-ors: $XOR_{j=i}^{k} GrayOrder(d_j)$ where $GrayOrder(d_j)$ returns 0 if strings following d_j are sorted in standard forward order or 1 if the strings are sorted in reverse order. This is the kind of Gray-code sorting done by Richards [39]. Both Gray code sorts improved compression by a very small amount (0.25%) on most data, so one can expect the gain of the relative version over the absolute to be miniscule, and certainly not worth an increase in the time complexity of the BWT.

Modifying Burrows-Wheeler algorithms to do Gray code sorting (absolute, not relative) instead of standard sorting is easy. For the algorithms that use a doubling technique (sort all strings by first $1, 2, 4, 8, 16, ..., m$ characters after each pass over the data) such as Sadakane’s algorithm and the Forward Radix Sort, one more bit of memory per string is needed to store the sorting direction of data following each string. For the direct comparison algorithms such as the original Burrows-Wheeler sort, no extra memory is needed. The direction bits can be computed from the single character being compared. In both cases, for each comparison of strings made, an additional XOR on the direction bits is needed, which does not increase the time complexity of the algorithms. For sorting into buckets, the direction bits determine whether a bucket is filled starting at the first element of the bucket as is done in a standard sort, or filled starting at the last element.

Modifying limited context versions of the block sort algorithms to do Gray code sort is not as straightforward. The direction bits maintained in algorithms which use a doubling technique do not help in determining sort direction for contexts limited to lengths that are not a power of 2. See Table 4.4 for an example on the data “abracadabra” with a context limit of 7 characters. There is no way to determine

	abracadabra
1	01100010110
2	10100111010
4	00111010000
8	10011010011
7	11111100010

Table 4.5: Direction bits for each 1, 2, 4, 8, and 7 character wrap-around substrings of "abracadabra"

the length 7 direction bits from only the length 4 direction bits. In a standard sort, determining the order of the substrings out to 7 characters from the order out to length 4 is done simply by overlapping, (ex. "abra" with "acad" to sort "abracad") which does not work for the direction bits.

More elaborate sorting schemes may be devised. Rather than order only single symbols, which can be done in a preprocessing step, an order for all pairs (or longer strings) can be defined. For example, in English, 'c' might be more like an 's' when preceded by certain letters and more like a 'k' otherwise. Ordering 'c' next to 's' when preceded by 'n' and next to 'k' when preceded by 'o' could improve compression. As noted previously on the reflected sort, the greatest gains came from reflecting the first columns. So such additional ordering would likely not improve compression much.

4.5 Summary

Sorting is a central part of compression based on the Burrows-Wheeler transform, and yet standard lexicographic sorting is only one of many possible ways to sort. In the previous chapter we considered alternative alphabet orderings based on both hand-selected (heuristic) and structured techniques (such as using a reduction to the traveling salesperson problem to compute an alphabet reordering). In this chapter, we considered reorderings based on larger sequences of characters and modifications of BWT algorithms needed to perform Gray code sorting. Improvements in compressed size were obtained by both alphabet reordering and selective reversal of ordering within columns of the sorted matrix.

Both techniques add to the compression time, but alphabet reordering adds almost nothing to the decompression time (and reordering with a fixed permutation adds almost nothing to the compression time). The reflected sorting provided additional improvements, and was only slightly slower in both compression and decompression.

CHAPTER 5

Analysis of Dynamic Update Algorithms

5.1 Introduction

There are several ways of analyzing dynamic update algorithms such as Move-To-Front. Early efforts focused on their performance as a strategy for dynamically updating a list data structure. In the List Update (also called Dictionary) problem, there is a list L of items and a series S of requests for those items. For any of a variety of reasons, the list is accessed sequentially when searching for a requested symbol. Obviously, searching time can be reduced by putting the most commonly requested symbols at the front of the list. Dynamic update algorithms reorder the list between requests to try to reduce the search time. The effectiveness of the various algorithms is really the second aspect to consider. First is the exact problem the algorithms are addressing.

Some ways in which dynamic update algorithms have been analyzed:

- S is independent identically distributed (i.i.d.)
- S is arbitrary
  - S must be served on-line
  - update algorithm must be memoryless

in combination with

- cost of request is rank of requested symbol in L
- cost of request is cost of describing the rank in L

Any convex cost function will serve; an algorithm that orders the list well for the simple "cost is equal to the rank" function does so for any convex function such as log [48]. Jensen's inequality, states that $f(E(cost)) \leq E(f(cost))$ for any convex function f. Most of the attention has been focused on the easy $c = \rho(s_t)$ function as some upper bound results can be extended to any convex cost function.

The cost function of interest is the cost function of an entropy coder. probability rank codes are a favorite in proofs, providing a simple $c(l) = 2\log\rho(l)$ function that is close to the lower bound cost given by the Kraft inequality, $\sum_{l\in L}|C|^{-c(l)} \leq 1$ where C is the coding alphabet for L. One can achieve good results with cost functions that are less than perfect. Throughout this chapter, we prove results using $c = \rho(s_t)$. We note here that one can use Jensen's inequality to extend some of the results to the cost of probability rank codes which are close to the actual costs of the entropy coder.

If S is i.i.d., then optimal performance can be approached. Frequency Count (with unlimited counters) will order the list optimally in the limit. Transpose has been conjectured to be the lowest cost memoryless algorithm. (It is interesting to note the severe emphasis papers of this era put on memory or "core" as it was commonly called– 256 bytes was considered a lot of core!) Move-To-Front performs relatively poorly at 1.386 times optimal on Zipf distributed data for large alphabets [40]. Experiments show the increase in entropy on data transformed by Move-To-Front is highest on data with entropy half of the maximum possible entropy. At maximum entropy, in which the distribution is uniform, all reasonable algorithms are optimal because the list order does not matter. Intuitively, MTF's performance should be worst at the balance point between where the order matters but is hard to determine because the entropy is high.

For data that is not i.i.d., we analyse the performance with different methods, since optimal performance is no longer easy. In Competitive Analysis, we pit an "adversary" against an algorithm. The adversary generates worst-case data so as to maximize the ratio of that algorithm's cost to the optimum cost. One of the interesting features of Competitive Analysis is that, unlike earlier methods, it shows that randomization can help. Randomized algorithms have better performance against some kinds of adversaries. The randomization prevents the adversary from computing the algorithm's state and therefore which request will maximize the cost ratio. However, randomization is not useful for the problems studied in this dissertation.

If S is arbitrary rather than i.i.d., and $c = \rho(s_t)$, then Move-To-Front is never worse than 2 times the cost of S to an optimum algorithm [40]. This result was later improved to $2 - \frac{2}{|L|-1}$ and shown, through Competitive Analysis, to be optimal for deterministic on-line algorithms [27]. The same worst-case type of analysis showed that Frequency Count and Transpose, the "best" algorithms for i.i.d., can be arbitrarily bad on worst-case data. No non-exponential optimal off-line algorithm for the List Update problem is known, which limits the experimental comparisons that can be easily made.

When we switch from the data structures problem to the problem of encoding i.i.d. data, the cost function changes. The cost is now the cost of describing the location of the requested item in the list, which is $O(\log n)$. Huffman coding is optimal per request, and arithmetic coding is optimal over the entire request sequence. Move-To-Front, with probability rank codes for describing the position within L, achieves cost of $\Theta(H \log H)$ where H is the entropy of the sequence [9].

The output of the Burrows-Wheeler Transform can be described as piecewise i.i.d. although it is not actually p.i.i.d. To demonstrate this, consider binary data in which '0' is more common than '1'. Long strings of '0's occur first in the sort and may be preceded by a '0' or a '1'. But the longest strings of '0's will be first in the sort order and are always preceded by a '1'. The sorting process does not collect the similar contexts together in a random fashion. Thus, unless the input data is all one character, the first symbol of BWT output cannot be the first symbol of the alphabet and the last symbol of output cannot be the last symbol of the alphabet. The set of output strings that BWT can produce is only a subset of the set of strings a p.i.i.d. source with analogous probabilities can produce. Another way to see this is to note that the BWT produces the same output string for each of m rotations of some input block of length m. Only the $\log m$ size pointer will differ. Also, not all rotations of valid BWT output are also valid BWT output.

Even so, algorithms suitable for p.i.i.d. data (good for encoding i.i.d. data and good at adapting to new data– a new piece of i.i.d. data) perform well on BWT output. The strategy of maintaining a list of symbols in the order of probability and then rapidly reordering the list as the probabilities seem to change, followed by entropy coding, produces among the best experimental results on BWT output. When adapting a list to a new distribution, there is naturally some overhead. This "overwork" becomes insignificant as the size of the i.i.d. piece goes to infinity, and so was at first ignored. But in p.i.i.d. data, overwork is important and rapid adaptation can reduce that cost. The tradeoff to continuous rapid adaptation is the higher cost incurred when no change is occurring, as happens when in the middle of coding a large i.i.d. piece.

Before the advent of Competitive Analysis, Bitner analyzed the overwork of Move-To-Front, Transpose, and other algorithms when starting with an arbitrary list on i.i.d. data [11]. He showed that although Transpose and Frequency Count are low cost in the limit, they can converge very slowly. Move-To-Front converges much more rapidly. In short, Move-To-Front has less overwork.

In the following sections, we extend this idea to analyze the overwork incurred when switching distributions, rather than just the overwork from starting the list in some arbitrary order. Much work on the steady state costs of MTF [40] and Timestamp [2] has been done, but we are not aware of any previous analysis of overwork for these algorithms. We show that M1FF has superior performance to MTF in steady state cost and similar total cost (asymptotic cost plus overwork). We introduce the Best x of $2x - 1$ family of algorithms, of which MTF, Timestamp, and Frequency Count are members, and show that the steady state cost is inversely proportional to x and the overwork is proportional.

Much of this analysis uses "list factoring", meaning we can concentrate on one pair of symbols at a time, independently of all other pairs. Most of the following algorithms order a pair 'a' and 'b' in the list relative to one another according to a limited number of recent occurrences of those symbols in the request sequence. In determining whether 'a' should be somewhere in front of 'b' or not, occurrences of other symbols can be ignored. Two algorithms that do not determine list order as described above are Transpose and Move One From Front (discussed later). As far as we know, no one has found a way to analyze asymptotic cost and overwork of Transpose and Move One From Front using list factoring or any other method.

5.2 Overwork and Steady State Cost

5.2.1 Timestamp Analysis

Following is an analysis of part of the Timestamp algorithm. In Albers' paper [1], Timestamp is a randomised algorithm which updates its list according to the MTF strategy with probability p or with the following strategy with probability $1-p$: move the requested item a in front of the frontmost item that was requested at most once since the previous request for a. The strategy chosen with probability $1-p$, the only part of Timestamp we are interested in, is actually Best 2 of 3, which is discussed later. To avoid confusion, we will use "Best 2 of 3" and not "Timestamp" when we mean the part of Timestamp selected with probability $1-p$.

Theorem 6 *The expected cost of Best 2 of 3 t requests after a switch from probability distribution Q to P is*

$$E(Cost_{B2}) = 1 + \sum_{1\le i<j\le n} \frac{p_i p_j (p_i^2 + 6p_i p_j + p_j^2)}{(p_i+p_j)^3} + OV_{B2}$$

and the overwork is

$$\begin{aligned} OV_{B2} \;=\; & \sum_{i=1}^{n}\sum_{j\neq i} p_i \Bigg[\left(\frac{q_j^3 + 3q_i q_j^2}{(q_i+q_j)^3} - \frac{p_j^3 + 3p_i p_j^2}{(p_i+p_j)^3} \right) (1-p_i-p_j)^t + \\ & \left(\frac{(p_i+p_j)q_j^2 + 2p_j q_i q_j}{(q_i+q_j)^2} - \frac{p_j^3 + 3p_i p_j^2}{(p_i+p_j)^2} \right) t(1-p_i-p_j)^{t-1} + \\ & \left(\frac{2q_j p_i p_j}{q_i+q_j} - \frac{2p_i p_j^2}{p_i+p_j} \right) \binom{t}{2} (1-p_i-p_j)^{t-2} \Bigg] \end{aligned}$$

Proof: We begin with observations similar to those made for MTF in chapter 2.

Probability of neither l_i or l_j in t requests: $(1-p_i-p_j)^t$

Probability of exactly one l_i or l_j in t requests: $t(p_i+p_j)(1-p_i-p_j)^{t-1}$

Probability of exactly u l_i or l_j in t requests: $\binom{t}{u}(p_i+p_j)^u(1-p_i-p_j)^{t-u}$

Given 3 occurrences of l_i or l_j, $\rho(l_j) < \rho(l_i)$ if the following sequences occurred, ignoring requests for other items: $l_j l_j l_j$, $l_i l_j l_j$, $l_j l_i l_j$, and $l_j l_j l_i$. The description of Best

2 of 3 is “put a in front of b if at least 2 of the last 3 occurrences for either were for a”, which is exactly what determines which of l_i and l_j is in front of the other. The probability of l_j being in front of l_i is

$$\begin{aligned}
&Prob(\rho(l_j) < \rho(l_i)) \\
= \; & \frac{q_j^3 + 3q_iq_j^2}{(q_i+q_j)^3}(1-p_i-p_j)^t + \\
& \frac{(p_i+p_j)q_j^2 + 2p_jq_iq_j}{(p_i+p_j)(q_i+q_j)^2} t(p_i+p_j)(1-p_i-p_j)^{t-1} + \\
& \frac{(q_i+q_j)p_j^2 + 2q_jp_ip_j}{(q_i+q_j)(p_i+p_j)^2}\binom{t}{2}(p_i+p_j)^2(1-p_i-p_j)^{t-2} + \\
& \frac{p_j^3+3p_ip_j^2}{(p_i+p_j)^3}\left(1 - \sum_{u=0}^{2}\binom{t}{u}(p_i+p_j)^u(1-p_i-p_j)^{t-u}\right) \\
= \; & \frac{p_j^3+3p_ip_j^2}{(p_i+p_j)^3} + \left(\frac{q_j^3+3q_iq_j^2}{(q_i+q_j)^3} - \frac{p_j^3+3p_ip_j^2}{(p_i+p_j)^3}\right)(1-p_i-p_j)^t + \\
& \left(\frac{(p_i+p_j)q_j^2+2p_jq_iq_j}{(p_i+p_j)(q_i+q_j)^2} - \frac{p_j^3+3p_ip_j^2}{(p_i+p_j)^3}\right) t(p_i+p_j)(1-p_i-p_j)^{t-1} + \\
& \left(\frac{(q_i+q_j)p_j^2+2q_jp_ip_j}{(q_i+q_j)(p_i+p_j)^2} - \frac{p_j^3+3p_ip_j^2}{(p_i+p_j)^3}\right)\binom{t}{2}(p_i+p_j)^2(1-p_i-p_j)^{t-2} \\
= \; & \frac{p_j^3+3p_ip_j^2}{(p_i+p_j)^3} + \left(\frac{q_j^3+3q_iq_j^2}{(q_i+q_j)^3} - \frac{p_j^3+3p_ip_j^2}{(p_i+p_j)^3}\right)(1-p_i-p_j)^t + \\
& \left(\frac{(p_i+p_j)q_j^2+2p_jq_iq_j}{(q_i+q_j)^2} - \frac{p_j^3+3p_ip_j^2}{(p_i+p_j)^2}\right) t(1-p_i-p_j)^{t-1} + \\
& \left(\frac{(q_i+q_j)p_j^2+2q_jp_ip_j}{q_i+q_j} - \frac{p_j^3+3p_ip_j^2}{p_i+p_j}\right)\binom{t}{2}(1-p_i-p_j)^{t-2} \\
= \; & \frac{p_j^3+3p_ip_j^2}{(p_i+p_j)^3} + \left(\frac{q_j^3+3q_iq_j^2}{(q_i+q_j)^3} - \frac{p_j^3+3p_ip_j^2}{(p_i+p_j)^3}\right)(1-p_i-p_j)^t + \\
& \left(\frac{(p_i+p_j)q_j^2+2p_jq_iq_j}{(q_i+q_j)^2} - \frac{p_j^3+3p_ip_j^2}{(p_i+p_j)^2}\right) t(1-p_i-p_j)^{t-1} + \\
& \left(\frac{2q_jp_ip_j}{q_i+q_j} - \frac{2p_ip_j^2}{p_i+p_j}\right)\binom{t}{2}(1-p_i-p_j)^{t-2}
\end{aligned}$$

a	a	a	b	b	b	b	c	c	b	b	a	a	c	c
b	1	a	2	a	2	b	3	b	1	c	2	b	3	a
c		b		a		a		b		b		c		b
a		c		b		a		a		a		b		c
b		b		c		c		a		a		a		b
c		c		c		c		c		c		c		a

Table 5.1: Best 2 of 3 algorithm on request sequence "abbcbac"

The above expression can be explained thus: The probability that $\rho(l_j) < \rho(l_i)$ is the probability that l_j is in front of l_i in the steady state, achieved for 3 or more occurrences of the 2 symbols, plus the overwork incurred for 0 occurrences plus overwork for 1 occurrence plus overwork for 2 occurrences.

$$\begin{aligned}
E(Cost_{B2}) &= 1 + \sum_{i=1}^{n} \sum_{j \neq i} p_i Prob(\rho(l_j) < \rho(l_i)) \\
&= 1 + \sum_{i=1}^{n} \sum_{j \neq i} p_i \frac{p_j^3 + 3p_i p_j^2}{(p_i + p_j)^3} + OV_{B2} \\
&= 1 + \sum_{1 \leq i < j \leq n} \frac{p_i p_j^3 + 3p_i^2 p_j^2 + 3p_i^2 p_j^2 + p_i^3 p_j}{(p_i + p_j)^3} + OV_{B2} \\
&= 1 + \sum_{1 \leq i < j \leq n} \frac{p_i p_j (p_i^2 + 6p_i p_j + p_j^2)}{(p_i + p_j)^3} + OV_{B2}
\end{aligned}$$

∎

Best 2 of 3 Algorithm

A simple algorithm to perform Best 2 of 3 is to use a double length list. That is, if the alphabet has 256 symbols, the list for Best 2 of 3 will contain 512. Each symbol occurs in the list twice. When a request is made, the position of an item is 1 plus the number of symbols for which both occurrences are in front of the second occurrence of the requested symbol. Then the list is updated by moving that second occurrence to the front. Table 5.1 demonstrates this algorithm on a sample sequence. See the section on Best x of $2x-1$ for another better algorithm.

5.2.2 Move One From Front Analysis

An improvement over MTF suggested by Balkenhol and co-authors [6, 7, 8] and by Schindler shall be referred to here as Move One From Front (M1FF).

Experimentally, M1FF performs better than MTF in BW compression.

The Move One From Front (M1FF) heuristic works as follows: After accessing an item at rank i, if $i \geq 3$ move the item to rank 2 (and move all items in ranks 2 through $i-1$ back one position) else move the item to rank 1.

From the standpoint of the List Update problem, M1FF is 2-competitive, which is optimal but not strictly optimal. Strictly optimal on-line algorithms such as MTF are $2 - 2/(|L|-1)$-competitive, which converges to 2 as $|L|$ goes to infinity. An algorithm's optimality under Competitive Analysis is important for BW compression (otherwise Transpose would be just as good an algorithm) but optimal vs. strictly optimal may not matter too much.

For the following proof, we use a cost function from Irani [27] with corrections to account for the necessity of paid exchanges in an optimal List Update algorithm from Borodin and El-Yaniv's book on Competitive Analysis [12].

Theorem 7 *M1FF is 2-competitive.*

Proof: To see that M1FF is 2-competitive, compare M1FF with MTF. The cost to M1FF to access a symbol is at most 1 more than the cost to MTF. Intuitively, one can see this by observing that M1FF behaves exactly like MTF except that a symbol can become lodged in position 1, increasing the cost of access for all other symbols by 1 over MTF's cost.

We use the following cost function for MTF on a pair of symbols l_i and l_j when a request for l_i has occurred:

$$c_{MTF}(i,j) = \begin{cases} 1 + 1/(|L|-1) & \text{if } l_j \text{ in front of } l_i \\ 1/(|L|-1) & \text{if } l_i \text{ in front of } l_j \end{cases}$$

The cost to serve one request for l_i is $\sum_{j \neq i} c(i,j) = \rho(l_i)$.

An optimum algorithm, OPT, has the same costs, but may have ordered the list better and so pay the lower $1/(|L|-1)$ more often than MTF. For every two requests for l_i where MTF pays for l_j being in front of l_i at a cost of $1 + 1/(|L|-1)$, there must be at least one request in between for l_j. If OPT does not pay for l_j (pays $1/(|L|-1)$ each time) to serve the two requests for l_i, then to serve the request for l_j OPT pays $1 + 1/(|L|-1)$ for l_i or pays $1/(|L|-1)$ plus 1 per the one or more paid exchanges of l_i with l_j. Therefore the ratio of MTF's cost to OPT's is at most $\frac{2+2/(|L|-1)}{1+2/(|L|-1)} = 2 - 2/(|L|+1)$.

Since M1FF can have at most a cost of 1 more per request than MTF, M1FF's cost is at most

$$c_{M1FF}(i,j) = \begin{cases} 1 + 2/(|L|-1) & \text{if } l_j \text{ in front of } l_i \\ 2/(|L|-1) & \text{if } l_i \text{ in front of } l_j \end{cases}$$

Therefore the ratio of M1FF's cost to OPT's is at most $\frac{2+4/(|L|-1)}{1+2/(|L|-1)} = 2$.

For a tight bound, consider the following sequence on an alphabet of 3 symbols, 'a', 'b', and 'c': aacbcbcbcbcb.... M1FF will put 'a' in position 1. 'b' and 'c' will continually swap in positions 2 and 3, and M1FF will spend 3 per occurrence. The optimal algorithm will put the list in the order 'cba' and never change it, spending 1 for each access to 'c' and 2 for each access to 'b' for an average cost of 1.5, which is exactly half of M1FF's cost.

Therefore M1FF is 2-competitive. ▮

The argument for the upper bound can be applied to M1FF2 and M1FF3, showing that they also are at least 2-competitive.

Why M1FF does better than MTF

The most common symbol in an alphabet is special. It is the only symbol which cannot occur with probability less than $\frac{1}{|L|}$ and the only one which can occur with probability greater than 0.5. All other symbols may have a probability of occurrence as low as zero. Therefore, moving the most common symbol away from the front of the list should be avoided as that inversion is potentially more expensive than any other. MTF does this every time any other symbol is encountered. M1FF may mistakenly move the most common symbol away from the front but does so less often and so performs better than MTF.

An improvement to M1FF

Can this desirable property of M1FF be retained while modifying it so that an adversary cannot generate a sequence costing 2 times optimal on an alphabet of only 3 symbols? Yes. Let version 2 (M1FF2) be defined as follows: After accessing item at position i, move the item to position 2 if the previously accessed item is in position 1 and $i \geq 3$. Otherwise move the item to position 1.

Comparisons of the compression rates achieved with the 2 versions show that they are very close. Both are much better than MTF. M1FF2 achieved higher compression than M1FF as one of the pair of algorithms used in the switching scheme (discussed later), while M1FF was the superior of the 2 when used alone.

But neither M1FF nor M1FF2 are easy to analyze. For M1FF, a symbol can becomed lodged in the first position. For M1FF2, consider a list of 3 items *abc* and a request sequence $(abc)^+$. If the list started in order *abc*, the final list order will be *bca*. But if the list started in order *cba* then the list will end in order *cab*. Equations that model this behavior are difficult to find.

M1FF3

The reasons for considering another version of M1FF are

- The expected cost can be formulated
- The overwork can be formulated
- Like other M1FF versions, it is better than MTF in practice
- It is strictly optimally competitive

How M1FF3 works: if the last two requests are for the same item ($s_{t-2} = s_{t-1}$) and the current request is for a different one ($s_t \neq s_{t-1}$), move the requested item to rank 2, else move it to the front. Item l_j is in front of item l_i if l_j is the most recently requested of the two, with one exception. If the last request for l_j is preceded by two requests for l_i, then l_i will be in front of l_j and vice versa.

First, we show that M1FF3 is strictly optimally competitive. We already know M1FF3 is 2-competitive because, like M1FF, it never pays more than 1 above MTF's cost per request. The argument used to prove MTF $2-2/(|L|+1)$-competitive works for M1FF3.

Theorem 8 *M1FF3 is* $2-2/(|L|+1)$*-competitive.*

Proof: The following cost function for a pair of symbols l_i and l_j when a request for l_i has occurred holds for M1FF3 as well as MTF:

$$c_{M1FF3}(i,j) = \begin{cases} 1 + 1/(|L|-1) & \text{if } l_j \text{ in front of } l_i \\ 1/(|L|-1) & \text{if } l_i \text{ in front of } l_j \end{cases}$$

The cost to serve one request for l_i is $\sum_{j \neq i} c(i,j) = \rho(l_i)$.

Consider 2 requests for l_i which are adjacent in the request sequence if requests for all other items are removed, and in which both times an optimal algorithm OPT does not pay for l_j and M1FF3 does. Then, exactly like MTF, either at least one request for l_j occurred between the two requests for l_i, or, specific to M1FF3, at least 2 requests for l_j immediately preceded the first request for l_i. For the first case, refer back to the proof of MTF's competitive ratio. In the second case, since there are two adjacent requests for l_j, an optimum strategy is to move l_j to the front before serving either. Then, after serving the 2 requests for l_j and before serving the first of the 2 requests for l_i, OPT must make at least 1 paid exchange to put l_j behind l_i. In both cases, the ratio of M1FF3's cost to OPT's is at most $\frac{2+2/(|L|-1)}{1+2/(|L|-1)} = 2-2/(|L|+1)$. ▮

And now, we present the formula for M1FF3's cost on p.i.i.d. data.

Theorem 9 *The expected cost of M1FF3 t requests after a switch from probability distribution Q to P is*

$$E(Cost_{M1FF3}) = 1 + \sum_{i=1}^{n} p_i \sum_{j \neq i} \frac{p_j + p_j^2 p_i - p_i^2 p_j}{p_i + p_j} + OV_{M1FF3}$$

and the overwork is

$$\begin{aligned} OV_{M1FF3} \quad = \quad & \left(\frac{q_j + q_j^2 q_i - q_i^2 q_j}{q_i + q_j} - \frac{p_j}{p_i + p_j} \right) (1 - p_i - p_j)^t \\ & + \begin{cases} 0 & \textit{if } t = 0 \\ (q_j^2 p_i - q_i^2 p_j)(1 - p_i - p_j)^{t-1} & \textit{if } t \geq 1 \end{cases} \\ & + \begin{cases} \frac{p_i^2 p_j - p_j^2 p_i}{p_i + p_j} & \textit{if } t \leq 1 \\ (q_j p_j p_i - q_i p_i p_j + \frac{p_i^2 p_j - p_j^2 p_i}{p_i + p_j})(1 - p_i - p_j)^{t-2} & \textit{if } t \geq 2 \end{cases} \end{aligned}$$

Proof: Probability l_j is most recently requested, given that a request for l_i or l_j has occurred: $\frac{p_j}{p_i + p_j}$.

Probability that an occurrence of l_i or l_j is the end of the sequence $l_i l_i l_j$: $\frac{p_i^2 p_j}{p_i + p_j}$.

Probability of no l_i or l_j in t requests: $(1 - p_i - p_j)^t$.

Probability of one or more l_i or l_j in t requests: $1 - (1 - p_i - p_j)^t$.

Probability that most recent l_i or l_j is part of $l_i l_i l_j$ in $t \geq 2$ requests: $\frac{p_i^2 p_j}{p_i + p_j}(1 - (1 - p_i - p_j)^{t-2})$.

Probability that $l_i l_i l_j$ occurred at time 0 and no l_i or l_j has occurred since: $q_i^2 p_j (1 - p_i - p_j)^{t-1}$.

Probability that $l_i l_i l_j$ occurred at time 1 and no l_i or l_j has occurred since: $q_i p_i p_j (1 - p_i - p_j)^{t-2}$.

$$E(Cost_{M1FF3}) = 1 + \sum_{i=1}^{n} p_i \sum_{j \neq i} \frac{p_j + p_j^2 p_i - p_i^2 p_j}{p_i + p_j} + OV_{M1FF3}$$

$$OV_{M1FF3} \quad = \quad \left(\frac{q_j + q_j^2 q_i - q_i^2 q_j}{q_i + q_j} - \frac{p_j}{p_i + p_j} \right) (1 - p_i - p_j)^t$$

$$+\begin{cases} 0 & \text{if } t = 0 \\ (q_j^2 p_i - q_i^2 p_j)(1 - p_i - p_j)^{t-1} & \text{if } t \geq 1 \end{cases}$$
$$+\begin{cases} 0 & \text{if } t \leq 1 \\ (q_j p_j p_i - q_i p_i p_j)(1 - p_i - p_j)^{t-2} & \text{if } t \geq 2 \end{cases}$$
$$-\begin{cases} \frac{p_j^2 p_i - p_i^2 p_j}{p_i + p_j} & \text{if } t \leq 2 \\ \frac{p_j^2 p_i - p_i^2 p_j}{p_i + p_j}(1 - p_i - p_j)^{t-2} & \text{if } t \geq 3 \end{cases}$$

$$\begin{aligned} OV_{M1FF3} &= \left(\frac{q_j + q_j^2 q_i - q_i^2 q_j}{q_i + q_j} - \frac{p_j}{p_i + p_j} \right) (1 - p_i - p_j)^t \\ &+ \begin{cases} 0 & \text{if } t = 0 \\ (q_j^2 p_i - q_i^2 p_j)(1 - p_i - p_j)^{t-1} & \text{if } t \geq 1 \end{cases} \\ &+ \begin{cases} \frac{p_i^2 p_j - p_j^2 p_i}{p_i + p_j} & \text{if } t \leq 1 \\ \left(q_j p_j p_i - q_i p_i p_j + \frac{p_i^2 p_j - p_j^2 p_i}{p_i + p_j} \right)(1 - p_i - p_j)^{t-2} & \text{if } t \geq 2 \end{cases} \end{aligned}$$

∎

5.2.3 Best x of $2x - 1$

A simple way to describe the second part of Timestamp, which is "move the requested item a in front of the frontmost item that was requested at most once since the previous request for a", is to note that, for any two symbols l_i and l_j, the one closest to the front will be the one that was requested 2 or more times in the last 3 requests for either. Call it "2 out of 3". Modifying that part of Timestamp to be "3 out of 5" or "4 out of 7" makes it roughly 3 or 4 competitive respectively, but reduces the steady state cost. This leads to a family of algorithms we call "Best x of $2x - 1$", or just "Best x" for brevity, because the symbol that goes to the front is the one that "wins" the most games, as is done in the playoffs of baseball and other sports. (And sports championship series are often called "Best of $2x - 1$", but we have not used that term in this work.) The Best x of $2x - 1$ heuristic is "given 2 symbols a and b, put a in front of b if a was requested x or more times in the last $2x - 1$ requests for either a or b."

We initialized Best x of $2x-1$ by setting all the counts as if the algorithm had been conditioned on the following request sequence: a string, repeated x times, which contains one occurrence of every symbol in the source alphabet, in reverse order of the order desired in the initial list of Best x of $2x-1$. Since all the counts start the same, Best x of $2x-1$ will behave like Frequency Count with all frequencies started at 0, adjusting quickly to the first data no matter how large x is.

A caution for those trying to invent new dynamic update algorithms (or most any other similar endeavor) is to beware of the impossible. One algorithm that defined how two symbols should be ordered relative to each other seemed very promising until we realized it wasn't possible because in some sequences, such as "aaccbabca" the list had to be ordered so that 'a' is in front of 'b', 'b' in front of 'c', and 'c' in front of 'a'. And that algorithm? Given all occurrences of two symbols 'a' and 'b' (ignore occurrences of other symbols), put in front whichever one most recently occurred two times in succession. For example, put 'b' in front of 'a' if the sequence is "$bb(ab)^*a$".

Best x of $2x-1$ is a valid algorithm. The relationship between two symbols is consistent for all pairs. That is, if a precedes b and b precedes c, then a precedes c. The following theorem formalizes this and provides a proof.

Theorem 10 *Let L be an ordered list dynamically updated according to Best x of $2x-1$'s rule. For any $a, b, c \in L$ if $\rho(a) < \rho(b)$ and $\rho(b) < \rho(c)$ then $\rho(a) < \rho(c)$ where $\rho(a)$ is the position of a in L.*

Proof: Consider the xth occurrence, counting backward from a time t, of each of a, b, and c in a sequence of requests. If a is in front of b in Best x of $2x-1$'s list, then the xth occurrence of a must be at a later time than the xth occurrence of b. Equally, if b precedes c in Best x of $2x-1$'s list, the xth occurrence of b is at a later time than that of c. Therefore, the xth occurrence of a comes later than the xth occurrence of c, and a precedes c in Best x of $2x-1$'s list. ▮

Having established that Best x of $2x-1$ works, how much does it cost? We derive formulas for the overwork when switching from one i.i.d. distribution to another and asymtotic cost on an i.i.d. sequence for Best x of $2x-1$.

Theorem 11 *The expected cost of Best x of $2x-1$ t requests after a switch from distribution Q to P is*

$$E(Cost_{Bx}) = 1 + \sum_{i=1}^{n} p_i \sum_{j \neq i} (p_i + p_j)^{1-2x} \sum_{w=0}^{x-1} \binom{2x-1}{w} p_i^w p_j^{2x-1-w} + OV_{Bx}$$

and the overwork is

$$\begin{aligned} OV_{Bx} = & \sum_{i=1}^{n} p_i \sum_{j \neq i} \sum_{v=0}^{2x-2} \binom{t}{v} (1 - p_i - p_j)^{t-v} \\ & \left((q_i + q_j)^{-\overline{v}} \sum_{w_1=0}^{\min(x-1,v)} \sum_{w_2=0}^{\min(x-1-w_1,\overline{v})} \binom{v}{w_1} p_i^{w_1} p_j^{v-w_1} \binom{\overline{v}}{w_2} q_i^{w_2} q_j^{\overline{v}-w_2} \right. \\ & \left. -(p_i + p_j)^{-\overline{v}} \sum_{w=0}^{x-1} \binom{2x-1}{w} p_i^w p_j^{2x-1-w} \right) \end{aligned}$$

where $\overline{v} = 2x - 1 - v$.

Proof: The equation is the sum of the probabilities of all the different possibilities. In the asymptotic cost, at least $2x-1$ of each l_i or l_j has occurred. Given the $2x-1$ occurrences (probability $(p_i+p_j)^{1-2x}$), if a majority are for l_j, then l_j is in front of l_i. So l_j is in front with $0 \leq w \leq x-1$ occurrences of l_i. The probability of an individual sequence that puts l_j in front of l_i is $p_i^w p_j^{2x-1-w}$. The number of such sequences is $\binom{2x-1}{w}$. Combining all the probabilities yields the asymptotic cost of Best x of $2x-1$.

Overwork is present as long as distribution Q affects the algorithm's list order. For each pair of symbols l_i, l_j, overwork is present if at least 1 l_i or l_j was drawn with probability q_i or q_j, leaving up to $2x-2$ symbols drawn from P. $0 \leq v < 2x-2$ symbols are drawn according to P. Then $2x-2-v \geq 1$ symbols are drawn according to Q and so contribute to the overwork. w and w_1 are the number of l_i's drawn according to P. For l_j to be in front of l_i, $0 \leq w \leq x-1$ and $0 \leq w_1 < \min(x-1, v)$. w_2 is the number of l_i's drawn according to Q. $0 \leq w_2 < \min(x-1-w_1, \overline{v})$. The nested summation for the q's enumerates every combination in which

- The total degree of q_i's, q_j's, p_i's, and p_j's is $2x-1$. So $w_1+v-w_1+w_2+\overline{v}-w_2 = 2x-1$.

- The power of p_j's plus q_j's is x or greater, and therefore greater than the power of p_i's plus q_i's.
- There is at least one q_i or q_j.
- All q's occur before all p's. So qqp is acceptable but pqq and qpq are not.

Best 1 of 1 is MTF and Best 2 of 3 is the second part of Timestamp, which have already been given. Examining the pattern of the equations from those two and Best 3 of 5, given below, it is easy to derive a generalized form for Best x of $2x-1$.

The costs for Best 3 of 5 and Best 4 of 7 are

$$\begin{aligned}
&E(Cost_{B3})\\
&= 1+\sum_{i=1}^{n}\sum_{j\neq i} p_i \frac{p_j^5+5p_ip_j^4+10p_i^2p_j^3}{(p_i+p_j)^5}+OV_{B3}\\
&= 1+\sum_{1\le i<j\le n}\frac{p_ip_j(p_j^4+5p_ip_j^3+20p_i^2p_j^2+5p_i^3p_j+p_i^4)}{(p_i+p_j)^5}+OV_{B3}\\
&E(Cost_{B4})\\
&= 1+\sum_{1\le i<j\le n}\frac{p_ip_j(p_j^6+7p_ip_j^5+21p_i^2p_j^4+70p_i^3p_j^3+21p_i^4p_j^2+7p_i^5p_j+p_i^6)}{(p_i+p_j)^7}+OV_{B4}
\end{aligned}$$

and the overwork for Best 3 of 5 is

$$\begin{aligned}
&OV_{B3}\\
&= \sum_{i=1}^{n} p_i \sum_{j\neq i}\left(\frac{q_j^5+5q_iq_j^4+10q_i^2q_j^3}{(q_i+q_j)^5}-\frac{a}{(p_i+p_j)^5}\right)b^t+\\
&\left(\frac{(p_i+p_j)q_j^4+4(p_i+p_j)q_iq_j^3+6p_jq_i^2q_j^2}{(q_i+q_j)^4}-\frac{a}{(p_i+p_j)^4}\right)tb^{t-1}+\\
&\left(\frac{(p_i+p_j)^2q_j^3+3(2p_ip_j+p_j^2)q_iq_j^2+3p_j^2q_i^2q_j}{(q_i+q_j)^3}-\frac{a}{(p_i+p_j)^3}\right)\binom{t}{2}b^{t-2}+\\
&\left(\frac{(3p_i^2+3p_jp_i+p_j^2)p_jq_j^2+2(3p_i+p_j)p_j^2q_jq_i+p_j^3q_i^2}{(q_i+q_j)^2}-\frac{a}{(p_i+p_j)^2}\right)\binom{t}{3}b^{t-3}+\\
&\left(\frac{(6p_i^2+4p_ip_j+p_j^2)p_j^2q_j+(4p_i+p_j)p_j^3q_i}{q_i+q_j}-\frac{a}{p_i+p_j}\right)\binom{t}{4}b^{t-4}
\end{aligned}$$

where

$$a = p_j^5 + 5p_i p_j^4 + 10p_i^2 p_j^3$$

and

$$b = 1 - p_i - p_j$$

. ▮

Algorithms for Best x of $2x - 1$

We extended the algorithm used for Best 2 of 3 for our experiments. Best 2 of 3 used a double sized list. Best x of $2x-1$ can be implemented with a list containing x copies of each symbol. In practice, with x seldom larger than 6, the above algorithm's time dependence on x was not a problem. Below, we give a better algorithm with time complexity independent of x.

The proof of Best x of $2x - 1$'s validity was the inspiration for this algorithm. Maintain $|L|$ linked lists, each composed of x nodes. When a request is served for some symbol s, attach a new node to the head of the linked list for s containing the current time t and remove the tail node of s. To determine the new order, sort the time given in the new tail into the time sorted list L.

5.2.4 Other Dynamic Update Algorithms

We briefly mention a few other dynamic update algorithms. As one has probably guessed by now, there are a very large number of dynamic update algorithms, each with its own features. The ones mentioned here are some of the more popular ones that have not been analysed with respect to steady state cost and overwork.

No formulas for Transpose are known. In part, this is because requests that occurred an arbitrarily long time ago can still be influencing all current list positions, and not, say, just the relative positions of only 2 items. The original M1FF has the same difficulty, as does M1FF2 and Frequency Count. For Frequency Count though, one can come up with formulas based on the Best x of $2x - 1$ family. Set the x parameter to more than the number of requests m (actually $2x - 1 \geq m$), and Best

x of $2x-1$ behaves the same as Frequency Count, with two cautions. Frequency Count is usually started with frequencies of zero, which is not the same as starting Best x of $2x-1$ by conditioning it on the uniform distribution. Also, Frequency Count's behavior when ties occur is usually minimum movement, meaning that if the frequency of 'a' is one greater than 'b' (and so 'a' is in front of 'b'), and a request for 'b' occurs, 'a' remains in front of 'b'. Best x of $2x-1$ can, of course, be conditioned with an arbitrary request sequence in which the alphabet is listed x times as described in the initialization of Best x of $2x-1$'s list, in which case for $2x-1 \geq m$, Best x of $2x-1$ will behave the same as a Frequency Count that resolves ties by putting the more recently requested items closer to the front.

Even when fairly simple equations do exist, simulations and experiments are an important way of learning about the properties of various algorithms. Bachrach and El-Yaniv experimented with many dynamic update algorithms [5], such as Move-Fraction(k), which moves the requested item forward a fraction (half, for instance) of its position in the list. Move-Fraction(k) is a popular and obvious choice when first dreaming up alternates to MTF. Many of the algorithms Bachrach and El-Yaniv tested beat MTF in Burrows-Wheeler compression on various test files. This demonstrates that perhaps MTF is not a very good strategy.

With equations for several dynamic update algorithms in hand, comparisons other than the empirical can be made. In the following sections, we compare Best x of $2x-1$ to Best $x+1$ of $2x+1$, which covers Best 1 of 1 (MTF) and Best 2 of 3.

5.3 Comparing Best x of $2x-1$ to Best $x+1$ of $2x+1$

Best x of $2x-1$ always has higher asymptotic cost than Best $x+1$ of $2x+1$. Overwork is not as simple. Eventually (for high enough t), the overwork of Best x is always smaller (closer to 0) than Best $x+1$. But as t increases, overwork goes to 0 anyway. We are more interested in the overwork when t is small. Overwork of Best x is more likely to be smaller for any value of t, but just as overwork can be negative, overwork can be smaller for Best x and for Best $x+2$ than for Best $x+1$ at low t.

One can use similar methods to show that M1FF3 has lower asymptotic cost than

MTF (Best 1 of 1).

Theorem 12 *The asymptotic cost on a distribution P of Best $x+1$ of $2x+1$ is less than or equal to the asymptotic cost on P of Best x of $2x-1$ for all $x \geq 1$, with equality if and only if $p_i = p_j$ for all i, j.*

Proof: The asymptotic cost of Best x of $2x-1$ is

$$1 + \sum_{i=1}^{n} p_i \sum_{j \neq i} (p_i + p_j)^{1-2x} \sum_{w=0}^{x-1} \binom{2x-1}{w} p_i^w p_j^{2x-1-w}$$

and the asymptotic cost of Best $x+1$ of $2x+1$ is

$$1 + \sum_{i=1}^{n} p_i \sum_{j \neq i} (p_i + p_j)^{-1-2x} \sum_{w=0}^{x} \binom{2x+1}{w} p_i^w p_j^{2x+1-w}$$

Let $\overline{w} = 2x - 2 - w$. The part of the asymptotic cost we concentrate upon is the cost of some two symbols l_i and l_j which, for Best x of $2x-1$ is

$$\begin{aligned}
& p_i(p_i+p_j)^{1-2x} \sum_{w=0}^{x-1} \binom{2x-1}{w} p_i^w p_j^{\overline{w}+1} + p_j(p_i+p_j)^{1-2x} \sum_{w=0}^{x-1} \binom{2x-1}{w} p_j^w p_i^{\overline{w}+1} \\
= \quad & p_i p_j (p_i+p_j)^{1-2x} \sum_{w=0}^{x-1} \binom{2x-1}{w} p_i^w p_j^{\overline{w}} + p_j p_i (p_i+p_j)^{1-2x} \sum_{w=0}^{x-1} \binom{2x-1}{w} p_j^w p_i^{\overline{w}} \\
= \quad & p_i p_j (p_i+p_j)^{1-2x} \sum_{w=0}^{x-1} \binom{2x-1}{w} (p_i^w p_j^{\overline{w}} + p_j^w p_i^{\overline{w}})
\end{aligned}$$

and, for Best $x+1$ of $2x+1$ is

$$p_i p_j (p_i+p_j)^{-1-2x} \sum_{w=0}^{x} \binom{2x+1}{w} (p_i^w p_j^{2x-w} + p_j^w p_i^{2x-w})$$

With some work on the above part of the asymptotic cost for Best x of $2x-1$ we show that it is always more than that of Best $x+1$ of $2x+1$.

$$(p_i+p_j)^{1-2x} \sum_{w=0}^{x-1} \binom{2x-1}{w} (p_i^w p_j^{\overline{w}} + p_j^w p_i^{\overline{w}})$$

$$= \quad (p_i+p_j)^{-1-2x}\sum_{w=0}^{x-1}\binom{2x-1}{w}(p_i^2+2p_ip_j+p_j^2)(p_i^w p_j^{\overline{w}} + p_j^w p_i^{\overline{w}})$$

Expanding the summation, we have

$$\begin{array}{lllllll}
\vdots & & & & & & \\
\binom{2x-1}{x-2} & (p_i^{x+2}p_j^{x-2}+ & 2p_i^{x+1}p_j^{x-1}+ & p_i^x p_j^x)+ & & & \\
\binom{2x-1}{x-1} & & (p_i^{x+1}p_j^{x-1}+ & 2p_i^x p_j^x+ & p_i^{x-1}p_j^{x+1})+ & & \\
\binom{2x-1}{x-1} & & (p_i^{x+1}p_j^{x-1}+ & 2p_i^x p_j^x+ & p_i^{x-1}p_j^{x+1})+ & & \\
\binom{2x-1}{x-2} & & & (p_i^x p_j^x+ & 2p_i^{x-1}p_j^{x+1}+ & p_i^{x-2}p_j^{x+2})+ & \\
\binom{2x-1}{x-3} & & & & (p_i^{x-1}p_j^{x+1}+ & 2p_i^{x-2}p_j^{x+2}+ & p_i^{x-3}p_j^{x+3})+ \\
\vdots & & & & & & \\
\binom{2x-1}{3} & (p_i^5p_j^{2x-5}+ & 2p_i^4p_j^{2x-4}+ & p_i^3p_j^{2x-3})+ & & & \\
\binom{2x-1}{2} & & (p_i^4p_j^{2x-4}+ & 2p_i^3p_j^{2x-3}+ & p_i^2p_j^{2x-2})+ & & \\
\binom{2x-1}{1} & & & (p_i^3p_j^{2x-3}+ & 2p_i^2p_j^{2x-2}+ & p_ip_j^{2x-1})+ & \\
\binom{2x-1}{0} & & & & (p_i^2p_j^{2x-2}+ & 2p_ip_j^{2x-1}+ & p_j^{2x})
\end{array}$$

Since $\binom{a-1}{b-1}+\binom{a-1}{b}=\binom{a}{b}$, each column except the middle 3 sums to

$$\begin{aligned}
&\left[\binom{2x-1}{w-2}+2\binom{2x-1}{w-1}+\binom{2x-1}{w}\right]p_i^w p_j^{2x-w}\\
=\;&\binom{2x+1}{w}p_i^w p_j^{2x-w}
\end{aligned}$$

This holds for the incomplete outer columns because $\binom{a}{b}=0$ for $b<0$ or $b>a$.

So the asymptotic cost of Best x of $2x-1$ is equal to the asymptotic cost of Best $x+1$ of $2x+1$ plus a low positive amount which decreases as x increases.

$$(p_i+p_j)^{1-2x}\sum_{w=0}^{x-1}\binom{2x-1}{w}(p_i^w p_j^{\overline{w}} + p_j^w p_i^{\overline{w}})$$

$$
\begin{aligned}
&= (p_i+p_j)^{-1-2x}\sum_{w=0}^{x}\binom{2x+1}{w}(p_i^w p_j^{2x-w}+p_j^w p_i^{2x-w}) + \\
&\quad \binom{2x-1}{x-1}(p_i^{x+1}p_j^{x-1}-2p_i^x p_j^x+p_i^{x-1}p_j^{x+1}) \\
&= (p_i+p_j)^{-1-2x}\sum_{w=0}^{x}\binom{2x+1}{w}(p_i^w p_j^{2x-w}+p_j^w p_i^{2x-w}) + \binom{2x-1}{x-1}(p_ip_j)^{x-1}(p_i-p_j)^2 \\
&> (p_i+p_j)^{-1-2x}\sum_{w=0}^{x}\binom{2x+1}{w}(p_i^w p_j^{2x-w}+p_j^w p_i^{2x-w}) \quad \text{if } p_i \neq p_j
\end{aligned}
$$

▮

Thus, the asymptotic cost of Best x of $2x-1$ is inversely proportional to x. Next we show that the overwork is usually proportional to x. Similar to the proof of asymptotic cost, we multiply the middle part of Best x by $(q_i^2+2q_iq_j+q_j^2)/(q_i+q_j)^2$, and compare to Best $x+1$. Thus, we can account for most of the terms in Best x. The few unmatched terms are more than balanced by the many unmatched terms in Best $x+1$ when t or x is sufficiently large, so that eventually the overwork on a distribution P is always related as follows: $|OV_{Bx+1}(P,t)| \leq |OV_{Bx}(P,t)|$. But overwork tends to 0 as t grows, and adaptation is too slow at large x, so this is not particularly interesting.

When t and x are small, matters are not so clear cut. Imagine a weighted die and a history of the rolls it has made. The amount of history is dependent on x. At some unknown time, the die will be switched for a different weighted die. The obvious action, to achieve the best guesses shortly after the switch occurs, is to discard old history quickly. But this can actually be bad if the new die is similar to the old one, creating a situation in which the old history provides the right answers for the wrong reasons. Overwork can fluctuate on data drawn from alphabets as small as 3 letters.

It is easiest to examine the relationship between 2 symbols l_i and l_j. Upon switching from Q to P, the probabilities are 0.5 that l_i and l_j should be swapped. If l_i and l_j are already in the best order, then the probability is 0.5 (0.25 of the total) that they are more likely to be in that order under P than under Q. So in 0.75 of the switches, more rapid adaptation to P is beneficial. However, proving this is difficult.

We showed 0.75 probability of positive overwork for MTF in Chapter 2. The speed of the adaptation is proportional to x.

We point out that because $\binom{a}{b} = 0$ for $b > a$ or $b < 0$, the minimum functions in the overwork formula are not necessary. Specifically, $\sum_{w_1=0}^{\min(x-1,v)} \sum_{w_2=0}^{\min(x-1-w_1,\overline{v})}$ can instead be $\sum_{w_1=0}^{x-1} \sum_{w_2=0}^{x-1-w_1}$.

Theorem 13 *Given random probability distributions P and Q and overwork $OV_{Bx+1}(Q,0) = 0$ and $OV_{Bx}(Q,0) = 0$, the overwork of Best $x+1$ of $2x+1$ on P at time t for two symbols l_i and l_j, $OV_{Bx+1}(P,t,i,j)$, is greater than $OV_{Bx}(P,t,i,j)$ with probability > 0.5.*

Proof: The overwork of Best x of $2x-1$ is

$$OV_{Bx} = \sum_{i=1}^{n} p_i \sum_{j \neq i} \sum_{v=0}^{2x-2} \binom{t}{v} (1-p_i-p_j)^{t-v}$$
$$\left[(q_i+q_j)^{-\overline{v}} \sum_{w_1=0}^{\min(x-1,v)} \binom{v}{w_1} p_i^{w_1} p_j^{v-w_1} \sum_{w_2=0}^{\min(x-1-w_1,\overline{v})} \binom{\overline{v}}{w_2} q_i^{w_2} q_j^{\overline{v}-w_2} \right.$$
$$\left. -(p_i+p_j)^{-\overline{v}} \sum_{w=0}^{x-1} \binom{2x-1}{w} p_i^w p_j^{2x-1-w} \right]$$

where $\overline{v} = 2x-1-v$. The overwork of Best $x+1$ of $2x+1$ is

$$OV_{B(x+1)} = \sum_{i=1}^{n} p_i \sum_{j \neq i} \sum_{v=0}^{2x} \binom{t}{v} (1-p_i-p_j)^{t-v}$$
$$\left[(q_i+q_j)^{-\overline{v}-2} \sum_{w_1=0}^{\min(x,v)} \binom{v}{w_1} p_i^{w_1} p_j^{v-w_1} \sum_{w_2=0}^{\min(x-w_1,\overline{v}+2)} \binom{\overline{v}+2}{w_2} q_i^{w_2} q_j^{\overline{v}+2-w_2} \right.$$
$$\left. -(p_i+p_j)^{-\overline{v}-2} \sum_{w=0}^{x} \binom{2x+1}{w} p_i^w p_j^{2x+1-w} \right]$$

W.l.o.g. $p_i \geq p_j$. We move the p_i inside and rearrange some terms to get

$$OV_{Bx} = \sum_{i=1}^{n} \sum_{j \neq i} \sum_{v=0}^{2x-2} \binom{t}{v} (1-p_i-p_j)^{t-v}$$

$$\left[(q_i+q_j)^{-\overline{v}}\sum_{w_1=0}^{\min(x-1,v)}\sum_{w_2=0}^{\min(x-1-w_1,\overline{v})}\binom{v}{w_1}\binom{\overline{v}}{w_2}p_i^{w_1+1}p_j^{v-w_1}q_i^{w_2}q_j^{\overline{v}-w_2}\right.$$
$$\left.-p_ip_j(p_i+p_j)^{-\overline{v}}\sum_{w=0}^{x-1}\binom{2x-1}{w}p_i^w p_j^{2x-2-w}\right]$$

This quantity is the total cost minus the asymptotic cost, multiplied by $(1-p_i-p_j)^{t-v}$, a factor dependent on t. Let $\mathcal{Q}_{x,v}$ and $\mathcal{P}_{x,v}$ represent these quantities.

$$\begin{aligned}\mathcal{Q}_{x,v}(i,j) &= (q_i+q_j)^{-\overline{v}}\textstyle\sum_{w_1=0}^{\min(x-1,v)}\sum_{w_2=0}^{\min(x-1-w_1,\overline{v})}\binom{v}{w_1}\binom{\overline{v}}{w_2}\ (p_i^{w_1+1}p_j^{v-w_1}q_i^{w_2}q_j^{\overline{v}-w_2}+\\ &\qquad p_j^{w_1+1}p_i^{v-w_1}q_j^{w_2}q_i^{\overline{v}-w_2})\\ \mathcal{P}_{x,v}(i,j) &= p_ip_j(p_i+p_j)^{-\overline{v}}\textstyle\sum_{w=0}^{x-1}\binom{2x-1}{w}(p_i^w p_j^{2x-2-w}+p_j^w p_i^{2x-2-w})\end{aligned}$$

We will show that the signs of $\mathcal{Q}$ and $\mathcal{P}$ depend on p_i, p_j, q_i, and q_j. W.l.o.g. $p_i \geq p_j$. Since overwork is 0 if $p_i = p_j$, we concentrate on $p_i > p_j$.

	$q_i \leq q_j$	$q_j < q_i \leq \frac{p_i}{p_j}q_j$	$\frac{p_i}{p_j}q_j < q_i$
probability	0.5	0.25	0.25
$-\mathcal{P}_{x+1,v}(i,j)+\mathcal{P}_{x,v}(i,j)$	$+$	$+$	$+$
$\mathcal{Q}_{x+1,v}(i,j)-\mathcal{Q}_{x,v}(i,j)$	$+$	$-$	$-$
$\mathcal{Q}_{x,v}(i,j)-\mathcal{P}_{x,v}(i,j)$	$+$	$+?$	$-?$

From the previous proof, which compares asymptotic costs of Best x of $2x-1$ to Best $x+1$ of $2x+1$, we know that $\mathcal{P}_{x,v}(i,j) \geq \mathcal{P}_{x+1,v}(i,j)$ with equality only if P is uniform. So $-\mathcal{P}_{x,v}(i,j) \leq -\mathcal{P}_{x+1,j}(i,j)$.

For each v from 0 to $2x-2$, we examine the difference between $\mathcal{Q}_{x+1,v}(i,j)$ and $\mathcal{Q}_{x,v}(i,j)$. We expand the summations in $\mathcal{Q}_{x,v}(i,j)$ for Best x. For each value of v

from 0 to $2x-2$, we get

$$\begin{array}{llllllllllll}
\binom{v}{0} & \binom{\overline{v}}{0} & (p_i^1 & p_j^v & q_i^0 & q_j^{\overline{v}} & + & p_j^1 & p_i^v & q_j^0 & q_i^{\overline{v}} &)+ \\
\binom{v}{0} & \binom{\overline{v}}{1} & (p_i^1 & p_j^v & q_i^1 & q_j^{\overline{v}-1} & + & p_j^1 & p_i^v & q_j^1 & q_i^{\overline{v}-1} &)+ \\
 & \vdots & & & & & & & & & & \\
\binom{v}{0} & \binom{\overline{v}}{x-1} & (p_i^1 & p_j^v & q_i^{x-1} & q_j^0 & + & p_j^1 & p_i^v & q_j^{x-1} & q_i^0 &)+ \\
\binom{v}{1} & \binom{\overline{v}}{0} & (p_i^2 & p_j^{v-1} & q_i^0 & q_j^{\overline{v}} & + & p_j^2 & p_i^{v-1} & q_j^0 & q_i^{\overline{v}} &)+ \\
 & \vdots & & & & & & & & & & \\
\binom{v}{1} & \binom{\overline{v}}{x-2} & (p_i^2 & p_j^{v-1} & q_i^{x-2} & q_j^0 & + & p_j^2 & p_i^{v-1} & q_j^{x-2} & q_i^0 &)+ \\
\vdots & & & & & & & & & & & \\
\binom{v}{v-1} & \binom{\overline{v}}{0} & (p_i^v & p_j^1 & q_i^0 & q_j^{\overline{v}} & + & p_j^v & p_i^1 & q_j^0 & q_i^{\overline{v}} &)+ \\
\binom{v}{v-1} & \binom{\overline{v}}{1} & (p_i^v & p_j^1 & q_i^1 & q_j^{\overline{v}-1} & + & p_j^v & p_i^1 & q_j^1 & q_i^{\overline{v}-1} &)+ \\
\binom{v}{v} & \binom{\overline{v}}{0} & (p_i^{v+1} & p_j^0 & q_i^0 & q_j^{\overline{v}} & + & p_j^{v+1} & p_i^0 & q_j^0 & q_i^{\overline{v}} &)
\end{array}$$

Similar to the proof for asymptotic cost, we multiply $\mathcal{Q}_{x,v}(i,j)$ by $(q_i+q_j)^{-2}(q_j^2+2q_iq_j+q_i^2)$ and subtract from $\mathcal{Q}_{x+1,v}(i,j)$ And, as in the previous proof, we use the properties of the binomials:

$$\begin{aligned}
& \left[\binom{\overline{v}}{w_2}+2\binom{\overline{v}}{w_2-1}+\binom{\overline{v}}{w_2-2}\right] q_i^{w_2} q_j^{\overline{v}+2-w_2} \\
= \quad & \binom{\overline{v}+2}{w_2} q_i^{w_2} q_j^{\overline{v}+2-w_2}
\end{aligned}$$

Also

$$\binom{v}{w} = \sum_{k=0}^{\min(c,v-c)} \binom{c}{k}\binom{v-c}{\min(w,v-w)-k}$$

Most of the terms cancel out, leaving

$$\begin{array}{rlllllllllll}
\binom{v}{0}\binom{\overline{v}}{x} & (& p_i^1 & p_j^v & q_i^x & q_j^{\overline{v}} & + & p_j^1 & p_i^v & q_j^x & q_i^{\overline{v}} & \\
 & - & p_i^0 & p_j^{v+1} & q_i^x & q_j^{\overline{v}} & - & p_j^0 & p_i^{v+1} & q_j^x & q_i^{\overline{v}} &)+ \\
\binom{v}{1}\binom{\overline{v}}{x-1} & (& p_i^2 & p_j^{v-1} & q_i^{x-1} & q_j^{\overline{v}+1} & + & p_j^2 & p_i^{v-1} & q_j^{x-1} & q_i^{\overline{v}+1} & \\
 & - & p_i^1 & p_j^v & q_i^{x-1} & q_j^{\overline{v}+1} & - & p_j^1 & p_i^v & q_j^{x-1} & q_i^{\overline{v}+1} &)+ \\
\binom{v}{2}\binom{\overline{v}}{x-2} & (& p_i^3 & p_j^{v-2} & q_i^{x-2} & q_j^{\overline{v}+2} & + & p_j^3 & p_i^{v-2} & q_j^{x-2} & q_i^{\overline{v}+2} & \\
 & - & p_i^2 & p_j^{v-1} & q_i^{x-2} & q_j^{\overline{v}+2} & - & p_j^2 & p_i^{v-1} & q_j^{x-2} & q_i^{\overline{v}+2} &)+ \\
\vdots & & & & & & & & & & & \\
\binom{v}{x}\binom{\overline{v}}{0} & (& p_i^{x+1} & p_j^0 & q_i^0 & q_j^{x+1} & + & p_j^{x+1} & p_i^0 & q_j^0 & q_i^{x+1} & \\
 & - & p_i^x & p_j^1 & q_i^0 & q_j^{x+1} & - & p_j^x & p_i^1 & q_j^0 & q_i^{x+1} &)
\end{array}$$

Each of the polynomials with more j's matches one with more i's, and vice-versa. All are paired off to get

$$\begin{aligned}
& c((p_i - p_j)q_i^{x-w_2}q_j^{\overline{v}+w_2} + (p_j - p_i)q_j^{x-w_2}q_i^{\overline{v}+w_2}) \\
= \; & c((p_i - p_j)(q_i^{x-w_2}q_j^{\overline{v}+w_2} - q_j^{x-w_2}q_i^{\overline{v}+w_2}))
\end{aligned}$$

for some w_2 and $c \geq 0$, which is non-negative when $q_i \leq q_j$ which occurs with probability 0.5. This holds for all v. ▮

Conjecture 1 *With probability 0.75, the overwork of Best x of $2x-1$ on random probability distribution P at time t for two symbols l_i and l_j is non-negative. $OV_{Bx}(P,t,i,j) \geq 0$ with probability 0.75.*

We continue examining the formulae from the previous proof to show how the conjecture of positive overwork with probability 0.75 might be proven. We conjecture that for each value of of v from 0 to $2x-2$ the probability is 0.75 that $\mathcal{Q}_{x,v}(i,j) - \mathcal{P}_{x,v}(i,j)$ is non-negative. That is,

$$(q_i + q_j)^{-\overline{v}-2} \sum_{w_1=0}^{x} \sum_{w_2=0}^{\min(x-w_1,\overline{v}+2)} \binom{v}{w_1}\binom{\overline{v}+2}{w_2} p_i^{w_1+1} p_j^{v-w_1} q_i^{w_2} q_j^{\overline{v}+2-w_2}$$

$$\begin{aligned} & -p_i p_j (p_i+p_j)^{-\overline{v}-2} \sum_{w=0}^{x} \binom{2x+1}{w} p_i^w p_j^{2x-w} \\ > \ & 0 \end{aligned}$$

The degree of the polynomials is the same. That is, $w_1 + v - w_1 + w_2 + \overline{v} + 2 - w_2 = 2x+1 = w + 2x + 1 - w$. And the number of polynomials is the same.

$$\sum_{w=0}^{x} \binom{2x+1}{w} = 2^{x-1} = \sum_{w_1=0}^{x} \sum_{w_2=0}^{\min(x-1-w_1,\overline{v})} \binom{v}{w_1}\binom{\overline{v}+2}{w_2}$$

By the properties of the binomials, $\binom{2x-1}{w} = \sum_{w_3=0}^{w} \binom{v}{w_3}\binom{\overline{v}}{\overline{w_3}}$. When $q_i = \frac{p_i}{p_j} q_j$ it is easy to see $\mathcal{P}_{x,v}(i,j) = \mathcal{Q}_{x,v}(i,j)$.

$$\begin{aligned} & (q_i+q_j)^{-\overline{v}} (p_i^{w_3+1} p_j^{v-w_3} q_i^{\overline{w_3}} q_j^{\overline{v}-\overline{w_3}} + p_j^{w_3+1} p_i^{v-w_3} q_j^{\overline{w_3}} q_i^{\overline{v}-\overline{w_3}}) \\ = \ & (q_i+q_j)^{-\overline{v}} \left(p_i^{w+1} p_j^{v-w} \left(\frac{p_j}{p_i} q_i\right)^{\overline{w_3}} q_j^{\overline{v}-\overline{w_3}} + p_j^{w+1-\overline{v}} p_i^{\overline{w}} q_j^{\overline{w_3}} \left(\frac{p_j}{p_i} q_i\right)^{\overline{v}-\overline{w_3}} \right) \\ = \ & (q_i+q_j)^{-\overline{v}} \left(\frac{p_j}{q_j}\right)^{-\overline{v}} \left(p_i^{w+1} p_j^{\overline{w}} \left(\frac{p_j}{p_i}\frac{q_i}{q_j}\right)^{\overline{w_3}} + p_j^{w+1} p_i^{\overline{w}} \left(\frac{p_j}{p_i}\frac{q_i}{q_j}\right)^{\overline{v}-\overline{w_3}} \right) \\ = \ & \left(p_j \frac{q_i}{q_j} + p_j\right)^{-\overline{v}} \left(p_i^{w+1} p_j^{\overline{w}} \left(\frac{p_j}{p_i}\frac{q_i}{q_j}\right)^{\overline{w_3}} + p_j^{w+1} p_i^{\overline{w}} \left(\frac{p_j}{p_i}\frac{q_i}{q_j}\right)^{\overline{v}-\overline{w_3}} \right) \\ = \ & (p_i+p_j)^{-\overline{v}} (p_i^{w+1} p_j^{\overline{w}} + p_j^{w+1} p_i^{\overline{w}}) \quad \text{if } q_i = \tfrac{p_i}{p_j} q_j \end{aligned}$$

When $q_i < \frac{p_i}{p_j} q_j$ (and $p_i > p_j$), for $w < x$ (and $v < 2x-1$), we need more to show that $\mathcal{P}_{x,v}(i,j) < \mathcal{Q}_{x,v}(i,j)$.

To represent all values of w_1 and w_2 in $\mathcal{Q}_{x,v}(i,j)$, we can make a triangular array with w_1 going from 0 to $x-1$ on one axis and w_2 doing the same on the other axis. The array is triangular because $w_1 + w_2 < x$. The elements can be grouped into triplets with $(w_1, w_2) = (0,0), (x-1,0), (0,x-1)$ and $(1,0), (x-2,1), (0,x-2)$ and so on. We believe these triplets sum to more than the matching parts of $\mathcal{P}_{x,v}(i,j)$ when $q_i < \frac{p_i}{p_j} q_j$. That is, for $(0,0), (x-1,0), (0,x-1)$:

$$
\begin{aligned}
& (q_i+q_j)^{-\overline{v}}\left[\binom{v}{0}\binom{\overline{v}}{0}(p_ip_j^v q_j^{\overline{v}}+p_jp_i^v q_i^{\overline{v}})+\right. \\
& \binom{v}{x-1}\binom{\overline{v}}{0}(p_i^x p_j^{v-x+1}q_j^{\overline{v}}+p_j^x p_i^{v-x+1}q_i^{\overline{v}})+ \\
& \left.\binom{v}{0}\binom{\overline{v}}{x-1}(p_ip_j^v q_i^{x-1}q_j^{\overline{v}-x+1}+p_jp_i^v q_i^{x-1}q_j^{\overline{v}})\right] \\
> & (p_i+p_j)^{-\overline{v}}\left[\binom{v}{0}\binom{\overline{v}}{0}(p_ip_j^{2x-1}+p_jp_i^{2x-1})+\right. \\
& \left.\left(\binom{v}{x-1}\binom{\overline{v}}{0}+\binom{v}{0}\binom{\overline{v}}{x-1}\right)(p_i^xp_j^x+p_j^xp_i^x)\right]
\end{aligned}
$$

The final part we conjecture is that when $q_i < \frac{p_i}{p_j}q_j$, the overwork is greater when x is greater. Specifically,

$$
\begin{aligned}
& \mathcal{Q}_{x+1,v}(i,j)-\mathcal{P}_{x+1,v}(i,j)-(\mathcal{Q}_{x,v}(i,j)-\mathcal{P}_{x,v}(i,j)) \\
= & \mathcal{Q}_{x+1,v}(i,j)-\mathcal{Q}_{x,v}(i,j)-\mathcal{P}_{x+1,v}(i,j)+\mathcal{P}_{x,v}(i,j) \\
> & 0\ ?
\end{aligned}
$$

And we conjecture that if $q_i > \frac{p_i}{p_j}q_j$, then overwork, which is negative, is less when x is greater.

Conjecture 2 *Given random probability distributions P and Q and overwork $OV_{Bx+1}(Q,0)=0$ and $OV_{Bx}(Q,0)=0$, the magnitude of the overwork of Best $x+1$ of $2x+1$ on P at time t for two symbols l_i and l_j is greater than that of Best x of $2x-1$. $|OV_{Bx+1}(P,t,i,j)| > |OV_{Bx}(P,t,i,j)|$.*

5.4 Costs on Zipf distribution

In this section, we give some data generated from the asymptotic cost and overwork formulae. In all cases, the cost function used is $c=\rho(l)$. As is traditional, the graphs

Algorithm	Cost	Ratio
Move-To-Front	2208.68	1.386
Move 1 From Front 3	2208.67	1.386
Best 2 of 3	1901.33	1.193
Best 3 of 5	1801.75	1.131
Best 4 of 7	1751.96	1.099
Best 5 of 9	1721.87	1.081
Best 6 of 11	1701.64	1.068
Best 7 of 13	1687.07	1.059
Best 8 of 15	1676.04	1.052
Best 9 of 17	1667.40	1.046
Optimal	1593.57	1

Table 5.2: Asymptotic cost on Zipf distribution

show time t (number of characters) on the x axis and cost c on the y axis.

Move-To-Front costs 1.386 times optimal on an i.i.d. source over a large alphabet distributed according to Zipf's law: $p_i = i^{-1}H_n^{-1}$ [40]. The asymptotic costs and ratio to optimal at $n = 16384$, which is large enough for our calculations of MTF's cost to reach 1.386 times optimal, for the algorithms with known equations are given in Table 5.2. There is not much difference between MTF and M1FF3 at such a large value of n. The ratio of Best x of $2x-1$'s cost to optimal approaches 1 as x increases.

The total cost (overwork plus asymptotic) for the algorithms when switching from the uniform distribution to a Zipf distribution, for $n = 64$ is in Figure 5.1. As can be seen, MTF and M1FF3 are very similar at this value of n.

The total cost when switching from a Zipf to a different Zipf distribution, for one of the 23 possibilities for $n = 4$ is graphed in Figure 5.2. The overwork in these cases is symmetric, the same from P to Q as from Q to P. This value of n is small enough for the difference between M1FF3 and Best 1 of 1 to be significant.

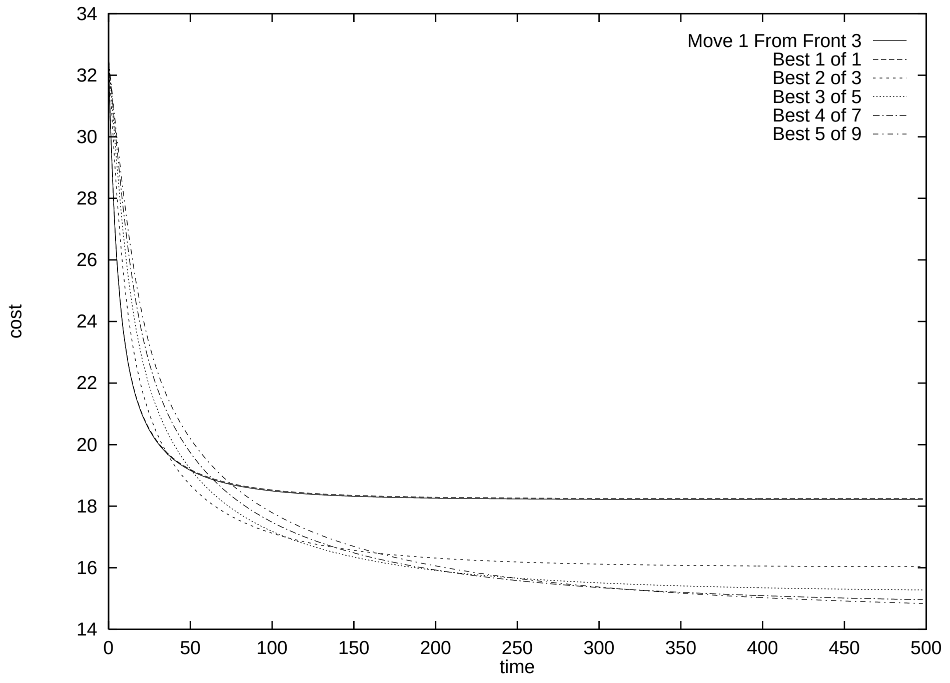


Figure 5.1: Total cost from uniform to Zipf distribution on alphabet of 64 symbols

5.5 Solo Performance of Dynamic Update Algorithms

Table 5.3 gives performance data of the various dynamic list updating algorithms on the Calgary Corpus and other files. The particular variations of the basic Burrows-Wheeler compression scheme used to produce the statistics in this chapter are the Gray code sort and a better order for only the text files [15], and an adaptive arithmetic coder similar to the one presented by Balkenhol, Kurtz, and Shtarkov [7]. It codes all 8 bit characters rather than only the ones actually used in the data. Each file was encoded in one block.

5.6 Combining two algorithms

The ideas for getting more compression by switching between the outputs of two universal source coding algorithms, presented by Volf and Willems [50], can be used to get higher compression of the p.i.i.d. data. In Volf and Willems' algorithm, an

			Best x of $2x-1$			
File	M1FF	MTF	2	3	4	5
bib	**26,886**	26,979	28,400	29,477	30,336	31,038
book1	224,018	225,980	**223,729**	224,541	226,107	227,719
book2	**153,034**	153,968	157,089	160,143	163,204	165,617
geo	57,197	57,386	56,853	56,333	56,073	**55,297**
news	**116,753**	117,240	123,570	127,016	129,859	131,963
obj1	**10,517**	10,579	10,945	11,248	11,488	11,648
obj2	**75,454**	75,822	82,139	85,885	88,623	90,853
paper1	**16,250**	16,305	17,489	18,279	18,920	19,394
paper2	**24,626**	24,708	25,570	26,237	26,856	27,388
pic	48,222	49,133	47,654	47,135	47,002	**46,932**
progc	**12,332**	12,368	13,324	13,968	14,508	14,895
progl	**15,287**	15,337	16,836	17,718	18,526	19,079
progp	**10,532**	10,539	11,991	12,800	13,324	13,817
trans	**17,349**	17,373	20,541	22,477	24,219	25,249
total	**808,457**	813,717	836,130	853,257	869,045	880,889
lena	582,732	583,003	**580,230**	582,377	585,697	589,164
lesms10	820,774	828,638	**816,364**	816,466	820,505	825,080

Table 5.3: Solo performance of dynamic update algorithms.

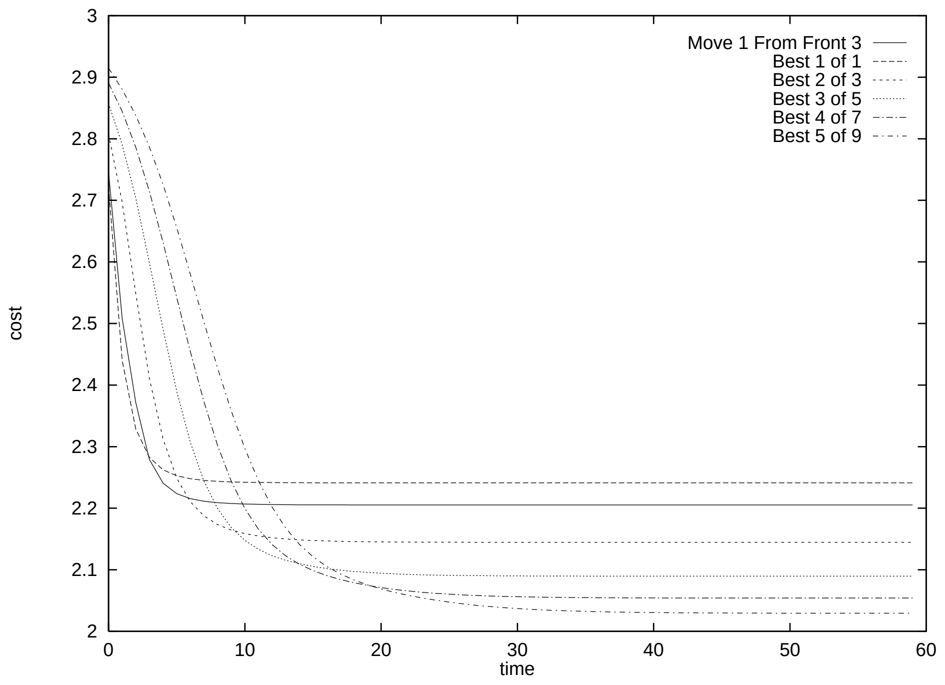


Figure 5.2: Total cost from $Q = \{0.12, 0.16, 0.24, 0.48\}$ to $P = \{0.48, 0.24, 0.16, 0.12\}$

estimator determines when it is profitable to switch between universal algorithms such as PPMD and LZ77. In BW compression, one can use the same technique to choose between two algorithms such as MTF and Best 2 of 3. With complementary algorithms, this can result in significant improvement.

The optimal switching algorithm is $O(m^2)$ which is not practical. Two variants, "snake", and "reduced complexity", (we call it "fat snake"), are linear in m. The switching algorithms can be described as state machines with the number of states dependent on m. Each state has two outgoing arcs and the arc chosen is based on whether the switching algorithm decided to stay with the current source coder for the next symbol, or switch to the other source coder. Switching decisions can be explicitly coded with bits. The switching algorithm makes decisions based on the cost to each of the source coders plus the cost of encoding the switching decision. The less switches are made, the more expensive a decision to switch becomes. These costs are computed with weighting functions which blend all decisions that lead to the same number of switches having occurred by a specific time into one state. So 5 switches over the course of 100 decisions will put the machine in the same state no matter which 5 of the 100 decisions were the switches. The snake algorithm further reduces the number of states from quadratic to linear by putting a ceiling on the number of switches the states can represent and wrapping around to a state representing 0 switches whenever that threshold is exceeded. A diagram of the states with a ceiling of 1 switch looks somewhat like a snake (or perhaps the path of arcs taken looks snake-like), hence the name. The higher the ceiling, the "fatter" the snake is.

By delaying the switch for one symbol, switches can be computed by the decoder. Thus, explicitly encoding when the switches occur is unnecessary and, in experiments, reduced the size of the compressed data.

The first attempt used the two extremes: MTF, the algorithm with the smallest overwork, with Transpose, the algorithm with the lowest asymptotic value. But since Transpose can be arbitrarily bad, a failure to change to MTF when desirable can significantly degrade the compression rate. Also, Transpose's speed of adaptation was so slow that it often had not adapted to a previous switch before a new switch occurred.

The first significant improvement used MTF and Best 2 of 3. Better yet was M1FF and Best 2 of 3. M1FF has overwork similar to MTF but better asymptotic performance. Since M1FF2 has a little less overwork than M1FF, using M1FF2 with Best 2 of 3 was even better.

This lead naturally to the idea of finding a competitive algorithm with lower steady state work than Best 2 of 3 to pair with M1FF2. Such an algorithm might not need to be 2-competitive to complement M1FF2. Modifications of the algorithm to perform part of Timestamp, which became the Best x of $2x - 1$ family, gave yet more improvement. Results of "snaking" between M1FF2 and various members of Best x of $2x - 1$ are given in Table 5.4 using the same BWT and entropy coder as in Table 5.3.

All the switching algorithms weigh based on the probabilities each of the two algorithms assigns to the symbol. For each symbol, new weights are computed from old ones by multiplying them by the probabilities each algorithm assigns to that symbol at that time. Thus, the weights must be normalized frequently or they will go to zero. The first implementation of the snake algorithm used single precision floating point values. When that was changed to double precision, compression improved slightly. Because the significant part of the values is the exponent, computing with logarithms of those values yields increased accuracy for the same machine precision.

The first trial of the snake algorithm did not use the actual probabilities generated by the arithmetic coder backends of each of the two algorithms. Instead, a simple coder was used to generate probabilities for both algorithms. This, of course, meant that if both dynamic update algorithms output the same value, the snake algorithm decided not to switch. When the scheme performed well, we naturally thought to get even better performance by using the actual probabilities generated by the arithmetic coder back ends of each algorithm. Instead, the performance was worse. So the results in Table 5.4 are those of the simple scheme, which performed better in practice. In the snake algorithm, more accurate probabilities should increase compression. Instead compression actually decreased, and it is not clear why. Perhaps more accurate computation is needed, or the snake needs to be "fatter", that is, contain more states so that less mistakes are made on switching or not switching. Delaying switches for one symbol also hurts compression and compromises the accuracy of the statistics, but in experiments, explicitly coding the switches and not delaying was worse. The $O(m^2)$ weighting algorithm always makes the best decisions on when to switch and one would expect higher compression from more accurate statistics since the switches are optimal. The snake algorithm is $O(m)$ but pays for that speed with occasional bad decisions on when to switch or not, and so more accurate statistics may not always increase compression.

An idea for further improvement would be to switch between more than 2 algorithms. Start with a version of M1FF, then switch to Best 2 of 3, then Best 3 of 5, and so on. Eventually, restart the cycle with M1FF when a new piece of the BWT output is detected.

The pieces of independent identically distributed (i.i.d.) data are not independent (p.i.i.d., not i.p.i.d.). Assuming that detecting the piece boundaries of the BWT output presents no difficulty, care must be taken in the use of that knowledge. Resetting

file	Best x of $2x-1$			
	2	3	4	5
bib	**26,927**	26,953	27,005	27,065
book1	222,121	220,971	220,614	**220,470**
book2	153,128	152,804	**152,796**	152,840
geo	56,744	56,458	56,263	**56,091**
news	117,022	**116,955**	117,083	117,169
obj1	**10,596**	10,649	10,671	10,683
obj2	**76,158**	76,424	76,623	76,733
paper1	**16,295**	16,300	16,334	16,336
paper2	24,615	24,616	**24,588**	24,618
pic	47,860	47,347	47,174	**47,066**
progc	**12,409**	12,429	12,429	12,439
progl	15,362	15,378	**15,350**	15,373
progp	**10,550**	10,557	10,558	10,582
trans	**17,380**	**17,380**	17,403	17,395
total	807,167	805,221	804,891	**804,860**
lena	**579,528**	579,746	580,341	580,943
lesms10	812,187	806,577	804,875	**804,265**

Table 5.4: Snake algorithm switching between M1FF2 and Best x of $2x-1$.

the statistics of the entropy coder whenever a piece boundary is crossed can worsen the amount of compression because often the previous piece is similar to the current piece. The switching method does not explicitly detect piece boundaries. Rather, the switching method weighs the cost of each algorithm's choice plus the cost of encoding a switch. Finding better places to switch between multiple algorithms, whether computed by weighting techniques or by some other method, can certainly increase the amount of compression. Also, better dynamic updating algorithms may yet be found.

5.7 Entropy Coder

The final stage of Burrows Wheeler Compression is the entropy coder. In general, entropy coders count the frequency of each symbol and use those statistics to generate optimal length (shortest possible) codes. Static versions count the frequencies before computing the lengths of the codes. Similar to the Frequency Count algorithm, adaptive versions adjust the lengths on-line, incrementing frequencies as symbols are received.

The entropy coder may be based on Huffman coding, arithmetic coding, or one of the many variants. To increase compression for Huffman codes, symbols may be blocked together, meaning there are codes for all pairs or all triples, etc. of the symbols. Large blocks are used in theoretical work, since coding inefficiencies due to using integral numbers of bits vanishes as the blocksize increases. However, this is of limited use in practice, as the code size increases exponentially with the block size. Arithmetic coding variants are legion. Some trade a little compression efficiency for speed, such as the idea of keeping the total frequency count a power of 2.

Most have some means of dealing with overflowing counts. The usual method is to divide all the frequency counts by 2, which reduces the coding efficiency slightly. But if the data is not identically distributed, reducing the counts is beneficial. A sliding window may be used. After the window is filled, new symbols are added to one end and removed from the other as in a queue. The total count is then the size of the window, which, to gain speed, should be a power of 2. Smaller windows mean poorer statistics but faster adjustments to changes in the data.

The exponentially increasing frequency count is an attempt to adjust to changing statistics. Instead of incrementing each frequency by 1, the increments are by $(1+c)^t$ where c is some value greater than 0. The larger c is, the more rapidly the statistics will be adjusted but also the poorer they will be.

As noted earlier, the output of Move-To-Front and the other List Update algorithms can be treated like ordered piecewise independent identically ditributed (o.p.i.i.d.) data. o.p.i.i.d. is piecewise independent identically distributed and ordered by frequency of occurrence. Although the exact probability of a symbol changes from piece to piece, its order by frequency relative to the other symbols does not change. While the assumption of o.p.i.i.d is not exactly correct, it is accurate enough to give valuable insight into the coding process. A standard entropy coder is within 1 bit of optimal for independent identically distributed (i.i.d.) data, but not for o.p.i.i.d. data.

The first improvement on standard entropy coding comes from the original BurrowsWheeler paper [13]. Run-length encode '0', the most common symbol, similar to the following: When encountering 3 or more '0's in a row, encode the 3 '0's normally and then encode the number of '0's following the initial 3.

The next improvement is Fenwick's modification [21] which is just as efficient as a standard coder but can adjust more quickly. Rather than encode each symbol separately, with the implied model of independence of symbols, break a symbol value x into 2 parts y and z where $2^y + z = x$ and $0 \geq z < 2^y$, and code using the statistics of the y's and z's rather than the x's. Experiments with other powers, specifically the golden ratio, where $\lceil 1.61^y \rceil + z = x$ and $0 \geq z < \lceil 1.61^y \rceil$, gave very similar compression ratios.

Balkenhol and co-authors [6, 7, 8] replace the run-length scheme with a Markov model. The Markov model tracks the last few values output by MTF as follows. The run-length encoder codes 0's only. The Markov model is order 3 and tracks 0's, 1's, and 2+'s for a total of 27 models. If a $z \geq 2$ is encoded, $z-2$ is passed to a Fenwick modified entropy coder. In their latest work, they merge the most similar among the 27 models, which causes counts to increment faster, for more accuracy and faster adaptation, and thus higher compression.

There are some other minor improvements worth mentioning. Balkenhol and co-authors suggest counting the number of symbols actually present in the data, rather than assuming 256 (for 8 bit data), and adjusting the entropy coder so that no coding space is wasted on symbols that do not occur. Initializing the symbol counts in the entropy coder to reflect a Zipf distribution rather than a uniform one, as is traditional, also saves a little space.

A suggestion for more improvement is to extend Fenwick's idea further. Break x into y and z as before, then, for sufficiently large z, break z into two parts as was done with x. This could be continued as long as the second part is sufficiently large. Effectively, one would be separately counting and encoding each 1 bit of the binary representation of x.

CHAPTER 6

Conclusion

Higher compression of the output of the Burrows-Wheeler Transform has been the focus of most of the Burrows-Wheeler research following the seminal paper. Since Move-To-Front performs relatively poorly as a stand-alone compression algorithm on data in general, it seemed reasonable that something better for BWT output could be easily discovered. Although algorithms such as Arnavut's Inversion Coding and interval coding have shown possibly better results, the original method of some dynamic list updating algorithm, such as Move 1 From Front, followed by an arithmetic coder, remains viable.

From a theoretic standpoint, proofs of BW compression's optimality suffer from the lack of tight bounds on MTF's performance. The proofs for the dictionary based algorithms are tight. But all the proofs are for sources producing stationary, ergotic data, a subset of sources producing arbitrary data. Much "real world" data is not stationary and ergotic. In the future, new kinds of sources may be defined for which proofs of optimality can be devised.

This work presented and analysed several techniques for getting higher compression from the Burrows-Wheeler Transform (BWT) without increasing the $O(m)$ time complexity of the method. No improvements in speed were presented. The ideas that increased the amount of compression were finding a better alphabet order, performing binary reflected Gray code sort instead of standard sorting in the BWT, and switching between Move 1 From Front and Best x of $2x - 1$ to encode BWT output.

The Best x of $2x - 1$ family of algorithms for dynamically updating lists was presented and analysed with respect to aymptotic cost and overwork. Also analysed with respect to overwork were Move-To-Front, Timestamp, and Move 1 From Front 3. We observed that Move-To-Front is Best 1 of 1 and part of Timestamp is Best 2 of 3. Comparing Best 1 of 1 to Best 2 of 3 showed Best 1 of 1 has less overwork and Best 2 of 3 has lower asymptotic cost. Although the goal was higher compression with the BWT, the results demonstrate that, like Competitive Analysis, analysis of the overwork is a worthwhile direction from which to approach List Update and similar problems.

BIBLIOGRAPHY

[1] Susanne Albers. Improved Randomized On-Line Algorithms for the List Update Problem. *SIAM J. Comput.*, 27:682–693, 1998.
`http://www.mpi-sb.mpg.de/~albers/`.

[2] Susanne Albers and Michael Mitzenmacher. Average Case Analysis of List Update Algorithms, with Applications to Data Compression. *Algorithmica.*, 21(3):312–329, 1998.
`http://www.mpi-sb.mpg.de/~albers/`.

[3] Arne Andersson and Stefan Nilsson. A new efficient radix sort. *Proc. 35th Annual IEEE Symposium on Foundations of Computer Science*, pages 714–721, 1994.
`http://www.nada.kth.se/~snilsson/public/papers/radix/index.html`.

[4] Ziya Arnavut. Move-To-Front and Inversion Coding. *Proc. Data Compression Conf.*, pages 193–202, 2000.

[5] Ran Bachrach and Ran El-Yaniv. Online List Accessing Algorithms and Their Applications: Recent Empirical Evidence. *Proc. of the 8th ACM-SIAM Symposium on Discrete Algorithms*, pages 53–62, 1997.

[6] Bernhard Balkenhol and Stefan Kurtz. Universal Data Compression Based on the Burrows and Wheeler-Transformation: Theory and Practice. Sonderforschungsbereich: Diskrete Strukturen in der Mathematik 98-069, Universitat Bielefeld, 1998. To appear in *IEEE Transactions on Computers.*
`http://www.mathematik.uni-bielefeld.de/~bernhard/`.

[7] Bernhard Balkenhol, Stefan Kurtz, and Yuri M. Shtarkov. Modifications of the Burrows and Wheeler Data Compression Algorithm. *Proc. Data Compression Conf.*, pages 188–197, 1999.
`http://www.mathematik.uni-bielefeld.de/~bernhard/`.

[8] Bernhard Balkenhol and Yuri M. Shtarkov. One attempt of a compression algorithm using the BWT.
`http://www.mathematik.uni-bielefeld.de/~bernhard/`.

[9] Bentley, Sleator, Tarjan, and Wei. A Locally Adaptive Data Compression Scheme. *Communications of the ACM*, 29(4):320–330, April 1986.

[10] J. Bentley and R. Sedgewick. Fast Algorithms for Sorting and Searching Strings. *Proc. 8th ACM-SIAM Symposium on Discrete Algorithms*, pages 360–369, 1997.

[11] James R. Bitner. Heuristics that Dynamically Organize Data Structures. *SIAM J. Comput.*, 8:82–110, 1979.

[12] Allan Borodin and Ran El-Yaniv. *Online Computation and Competitive Analysis.* Cambridge University Press, 40 West 20th Street, New York, NY 10011-4211, USA, 1998. ISBN 0-521-56392-5.

[13] M. Burrows and D. J. Wheeler. A Block–sorting Lossless Data Compression Algorithm. SRC Research Report 124, Digital Systems Research Center, Palo Alto, CA, May 1994.
`http://gatekeeper.dec.com/pub/DEC/SRC/`
`research-reports/abstracts/src-rr-124.html`.

[14] Brenton Chapin. Switching Between Two On-line List Update Algorithms for Higher Compression of Burrows-Wheeler Transformed Data. *Proc. Data Compression Conf.*, pages 183–192, 2000.

[15] Brenton Chapin and Stephen R. Tate. Higher Compression from the Burrows-Wheeler Transform by Modified Sorting. *Proc. Data Compression Conf.*, page 551, 1998.

[16] J. G. Cleary and I. H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, 32(4):396–402, 1984.

[17] T. M. Cover and J. A. Thomas. *Elements of Information Theory.* John Wiley & Sons, Inc., New York, 1991.

[18] Michelle Effros. Universal Lossless Source Coding with the Burrows-Wheeler Transformation. *Proc. Data Compression Conf.*, pages 178–187, 1999.

[19] Michelle Effros. PPM Performance with BWT Complexity: A new method for lossless data compression. *Proc. Data Compression Conf.*, pages 203–212, 2000.

[20] Peter Elias. Interval and Recency Rank Source Coding: Two On-Line Adaptive Variable-Length Schemes. *IEEE Transactions on Information Theory*, 33(1):3–10, 1987.

[21] P. Fenwick. Block Sorting Text Compression — Final Report. Technical Report 130, The University of Auckland, Department of Computer Science, March 1996. `ftp://ftp.cs.auckland.ac.nz/out/peter-f/TechRep130.ps`.

[22] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, New York, 1979.

[23] H. Guo and C.S. Burrus. Waveform and Image Compression Using the Burrows Wheeler Transform and the Wavelet Transform. *Proceedings of the 1997 IEEE International Conference on Image Processing*, 1997.

[24] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proc. Inst. Radio Eng.*, 40(9):1098–1101, September 1952.

[25] Chat Hurwitz. Traveling Salesperson Dispersion: Performance and Description of a Heuristic. Senior project, Cal Poly San Luis Obispo, 1992. Software available from the Stony Brook Algorithm Repository at `http://www.cs.sunysb.edu/~algorith/`.

[26] Independent Joint Photography Experts Group (JPEG) Group. Source code available at `http://www.ijg.org`.

[27] Sandy Irani. Two results on the list update problem. *Information Processing Letters*, 38(6):301–306, 1991.

[28] Holger Kruse and Amar Mukherjee. Preprocessing Text to Improve Compression Ratios. *Proc. Data Compression Conf.*, page 555, 1998.

[29] G. G. Langdon. An introduction to arithmetic coding. *IBM Journal of Research and Development*, 28:135–149, 1984.

[30] N. Jesper Larsson. The Context Trees of Block Sorting Compression. *Proc. Data Compression Conf.*, pages 189–198, 1998.

[31] A. Lempel and J. Ziv. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, 23:337–343, 1977.

[32] A. Lempel and J. Ziv. Compression of Individual Sequences via Variable-rate Coding. *IEEE Transactions on Information Theory*, 24:530–536, 1978.

[33] Ming Li and Paul Vitany. *An Introduction to Kolmogorov Complexity and Its Applications.* Springer-Verlag New York, Inc., 175 Fifth Av., New York, NY 10010, USA, 1997. ISBN 0-387-94868-6.

[34] Guy Louchard and Wojciech Szpankowski. Average Redundancy Rate of the Lempel-Ziv Code. *Proc. Data Compression Conf.*, pages 92–101, 1996.

[35] Giovanni Manzini. An Analysis of the Burrows-Wheeler Transform. *Proc. 10th ACM-SIAM Symposium on Discrete Algorithms*, pages 669–677, 1999. `http://www.acm.org/pubs/citations/proceedings/soda/314500/p669-manzini/` also `www.mfn.unipmn.it/~manzini/math/manzini0.htm`.

[36] Neri Merhav. On the Minimum Description Length Principle for Sources with Piecewise Constant Parameters. *IEEE Transactions on Information Theory*, 39(6):1962–1967, 1993.

[37] Motion Picture Experts Group. Documentation and source code for MPEG1–layer 3 audio codec available at `http://www.mp3-tech.org`.

[38] M. Nelson. Data Compression with the Burrows-Wheeler Transform. *Dr. Dobb's Journal*, page 46ff, September 1996.
`http://www.dogma.net/markn/articles/bwt/bwt.htm`.

[39] Dana Richards. Data Compression and Gray-code Sorting. *Information Processing Letters 22.*, pages 201–205, 1986.

[40] Ronald Rivest. On Self-Organizing Sequential Search Heuristics. *Communications of the ACM*, 19(2):63–67, February 1976.

[41] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis. An analysis of several heuristics for the traveling salesman problem. *SIAM J. Comput.*, 6:563–581, 1977.

[42] Kunihiko Sadakane. A Fast Algorithm for Making Suffix Arrays and for Burrows-Wheeler Transformation. *Proc. Data Compression Conf.*, pages 129–138, 1998.

[43] Michael Schindler. A Fast Block Sorting Algorithm for Lossless Data Compression. *Proc. Data Compression Conf.*, 1997.
`http://www.compressconsult.com/st/`.

[44] Julian Seward. `bzip2`. Burrows-Wheeler compression software.
`http://sourceware.cygnus.com/bzip2/`.

[45] Julian Seward. On the Performance of BWT Sorting Algorithms. *Proc. Data Compression Conf.*, pages 173–182, 2000.

[46] C. E. Shannon. A Mathematical Theory of Communication. *Bell System Tech. J.*, 27, 1948.

[47] Yuri M. Shtarkov and Tjalling J. Tjalkens. The Redundancy of the Ziv-Lempel Algorithm for Memoryless Sources. In *11th Symposium on Information Theory*

in the Benelux, 1990.
`http://ei1.ei.ele.tue.nl/~tjalling/zivlem/zivlem.html`.

[48] D. D. Sleator and R. E. Tarjan. Amortized Efficiency of List Update and Paging Rules. *Communications of the ACM*, 28(2):202–208, 1985.

[49] Boris Teia. A lower bound for randomized list update algorithms. *Information Processing Letters*, 47:5–9, 1993.

[50] Paul A. J. Volf and Frans M. J. Willems. Switching Between Two Universal Source Coding Algorithms. *Proc. Data Compression Conf.*, pages 491–500, 1998.

[51] Frans M. J. Willems. Coding for a Binary Independent Piecewise-Identically-Distributed Source. *IEEE Transactions on Information Theory*, 42(6):2210–2216, 1996.

[52] Portable Network Graphics.
`http://www.libpng.org/pub/png`.

[53] Project Gutenberg.
`http://promo.net/pg/`.