The Burrows-Wheeler Transform: Ten Years Later

DIMACS, August 19-20, 2004

Organizers:

Paolo Ferragina, University of Pisa.

Giovanni Manzini, University of Piemonte Orientale.

S. Muthu Muthukrishnan, Rutgers University.

List of Contributions

The Pre-history and Future of the Block-Sorting Compression Algorithm

Mike Burrows
Google, Inc.

David Wheeler
University of Cambridge, Computer Laboratory

I have organized the talk in two parts. The first is about the years before the algorithm was published, and the second looks forward to the next ten years.

Unfortunately, my co-author (and the algorithm's inventor) David Wheeler is unable to come to the workshop. However, he has given me some notes about the invention of the algorithm. I will begin by recounting some of his early thoughts; some have stood the test of time, while others have not. I'll also describe our efforts to make a practical compressor based on the algorithm, and various things that didn't make it into the original paper.

The compression achieved by the algorithm was good from the first, and further gains have been achieved since its publication. However, its compression and decompression speed have received less attention. This is curious, because it is its speed rather than its compression ratio that potential users find most irksome. One might think that the steady improvement in CPU performance would lead to fewer concerns about its performance; certainly, larger memories have made its memory footprint less troublesome that it was. Nevertheless, the unusual structure of the algorithm has rendered recent advances in CPU microarchitecture largely ineffective at improving its performance, while other compression algorithms have benefitted. In contrast, I suspect that the next ten years of improvements in hardware are more likely to favour block-sorting over competing techniques, given suitable algorithmic changes. I hope to report on one such promising change (joint work with my colleagues Sean Quinlan and Sean Doward at Google).

An Error-Resilient Blocksorting Compression Algorithm

Lee Butterman
Dept. of Computer Science, Brown University
Lee_Butterman@brown.edu

Nasir Memon
Dept. of Computer Science, Polytechnic University
memon@poly.edu

One key limitation of adaptive lossless compression systems is the susceptibility to errors in the compressed bit stream. The inherent design of these techniques often requires discarding all data subsequent to the error. This is particularly problematic in the Burrows-Wheeler Blocksorting Transform (BWT), which is used in practice to compress files around 1MB at a time. In the original BWT, each corrupted block must be completely discarded, but decreasing blocksize increases the bit-per-character (bpc) rates dramatically. Error-correcting codes (such as Reed-Solomon codes) can be used, but their design allows for a maximum pre-fixed error rate, and if the channel errors exceed this, the whole block is again lost. In this paper we present an error-resilient version of the BWT that has error-free output in low channel noise, and gracefully degrades output quality as errors increase by scattering output errors, thereby avoiding the significant error propagation typical with adaptive lossless compression systems (and BWT in particular). The techniques we develop also yields interesting new insights into this popular compression algorithm. We show how to perform the inverse blocksort to decode both forwards and backwards by inverting the original permutation vector T. We periodically save the values of T as overhead. When the inverse blocksorter encounters an error, it can start at a specific position that corresponds to one of the saved T anchors, and it can bi-directionally decode until reaching an error. When there is more than one error between two anchors, we can decode forwards up to the first error and backwards from the next anchor until the last error between the anchors. This also generates subloops, decodable text between the anchors that is not connected. The problem of reconnecting them can be formulated as a TSP problem and can be solved by an appropriate heuristic.

Compression Boosting Using the Burrows-Wheeler Transform

Paolo Ferragina
University of Pisa, Italy
ferragina@di.unipi.it

Raffaele Giancarlo
University of Palermo, Italy
raffaele@math.unipa.it

Giovanni Manzini
University of Piemonte Orientale
manzini@unipmn.it

Marinella Sciortino
University of Palermo, Italy
mari@math.unipa.it

We discuss a general boosting technique for data compression. Qualitatively, our technique takes a good compression algorithm and turns it into an algorithm with a better compression performance guarantee. Our technique displays the following remarkable properties: (a) it can turn *any memoryless* compressor into a compression algorithm that uses the "best possible" contexts; (b) it is very simple and *optimal* in terms of time; (c) it admits a decompression algorithm again optimal in time. To the best of our knowledge, this is the first boosting technique displaying these properties.

Technically, our boosting technique builds upon three main ingredients: the Burrows-Wheeler Transform, the Suffix Tree data structure, and a greedy algorithm to process them. Specifically we show that there exists a proper partition of the Burrows-Wheeler Transform of a string s that shows a deep combinatorial relation with the k-th order entropy of s. That partition can be identified via a greedy processing of the suffix tree of s with the aim of minimizing a proper objective function over its nodes. The final compressed string is then obtained by compressing individually each substring of the partition by means of the base compressor we wish to boost.

Our boosting technique is inherently combinatorial because it does not need to assume any prior probabilistic model about the source emitting s, and it does not deploy any training, parameter estimation and learning. Various corollaries are derived from this main achievement. Among the others, we show analytically that using our booster we get better compression algorithms than some of the best existing ones, i.e., LZ77, LZ78, PPMC and the ones derived from the Burrows-Wheeler Transform. Further, we settle analytically some long standing open problems about the algorithmic structure and the performance of BWT-based compressors. Namely, we provide the first family of BWT algorithms that do not use Move-To-Front or Symbol Ranking as a part of the compression process.

Using the Burrows-Wheeler Transform for PPM compression without escapes

Peter Fenwick
Department of Computer Science,
The University of Auckland, Auckland, New Zealand
p.fenwick@auckland.ac.nz

High performance lossless data compression is dominated by two algorithms. The older algorithm, PPM or Prediction by Partial Matching, gives slightly better compression, but requires rather more complex data structures and has rather slower execution. The newer algorithm, based on the Burrows Wheeler transform, uses simpler data structures and is faster but with slightly poorer compression.

Although PPM is now a relatively mature technique, Burrows Wheeler compression is still not well understood, although knowledge is certainly improving. (It must be recognized that PPM at the same "age", 10 years, was also still largely a collection of ad hoc rules.) Some previous work has tried to combine Burrows Wheeler and PPM techniques, usually by adapting PPM to process the permuted output of the Burrows Wheeler transformation and bypassing the more usual Move-to-Front recoding of Burrows Wheeler. This work has generally given little advantage, largely because the permuted output has a context structure which is not easily exploited by PPM techniques.

The performance of PPM is limited largely by the handling of escapes, to control the coding of symbols that have not been seen in a higher order. An escape is a symbol that must be encoded as any other, according to its probability. But whereas we can give a good estimate of the probabilities of known symbols (which we have seen), it is much more difficult to estimate the probability of encountering an unexpected symbol. Improvements in PPM have largely followed from improvements in estimating escape probabilities.

This paper discusses work-in-progress on a new technique for combining Burrows Wheeler and PPM. While most prior work on Burrows Wheeler compression emphasises data manipulation after the forward transform and before its inverse, the current approach looks much more closely at the inverse transform.

In conventional Burrows Wheeler compression the reverse transform generates a single text string, corresponding to the original compressed text. But the reverse transform can do much more than this and can generate contexts for *all* of the symbols in the permuted data (and therefore contexts for every input symbol). The usual transform generates the context for either the last input symbol (with forward contexts) or the first symbol (with reverse contexts and producing reversed output). Conventional Burrows Wheeler produces its single output context to an unbounded order, so recovering the entire input text.

Here we use the reverse transform to generate contexts for every symbol, but only to some small constant order, say 6 or 8. Because we generate all contexts for the input, a PPM compressor can operate at constant order, without escapes. With contexts needed to only say 6 or 8 symbols, the forward transform can be simplified to use comparisons to

only a similar length. A slight modification to the forward comparison allows like symbols to be grouped within their contexts, with run-length encoding improving the information density.

The compressor starts by using a normal forward Burrows Wheeler transform, but with limited comparison lengths. Its permuted output is encoded (in any "standard Burrows Wheeler" manner) as the context-definition part of the compressed output. The permuted output is also used to generate the complete suite of constant order PPM contexts, using the modified reverse transform. Finally, the original data is processed by the constant order PPM compressor, using the contexts generated from the Burrows Wheeler processing. The PPM output forms the second part of the compressed output.

The decompressor first creates the contexts, using the same code as was used in compression. It then uses the second part of the compressed data to drive the PPM decoder and recover the original text. Note here that PPM codes are generated and emitted only as needed; many PPM contexts are deterministic and their one symbol can be emitted immediately with no guidance from the compressor.

Results so far do not give the expected improvements, because it turns out not that much cheaper to send the reduced context information than the full Burrows Wheeler output, and also because many symbols are encoded twice, once for the contexts and once for PPM. But two lines of future study are suggested –

1. It is often possible to develop contexts rather longer than the Burrows Wheeler comparison. This may reduce the costs of sending the contexts.

2. It is suggested that there may be a connection between error correction coding and compression which would allow some PPM codes to be replaced by erasure symbols. Techniques allied to trellis or Viterbi coding may then allow the contexts to be recovered.

Generalized Burrows-Wheeler Transform

Sabrina Mantaci
University of Palermo, Dipartimento di Matematica ed Applicazioni,
sabrina@math.unipa.it

Antonio Restivo
University of Palermo, Dipartimento di Matematica ed Applicazioni,
restivo@math.unipa.it

Marinella Sciortino
University of Palermo, Dipartimento di Matematica ed Applicazioni,
mari@math.unipa.it

Michael Burrows and David Wheeler introduced in 1994 (cf. [1]) a reversible transformation on strings (BWT from now on) that arouses considerable interest and curiosity in the field of Data Compression.

Most of the studies on the Burrows Wheeler Transform have been experimental and developed within the Data Compression community. Very recently some combinatorial aspects of this transform have been investigated (cf. [4, 5]). For instance, by using the Burrows-Wheeler transform, one can derive a further characterization of Standard words, that are very important objects in the field of Combinatorics on Words: more specifically, Standard words correspond to the extremal case, for binary alphabets, of BWT, in the sense that the transform produces a total clustering of all the instances of any character.

Moreover it has been proved (cf. [2]) that there exists a very close relation between BWT and a transformation, described by Gessel and Reutenauer in [3], based on a bijection between finite words and the set of permutations with a given cyclic structure and a given descent set. Actually in this formalism, BWT corresponds to the special case in which the considered permutations are cyclic and the cardinality of its descent set is less than the cardinality of the alphabet.

Now, by starting from the Gessel and Reutenauer transform, we can define a generalized Burrows-Wheeler transform (denoted by $GBWT$) applied to a set of n words. This transformation involves an order relation between words that is different from the lexicographic one. The sorting process derived from such an order relation is realized by using the Fine and Wilf Theorem for n periods (cf. [6]).

Moreover the $GBWT$ allows to introduce a new notion of similarity between words that takes into account how equal factors of two words appear in similar contexts.

References

[1] M. Burrows and D.J. Wheeler. A block sorting data compression algorithm. Technical report, DIGITAL System Research Center, 1994.

[2] M. Crochemore, J. Désarménien, and D. Perrin. A note on the Burrows-Wheeler transformation. *Theoret. Comput. Sci.* to appear.

[3] I. M. Gessel and C. Reutenauer. Counting permutations with given cycle structure and descent set. *J. Combin. Theory Ser. A*, 64(2):189–215, 1993.

[4] S. Mantaci, A. Restivo, and M. Sciortino. Burrows-Wheeler transform and Sturmian words. *Informat. Proc. Lett.*, 86:241–246, 2003.

[5] S. Mantaci, A. Restivo, and M. Sciortino. Combinatorial aspects of the Burrows-Wheeler transform. *TUCS (Turku Center for Computer Science) General Pubblication*, 25:292–297, 2003. proc. WORDS 2003.

[6] R. Tijdeman and L.Zamboni. Fine and Wilf words for any periods. *Indag. Math.*, 14(1):135–147, 2003.

A Survey of Suffix Sorting

Martin Farach-Colton
Department of Computer Science, Rutgers University

Suffix trees are the fundamental data structure of stringology. The suffix array is a low-memory cousin of the suffix tree. In the last 10 years, we have come to realize that building such data structures is equivalent to sorting the suffixes of a string. Such suffix sorting is the key algorithmic component of the Burrows-Wheeler transform. We will show the equivalences between suffix trees, suffix arrays and suffix sorting, as well as the algorithm that suffix sorts in the same time taken to simply sort the characters of the string. The field has matured to the point where we may well have the suffix-sorting algorithm "from the book".

Fast BWT in Small Space by Blockwise Suffix Sorting

Juha Kärkkäinen
Department of Computer Science, University of Helsinki, Finland
juha.karkkainen@cs.helsinki.fi

The usual way to compute the Burrows-Wheeler transform (BWT) [3] of a text is by constructing the suffix array of the text. Even with space-efficient suffix array construction algorithms [12, 2], the space requirement of the suffix array itself is often the main factor limiting the size of the text that can be handled in one piece, which is crucial for constructing compressed text indexes [4, 5]. Typically, the suffix array needs $4n$ bytes while the text and the BWT need only n bytes each and sometimes even less, for example $2n$ bits each for a DNA sequence.

We reduce the space dramatically by constructing the suffix array in *blocks* of lexicographically consecutive suffixes. Given such a block, the corresponding block of the BWT is trivial to compute.

Theorem 1 *The BWT of a text of length n can be computed in $\mathcal{O}(n \log n + n\sqrt{v} + D_v)$ time (with high probability) and $\mathcal{O}(n/\sqrt{v} + v)$ space (in addition to the text and the BWT), for any $v \in [1, n]$. Here $D_v = \sum_{i \in [0,n)} \min(d_i, v) = \mathcal{O}(nv)$, where d_i is the length of the shortest unique substring starting at i.*

Proof (sketch). Assume first that the text has no repetitions longer than v, i.e., $d_i \leq v$ for all i. Choose a set of $\mathcal{O}(v)$ random suffixes that divide the suffix array into blocks. The sizes of the blocks are counted in $\mathcal{O}(n \log v + D_v)$ time using the string binary search technique from [11]. Blocks are then combined to obtain $\mathcal{O}(\sqrt{v})$ blocks of size $\mathcal{O}(n/\sqrt{v})$. The suffixes in a block are collected in $\mathcal{O}(n)$ time and $\mathcal{O}(v)$ extra space using a modified Knuth–Morris–Pratt algorithm with (the prefixes of) the bounding suffixes as patterns. A block B is sorted in-place in $\mathcal{O}(|B| \log |B| + D_v(B))$ time using the multikey quicksort [1], where $D_v(B)$ is as D_v but summed over the suffixes in B. Repetitions longer than v are handled in all stages with the difference cover sampling (DCS) data structure from [2] that supports constant time order comparison of any two suffixes that have a common prefix of length v. The DCS data structure can be constructed in $\mathcal{O}((n/\sqrt{v}) \log(n/\sqrt{v}) + D_v(C))$ time and $\mathcal{O}(n/\sqrt{v} + v)$ space, where C is a set of $\mathcal{O}(n/\sqrt{v})$ suffixes. □

With the choice of $v = \log^2 n$, we get an algorithm using $\mathcal{O}(n)$ bits of space and running in $\mathcal{O}(n \log n)$ time on average and in $\mathcal{O}(n \log^2 n)$ time in the worst case. The algorithm is also fast and space-efficient in practice. The following table shows the space requirement of a practical implementation for some v (not including the text, the BWT and about $16v + \mathcal{O}(\log n)$ bytes).

v	16	32	64	128	256	512	1024	2048
bits	$20n$	$14n$	$9n$	$6.5n$	$5n$	$3.5n$	$2.5n$	$1.8n$

For small v, the runtime is dominated by the sorting of blocks making the performance similar to the algorithm in [2], which is competitive with the best algorithms. For larger v, the time needed for the $\mathcal{O}(\sqrt{v})$ scans to collect suffixes for a block takes over. The D_v term is dominant only in pathological cases.

There are two other categories of algorithms for computing the BWT when there is not enough space for the suffix array: compressed suffix array construction [10, 6, 7] and external memory suffix array construction [8, 9]. Our guess is that the blockwise suffix sorting is the fastest alternative in practice when v is not too large, and we are in the process of verifying this experimentally.

References

[1] Jon L. Bentley and Robert Sedgewick. Fast algorithms for sorting and searching strings. In *Proc. 8th Annual Symposium on Discrete Algorithms*, pages 360–369. ACM, 1997.

[2] Stefan Burkhardt and Juha Kärkkäinen. Fast lightweight suffix array construction and checking. In *Proc. 14th Annual Symposium on Combinatorial Pattern Matching*, volume 2676 of *LNCS*, pages 55–69. Springer, 2003.

[3] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, SRC (digital, Palo Alto), May 1994.

[4] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proc. 41st Annual Symposium on Foundations of Computer Science*, pages 390–398. IEEE, 2000.

[5] Paolo Ferragina and Giovanni Manzini. An experimental study of an opportunistic index. In *Proc. 12th Annual Symposium on Discrete Algorithms*, pages 269–278. ACM–SIAM, 2001.

[6] Wing-Kai Hon, Tak-Wah Lam, Kunihiko Sadakane, and Wing-Kin Sung. Constructing compressed suffix arrays with large alphabets. In *Proc. 14th International Symposium on Algorithms and Computation*, volume 2906 of *LNCS*, pages 240–249. Springer, 2003.

[7] Wing-Kai Hon, Kunihiko Sadakane, and Wing-Kin Sung. Breaking a time-and-space barrier in constructing full-text indices. In *Proc. 44th Annual Symposium on Foundations of Computer Science*, pages 251–260. IEEE, 2003.

[8] Juha Kärkkäinen and S. Srinivasa Rao. Full-text indexes in external memory. In U. Meyer, P. Sanders, and J. Sibeyn, editors, *Algorithms for Memory Hierarchies (Advanced Lectures)*, volume 2625 of *LNCS*, chapter 7, pages 149–170. Springer, 2003.

[9] Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In *Proc. 30th International Conference on Automata, Languages and Programming*, volume 2719 of *LNCS*, pages 943–955. Springer, 2003.

[10] Tak-Wah Lam, Kunihiko Sadakane, Wing-Kin Sung, and Siu-Ming Yiu. A space and time efficient algorithm for constructing compressed suffix arrays. In *Proc. 8th Annual International Conference on Computing and Combinatorics*, volume 2387 of *LNCS*, pages 401–410. Springer, 2002.

[11] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, October 1993.

[12] G. Manzini and P. Ferragina. Engineering a lightweight suffix array construction algorithm. In *Proc. 10th Annual European Symposium on Algorithms*, volume 2461 of *LNCS*, pages 698–710. Springer, 2002.

Efficient Computation of the Burrows-Wheeler Transform

Kunihiko Sadakane
Department of Computer Science and Communication Engineering,
Kyushu University, Japan
sada@csce.kyushu-u.ac.jp

The Burrows-Wheeler Transform [1] (BWT) is the core technique that unifies text search and compression. Many self-indexing text indices use this technique [2, 7, 3]. In spite of its importance, there has been no time and space efficient algorithm to compute the BWT. Though it can be computed in linear time by using linear time algorithms for constructing suffix trees or suffix arrays, they need much more space than the output. On the other hand, the BWT can be computed using $O(\log n)$-bit extra space although it will require $O(n^3)$ time for a text of length n.

In this talk, we review the development of time and space efficient algorithms for computing the BWT. The algorithms runs in optimal space, i.e., $O(n \log |\mathcal{A}|)$-bit working space for a text of length n on alphabet $\mathcal{A}$. The time complexities are $O(n|\mathcal{A}| \log n)$ [6], $O(n \log n)$ [4], and $O(n \log \log |\mathcal{A}|)$ [5].

References

[1] M. Burrows and D. J. Wheeler. A Block-sorting Lossless Data Compression Algorithms. Technical Report 124, Digital SRC Research Report, 1994.

[2] P. Ferragina and G. Manzini. Opportunistic Data Structures with Applications. In *41st IEEE Symp. on Foundations of Computer Science*, pages 390–398, 2000.

[3] R. Grossi, A. Gupta, and J. S. Vitter. When indexing equals compression: experiments with compressing suffix arrays and applications. In *Proc. ACM-SIAM SODA 2004*, pages 636–645, 2004.

[4] W. K. Hon, T. W. Lam, K. Sadakane, and W. K. Sung. Constructing Compressed Suffix Arrays with Large Alphabets. In *Proc. of ISAAC*, pages 240–249. LNCS 2906, 2003.

[5] W. K. Hon, K. Sadakane, and W. K. Sung. Breaking a Time-and-Space Barrier in Constructing Full-Text Indices. In *Proc. IEEE FOCS*, pages 251–260, 2003.

[6] T. W. Lam, K. Sadakane, W. K Sung, and S. M Yiu. A Space and Time Efficient Algorithm for Constructing Compressed Suffix Arrays. In *Proc. COCOON*, pages 401–410. LNCS 2387, 2002.

[7] K. Sadakane. New Text Indexing Functionalities of the Compressed Suffix Arrays. *Journal of Algorithms*, 48(2):294–313, 2003.

The FM-Index: A Compressed Full-Text Index Based on the BWT

Paolo Ferragina
Università di Pisa
ferragina@di.unipi.it

Giovanni Manzini
Università del Piemonte Orientale
manzini@unipmn.it

In this talk we address the issue of indexing compressed data both from the theoretical and the practical point of view.

We start by introducing the *FM-index* data structure [2] that supports substring searches and occupies a space which is a function of the entropy of the indexed data. The key feature of the FM-index is that it encapsulates the indexed data (*self-index*) and achieves the space reduction at no significant slowdown in the query performance. Precisely, given a text $T[1,n]$ to be indexed, the FM-index occupies at most $5nH_k(T)+o(n)$ bits of storage, where $H_k(T)$ is the k-th order entropy of T, and allows the search for the *occ* occurrences of a pattern $P[1,p]$ within T in $O(p+occ\log^{1+\epsilon} n)$ time, where $\epsilon > 0$ is an arbitrary constant fixed in advance.

The design of the FM-index is based upon the relationship between the Burrows-Wheeler compression algorithm [1] and the suffix array data structure [9]. It is therefore a sort of *compressed suffix array* that takes advantage of the compressibility of the indexed data in order to achieve space occupancy close to the Information Theoretic minimum. Indeed, the design of the FM-index does not depend on the parameter k and its space bound holds *simultaneously over all* $k \geq 0$.

These remarkable theoretical properties have been validated by experimental results [3, 4] and applications [7, 10]. In particular it has been shown that the FM-index achieves a space occupancy close to the best known compressors and, unlike them, it allows to search for arbitrary substrings in a hundred of megabytes within few millisecs, since it does not decompress the whole file.

We will conclude the talk by sketching two intriguing variants of the FM-index. One achieves $O(p+occ)$ query time (i.e. *output sensitivity*) and uses $O(nH_k(T)\log^{\epsilon} n)+o(n)$ bits of storage. This data structure exploits the interplay between two compressors: the Burrows-Wheeler algorithm and the LZ78 algorithm [11]. Our other proposal [8] combines two recent and elegant techniques—the compression boosting [5] and the wavelet tree [6]—to design a variant of the FM-index that scales well with the size of the input alphabet.

References

[1] M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

[2] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc. of the 41st IEEE Symposium on Foundations of Computer Science*, pages 390–398, 2000.

[3] P. Ferragina and G. Manzini. An experimental study of a compressed index. *Information Sciences: special issue on "Dictionary Based Compression"*, 135:13–28, 2001.

[4] P. Ferragina and G. Manzini. An experimental study of an opportunistic index. In *Proc. 12th ACM-SIAM Symposium on Discrete Algorithms*, pages 269–278, 2001.

[5] P. Ferragina and G. Manzini. Compression boosting in optimal linear time using the Burrows-Wheeler transform. In *Proc. 15th ACM-SIAM Symposium on Discrete Algorithms (SODA '04)*, 2004.

[6] R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th Annual ACM-SIAM Symp. on Discrete Algorithms (SODA '03)*, pages 841–850, 2003.

[7] J. Healy, E. E. Thomas, J. T. Schwartz, and M. Wigler. Annotating large genomes with exact word matches. *Genome Research*, 13:2306–2315, 2003.

[8] P. Ferragina, G. Manzini, V. Mäkinen and G. Navarro. *An Alphabet-Friendly FM-index*. Subbmitted. 2004.

[9] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[10] K. Sadakane and T. Shibuya. Indexing huge genome sequences for solving various problems. *Genome Informatics*, 12:175–183, 2001.

[11] J. Ziv and A. Lempel. Compression of individual sequences via variable length coding. *IEEE Transaction on Information Theory*, 24:530–536, 1978.

Run-Length FM-index

Veli Mäkinen
Department of Computer Science, Univ. of Helsinki, Finland.

Gonzalo Navarro
Department of Computer Science, Univ. of Chile, Chile.

Abstract

The FM-index is a succinct text index needing only $O(H_k n)$ bits of space, where n is the text size and H_k is the kth order entropy of the text. FM-index assumes constant alphabet; it uses exponential space in the alphabet size, σ. In this paper we show how the same ideas can be used to obtain an index needing $O(H_k n)$ bits of space, with the constant factor depending only logarithmically on σ. Our space complexity becomes better as soon as $\sigma \log \sigma > \log n$, which means in practice for all but very small alphabets, even with huge texts. We retain the same search complexity of the FM-index.

FM-index

The FM-index [3] is based on the *Burrows-Wheeler transform (BWT)* [1], which produces a permutation of the original text, denoted by $T^{bwt} = bwt(T)$. String T^{bwt} is a result of the following *forward* transformation: (1) Append to the end of T a special end marker \$, which is lexicographically smaller than any other character; (2) form a *conceptual* matrix $\mathcal{M}$ whose rows are the cyclic shifts of the string $T\$$, sorted in lexicographic order; (3) construct the transformed text L by taking the last column of $\mathcal{M}$. The first column is denoted by F.

The *suffix array* $\mathcal{A}$ of text $T\$$ is essentially the matrix $\mathcal{M}$: $\mathcal{A}[i] = j$ iff the ith row of $\mathcal{M}$ contains string $t_j t_{j+1} \cdots t_n \$ t_1 \cdots t_{j-1}$. Given the suffix array, the search for the occurrences of the pattern $P = p_1 p_2 \cdots p_m$ is trivial. The occurrences form an interval $[sp, ep]$ in $\mathcal{A}$ such that suffixes $t_{\mathcal{A}[i]} t_{\mathcal{A}[i]+1} \cdots t_n$, $sp \leq i \leq ep$, contain the pattern as a prefix. This interval can be searched for using two binary searches in time $O(m \log n)$ [5].

The suffix array of text T is represented implicitly by T^{bwt}. The novel idea of the FM-index is to store T^{bwt} in compressed form, and to simulate a *backward search* in the suffix array as follows:

Algorithm FM_Search($P[1,m]$,$T^{bwt}[1,n]$)
(1) $c = P[m]$; $i = m$;
(2) $sp = C_T[c] + 1$; $ep = C_T[c+1]$;
(3) **while** $(sp \leq ep)$ **and** $(i \geq 2)$ **do**
(4) $\quad c = P[i-1]$;
(5) $\quad sp = C_T[c] + Occ(T^{bwt}, c, sp-1) + 1$;
(6) $\quad ep = C_T[c] + Occ(T^{bwt}, c, ep)$;
(7) $\quad i = i - 1$;
(8) **if** $(ep < sp)$ **then return** "not found" **else return** "found $(ep - sp + 1)$ occs".

The above algorithm finds the interval $[sp, ep]$ of $\mathcal{A}$ containing the occurrences of the pattern P. It uses the array C_T and function $Occ(X, c, i)$, where $C_T[c]$ equals the number of occurrences of characters $\{\$, 1, \ldots, c-1\}$ in the text T and $Occ(X, c, i)$ equals the number of occurrences of character c in the prefix $X[1, i]$.

Ferragina and Manzini [3] go on to describe an implementation of $Occ(T^{bwt}, c, i)$ that uses a compressed form of T^{bwt}; they show how to compute $Occ(T^{bwt}, c, i)$ for any c and i in constant time. However, to achieve this they need exponential space (in the size of the alphabet).

Run-Length FM-Index

Our idea is to exploit run-length compression to represent T^{bwt}. An array S contains one character per run in T^{bwt}, while an array B contains n bits and marks the beginnings of the runs.

Definition 2 *Let string $T^{bwt} = c_1^{\ell_1} c_2^{\ell_2} \ldots c_{n'}^{\ell_{n'}}$ consist of n' runs, so that the i-th run consists of ℓ_i repetitions of character c_i. Our representation of T^{bwt} consists of string $S = c_1 c_2 \ldots c_{n'}$ of length n', and bit array $B = 10^{\ell_1 - 1} 10^{\ell_2 - 1} \ldots 10^{\ell_{n'} - 1}$.*

It is clear that S and B contain enough information to reconstruct T^{bwt}: $T^{bwt}[i] = S[rank(B, i)]$, where $rank(B, i)$ is the number of 1's in $B[1 \ldots i]$ (so $rank(B, 0) = 0$). Function $rank$ can be computed in constant time using $o(n)$ extra bits [4, 6, 2]. Hence, S and B give us a representation of T^{bwt} that permits us accessing any character in constant time and requires at most $n' \log \sigma + n + o(n)$ bits. The problem, however, is not only how to access T^{bwt}, but also how to compute $C_T[c] + Occ(T^{bwt}, c, i)$ for any c and i.

In the following we show that the above can be computed by means of a bit array B', obtained by reordering the runs of B in lexicographic order of the characters of each run. Runs of the same character are left in their original order. The use of B' will add $n + o(n)$ bits to our scheme. We also use C_S, which plays the same role of C_T, but it refers to string S.

Definition 3 *Let $S = c_1 c_2 \ldots c_{n'}$ of length n', and $B = 10^{\ell_1 - 1} 10^{\ell_2 - 1} \ldots 10^{\ell_{n'} - 1}$. Let $p_1 p_2 \ldots p_{n'}$ be a permutation of $1 \ldots n'$ such that, for all $1 \leq i < n'$, either $c_{p_i} < c_{p_{i+1}}$ or $c_{p_i} = c_{p_{i+1}}$ and $p_i < p_{i+1}$. Then, bit array B' is defined as $B' = 10^{\ell_{p_1} - 1} 10^{\ell_{p_2} - 1} \ldots 10^{\ell_{p_{n'}} - 1}$.*

We now give the theorems that cover different cases in the computation of $C_T[c] + Occ(T^{bwt}, c, i)$ (see [7] for proofs). They make use of *select*, which is the inverse of *rank*: $select(B', j)$ is the position of the jth 1 in B' (and $select(B', 0) = 0$). Function *select* can be computed in constant time using $o(n)$ extra bits [4, 6, 2].

Theorem 4 *For any $c \in \Sigma$ and $1 \leq i \leq n$, such that $T^{bwt}[i] \neq c$, it holds*

$$C_T[c] + Occ(T^{bwt}, c, i) \;=\; select(B', C_S[c] + 1 + Occ(S, c, rank(B, i))) - 1$$

Theorem 5 *For any $c \in \Sigma$ and $1 \leq i \leq n$, such that $T^{bwt}[i] = c$, it holds*

$$\begin{aligned} C_T[c] + Occ(T^{bwt}, c, i) \;=\; & select(B', C_S[c] + Occ(S, c, rank(B, i))) \\ & + i - select(B, rank(B, i)). \end{aligned}$$

Since functions *rank* and *select* can be computed in constant time, the only obstacle to use the theorems is the computation of Occ over string S.

Instead of representing S explicitly, we will store one bitmap S_c per text character c, so that $S_c[i] = 1$ iff $S[i] = c$. Hence $Occ(S, c, i) = rank(S_c, i)$. It is still possible to determine in constant time whether $T^{bwt}[i] = c$ or not: an equivalent condition is $S_c[rank(B, i)] = 1$.

According to [8], a bit array of length n' where there are f 1's can be represented using $\log \binom{n'}{f} + o(f) + O(\log \log n')$ bits, while still supporting constant time access and constant time *rank* function for the positions with value 1. It can be shown (see [7]) that the overall size of these structures is at most $n'(\log \sigma + 1.44 + o(1)) + O(\sigma \log n')$.

We have shown in [7] that the number of runs in T^{bwt} is limited by $2H_k n + \sigma^k$. By adding up all our space complexities we obtain $2n(H_k(\log \sigma + 1.44 + o(1)) + 1 + o(1)) + O(\sigma \log n) = 2n(1 + H_k \log \sigma)(1 + o(1))$ bits of space if $\sigma = o(n/\log n)$, for any fixed k.

References

[1] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. *DEC SRC Research Report 124*, 1994.

[2] D. Clark. *Compact Pat Trees*. PhD thesis, University of Waterloo, 1996.

[3] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc. FOCS'00*, pp. 390–398, 2000.

[4] G. Jacobson. *Succinct Static Data Structures*. PhD thesis, CMU-CS-89-112, Carnegie Mellon University, 1989.

[5] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22, pp. 935–948, 1993.

[6] I. Munro. Tables. In *Proc. FSTTCS'96*, pp. 37–42, 1996.

[7] V. Mäkinen and G. Navarro. New search algorithms and time/space tradeoffs for succinct suffix arrays. Technical report C-2004-20, Dept. Computer Science, Univ. Helsinki, April 2004.

[8] R. Raman, V. Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. SODA'02*, pp. 233–242, 2002.

Entropy-Compressed Indexes for Multidimensional Pattern Matching

Roberto Grossi
University of Pisa, Italy

Ankur Gupta
Duke University, USA

Jeffrey Scott Vitter
Purdue University, USA

Abstract

In this talk, we will discuss the challenges involved in developing a multidimensional generalizations of compressed text indexing structures. These structures depend on some notion of Burrows-Wheeler transform (BWT) for multiple dimensions, though naive generalizations do not enable multidimensional pattern matching. We study the 2D case to possibly highlight combinatorial properties that do not emerge in the 1D case. We also present related work in 2D pattern matching and indexing.

Introduction

Suffix arrays and suffix trees are ubiquitous data structures at the heart of several text and string algorithms. They are used in a wide variety of applications, including pattern matching, text and information retrieval, Web searching, and sequence analysis in computational biology [14]. Compressed suffix arrays [13, 18, 19] and opportunistic FM-indexes [7, 8] represent new trends in the design of advanced indexes for full-text searching of documents, in that they support the functionalities of suffix arrays and suffix trees, which are more powerful than classical inverted files, yet they also overcome the aforementioned space limitations by exploiting, in a novel way, the notion of text compressibility and the techniques developed for succinct data structures and bounded-universe dictionaries.

Grossi and Vitter [13] developed the compressed suffix array using $2n \log |\Sigma|$ bits in the worst case with $o(m)$ searching time. Sadakane [18, 19] related the space bound to the order-0 empirical entropy H_0. Ferragina and Manzini devised the FM-index [7, 8], which is based on the Burrows-Wheeler transform (BWT) and is the first to encode the index size with respect to the hth-order empirical entropy H_h of the text. Navarro [17] recently developed an index requiring $4nH_h + o(n)$ bits, and boasts fast search. Grossi, Gupta, and Vitter [11] exploited the higher-order entropy H_h of the text to represent a compressed suffix array in just $nH_h + O(n \log\log n / \log_{|\Sigma|} n)$ bits. The index is optimal in space, apart from lower-order terms, achieving asymptotically the empirical entropy of the text (with a multiplicative constant 1). These data structures also have practical significance, as detailed in [12].

An interesting extension, with practical applications related to image matching, is to develop a data structure that achieves similar space bounds as the 1-D case and the same time bounds as known multidimensional data structures. Multidimensional data present a new challenge when trying to capture entropy, as now the critical notion of *spatial information* also enters into play. (In a strict sense, this information was always present, but we can anticipate more dependence upon spatially linked data.) Stronger notions of compression are applicable, yet the searches are more complicated. Achieving both, is again, a challenge.

Multidimensional Matching

We define a text matrix $T^{(d)}$ as a hypercube in d dimensions with length n, where each symbol is drawn from the alphabet $\Sigma = \{0, 1, \ldots, \sigma\}$. For example, $T^{(2)}$ represents an $n \times n$ text matrix, and $T^{(1)} = T$ simply represents a text document with n symbols.

Handling high-order entropy (and other entropy notions) for multidimensional data in a practical way is difficult. We generalize the notion of hth order entropy as follows. For a given text $T^{(d)}$, we define $H_h^{(d)}$ as

$$H_h^{(d)} = \sum_{x \in A^{(d)}} \sum_{y \in \Sigma} -\operatorname{Prob}[y, x] \cdot \log \operatorname{Prob}[y|x],$$

where $A^{(d)}$ is a d-dimensional text matrix with length h.

A common method used to treat data more contextually (and thus, consider spatial information explicitly) is to *linearize* the data. Linearization is the task of performing somewhat of a "map" to the 1-D case (so that the data is again laid out as we are accustomed to). One technique is described in [15, 16]. Linearization is primarily performed to meet the constraints put forth by Giancarlo [9, 10] in order to support pattern matching in 2-D. (These constraints are readily generalized to multidimensions.)

One major goal of ours in multidimensional matching is to improve the space requirement, without affecting the search times already achieved in literature. Not considering space-efficient solutions (which are absent from current literature), the 2-D pattern matching problem is widely studied by Amir, Benson, Farach, and other researchers [2, 4, 6, 5, 3]. In particular, Amir and Benson [1] give compressed matching algorithms for 2-dimensional cases; however, their pattern matching is not indexing and it needs the scan over entire compressed text.

Suffix arrays and trees have been generalized to multiple dimensions, and a great deal of literature is available that describes various incarnations of these data structures [15, 16], but the vast majority of them discuss just the construction time of these powerful structures. Little work has been done on space-efficient versions of these structures, nor has any real emphasis been given to achieving optimal performance. The hurdles are far more basic than that.

The primary difficulty stems from the fact that there is no clear multidimensional analogue for the Burrows-Wheeler transform (BWT) that still allows for multidimensional pattern matching. The BWT is critical to achieving high-order entropy in one dimension [13, 7, 8, 11, 12]; there, each suffix of the text is sorted and can be indexed using a variety of tricks [7, 8, 11]. Even with just two dimensions, the problem becomes difficult to solve.

In order to support multidimensional pattern matching, the data should be considered from a localized view of the data, namely in terms of hypercubes (which in 1-D is simply

a contiguous sequence of symbols) starting at each position of the text. However, a BWT cannot be formed explicitly upon such a view, as any such localized view violates the critical invariant that suffixes must overlap perfectly. Nevertheless, some basic notions have been explored [9, 10, 15, 16] as a first step in tackling these limitations.

Goals of Study

We hope to make major inroads beyond [15, 16] by developing the crucial notion of a multidimensional BWT. We study the 2D case to highlight combinatorial properties that do not appear in the 1D case, as a first step towards developing a general multidimensional framework. In particular, we are considering a series of novel transformations of the data that simultaneously allow fast access to the data, ease of compression, and do not violate the various constraints proposed by [10]. We then hope to apply it to build a multidimensional suffix array while still retaining the best-known performance bounds (both theoretically and in practice). In addition, much of the literature only discusses extensions to 2-D. We hope to develop data structures that operate for any dimension d and address these two problems:

1. Is there a multidimensional analogue to the Burrows-Wheeler transform captures spatial information and still allows multidimensional pattern matching?
2. Is it possible to achieve a multidimensional suffix array that operates on d-dimensional data in just $n^d H_h + o(n^d)$ bits with $O(\text{polylog} n^d)$ time?

References

[1] A. Amir and G. Benson. Efficient Two-Dimensional Compressed Matching. *DCC*, 1992, 279–288.

[2] A. Amir and E. Porat and M. Lewenstein. Approximate subset matching with Don't Cares. *SODA*, 2001, 305–306.

[3] A. Amir and M. Farach. Efficient 2-Dimensional Approximate Matching of of Half-rectangular Figures. Info. and Computation, 118(1):1–11, 1995.

[4] A. Amir and M. Lewenstein and E. Porat. Faster algorithms for string matching with k mismatches. *SODA*, 2000, 794–803.

[5] A. Amir and G. Benson and M. Farach. Let Sleeping Files Lie: Pattern Matching in Z-Compressed Files. *SODA*, 1994.

[6] A. Amir and G. Benson and M. Farach. Optimal Two-Dimensional Compressed Matching. *ICALP*, 1994, 215–226.

[7] P. Ferragina and G. Manzini. Opportunistic Data Structures with Applications. *FOCS*, 2000.

[8] P. Ferragina and G. Manzini. An Experimental Study of an Opportunistic Index. *SODA*, 2001.

[9] R. Giancarlo and R. Grossi. Multi-Dimensional Pattern Matching with Dimensional Wildcards. *CPM*, 1995, 90–101.

[10] R. Giancarlo and R. Grossi. On the construction of classes of of suffix trees for square matrices: algorithms and applications. Info. and Computation, 130(2):151–182, 1996.

[11] R. Grossi and A. Gupta and J. S. Vitter. High-Order Entropy-Compressed Text Indexes. *SODA*, 2003.

[12] R. Grossi and A. Gupta and J. S. Vitter. When Indexing Equals Compression: Experiments on Suffix Arrays and Trees. *SODA*, 2004.

[13] R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *STOC*, 2000, 397–406.

[14] D. Gusfield. Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology Cambridge University Press, 1997.

[15] D. Kim and Y. Kim and K. Park. Constructing Suffix Arrays for Multi-dimensional Matrices. *CPM*, 1998, 126–139.

[16] D. Kim and Y. Kim and K. Park. Generalizations of suffix arrays to multi-dimensional matrices. *TCS*, 2003.

[17] G. Navarro. The LZ-index: A Text Index Based on the Ziv-Lempel Trie. Manuscript.

[18] K. Sadakane. Compressed text databases with efficient query algorithms based on the compressed suffix array. *ISAAC*, 2000, 410–421.

[19] K. Sadakane. Succinct representations of *lcp* information and improvements in the compressed suffix arrays. *SODA*, 2002.

Fast Gapped Variants Lempel-Ziv-Welch Compression

Alberto Apostolico[1]
Dipartimento di Ingegneria dell'Informazione,
Università di Padova, Padova, Italy
and
Department of Computer Sciences,
Purdue University, Computer Sciences Building, West Lafayette, IN 47907, USA.
axa@dei.unipd.it

This talk presents variants of classical data compression paradigms by Ziv, Lempel, and Welch in which the phrases used in compression are strings of intermittently solid and wild characters. Such strings are suitably selected among patterns that are produced in turn, in a deterministic fashion, by the self-correlation of the source string. In particular, adaptations and extensions of the classical LZ78 paradigm as implemented by Welch are developed along these lines, and they are seen to be susceptible of simple linear time implementation. Both lossy and lossless schemata are considered, and preliminary analyses of performance are attempted.

[1]Work Supported in part by the Italian Ministry of University and Research under the National Projects FIRB RBNE01KNFP, and PRIN "Combinatorial and Algorithmic Methods for Pattern Discovery in Biosequences", and by the Research Program of the University of Padova.

Vcodex: A Platform of Data Transformers

Kiem-Phong Vo
AT&T Labs – Research
180 Park Avenue, Florham Park, NJ 07932, USA
kpv@research.att.com

Modern data compression methods are often built upon other existing techniques. However, such works tend to focus on the theoretical side of *algorithm engineering*, i.e., the design and analysis of the underlying data structures and algorithms while implementation quality is mostly constrained to producing working tools and often only for the few hardware/OS platforms used by the tool developers. Not much attention is given to *software engineering*, i.e., the design and implementation of standardized interfaces to make such data structures and algorithms reusable. In addition, virtually every compressor uses its own ad-hoc data format and little thought is ever given to *data engineering* so that data produced by one tool can be further processed by another. Inadequate engineering considerations make it hard to experiment with different combinations of compression techniques in solving particular data needs. Below are a few relevant aspects of algorithm, software and data engineering in building and using compression tools:

- *Algorithm engineering*: Algorithm design aims at achieving desired computational bounds while producing good compression rates. But, to maximize performance, attention must also be paid to tuning the performance of such algorithms by making effective use of hardware and OS features such as specialized instructions or available cache and memory.
- *Software engineering*: Data compressors are routinely built from smaller components. For example, the popular *gzip* compressor combines a front-end Ziv-Lempel compressor and a back-end Huffman coder. Similarly, Burrows-Wheeler compressors often contains phases based on variations of move-to-front, run-length and entropy coding. Such data transformers should be encapsulated in standardized interfaces allowing easy composition in building new compression tools.
- *Data engineering*: Large gains can be obtained by classifying data based on certain common characteristics and designing algorithms to match such classifications. Examples of this are specialized compressors such as the Pzip compressor by Buchsbaum et al. for compressing table data and the Xmill compressor by Liefke and Suciu to compress XML data. Another aspect of data engineering is to design the persistent data, i.e., the compressed output data, so that they can be easily extended and used as new data transforming techniques are invented.

Vcodex is a software platform of data transforming algorithms that can be used as building blocks to construct data compressors as well as other data processing tools. Its

main constributions include an extensible software architecture allowing easy additions of new data transformers, a flexible and self-describing data encoding format, and a number of new and efficient algorithms and heuristics for suffix sorting, compressing table data, etc. This talk discusses the software and data architectures provided by Vcodex. Examples will be given to show how to build common types of compressors based on generalized Ziv-Lempel and Burrows-Wheeler transforms as well as other exotic compressors dealing with different types of table data. Time permitting, the talk will also give overviews of the algorithms and data structures underlying the various techniques.

Remote File and Data Synchronization: State of the Art and Open Problems

Torsten Suel
CIS Department, Polytechnic University Brooklyn
suel@poly.edu

Remote file and data synchronization tools are used to efficiently maintain large replicated collections of files or record-based data in distributed environments with limited bandwidth. Common application scenarios include synchronization of data between accounts or mobile devices, content distribution and web caching networks, web site mirroring, storage networks, database replication, and large scale web search and mining. A number of techniques have been studied by academic researchers over the last few years, and many companies have developed and deployed tools for various applications.

After a general introduction, we will focus on the remote file synchronization problem, which in simplified form is stated as follows: given two versions of a file on different machines, say an outdated and a current version, how can we update the outdated version with minimum communication cost, by exploiting the significant similarity that often exists between versions? A well-known open source tool for this problem called rsync is widely used in practice, and recently researchers have proposed several possible improvements over the rsync protocol. We will describe theoretical and practical approaches to the file synchronization problem, discuss relationships to compression and coding theory, and list open challenges. We will also discuss some of our own recent work on the problem. We hope to convince the audience that these problems are important in practice and of fundamental interest, and that the compression and algorithms communities in particular have much to contribute.

Toward Ubiquitous Compression (Synopsis)

Fred Douglis
IBM T. J. Watson Research Center
douglis@acm.org

Introduction

Mark Weiser once defined ubiquitous computing as computers that blend into the environment so well that they are effectively invisible [14]. Basic compression (such as with `zip`) is starting to achieve the same level of ubiquity, but it has far to go. This paper considers the evolution of on-line compression in a number of real-world applications and conjectures on ways in which compression should continue to evolve. It is written from the perspective of a consumer of compression technologies, not one who writes the compressors in the first place.

About Compression

In this paper, compression refers to any technique that encodes data more compactly. Thus it includes not only "traditional" compression [7], which encodes an object by reducing the redundancy within it or by encoding the object relative to a well- known dictionary, but also approaches that encode different objects relative to each other. Examples include duplicate suppression [1], which replaces identical objects (at some granularity) with references to an existing copy, and delta-encoding [5], which encodes an object with respect to another specific object.

Tradeoffs

Compression requires computing resources, such as processor cycles and memory. In some environments the expenditure of these resources is clearly justified. This may be because the savings from compression are substantial, because the compressed object will be saved in compressed form for a prolonged time, or other reasons.

Some forms of compression are automatic and implicit. For example, modems typically compress data being transmitted over a phone line. By doing the compression in hardware, modems ensure that the performance of compression is comparable to the speed of transmission; .i.e, transmission is not delayed by the act of compressing or uncompressing data. For stored data, the notion of compressing data that will not be accessed for a long time has been well understood for decades.

Quite some time ago, I expounded on the trends in processing speed and suggested that as processors got faster more quickly than networks, distributed systems should consider automatically compressing data in software in an end-to-end network connection [3]. While I did not anticipate the rapid increase in network bandwidth that accompanied the

ever-increasing processor performance, and memory bandwidth is proving to be another important factor, the general approach still applies:

Systems should automatically compress data whenever the benefits from transmitting or storing compressed data outweigh the costs.

Just as modems compress transparently, distributed systems, storage systems, and especially distributed storage systems should be cognizant of the tradeoffs in dynamic compression and integrate compression techniques.

Where We Are

On-line compression, where everything that goes over a link or into or out of a file system is compressed and later uncompressed on the "critical path" to accessing the data, is by now a well-understood and commonly used technique. For several years, users have had an option to compress everything written to a Windows file system. More recently, there have been examples of other techniques to trade processing for data size, using technology beyond simple zip-style data compression, such as:

- Rsync [13], which allows a user to synchronize versions of a file on two different systems by identifying common blocks. It uses a rolling checksum to match blocks on the sender that are already contained in the receiver's copy. The same rolling checksum technique can be applied to storage systems to find duplication across files [2].
- Link-level duplicate detection [12], which stores fingerprints representing occasional substrings of streamed data and then matches repeated data by seeing the same fingerprint again.
- Storage-level duplicate detection, which uses strong checksums to find when files or blocks are stored multiple times and instead save only one copy. Blocks can be fixed-size [11] or with boundaries defined by the content itself [8], the latter preventing changes in one block from affecting subsequent ones but at a higher processing cost [10].
- Delta-encoding [5], which compresses a file or block against another file or block that is similar, by representing only the differences. It can use resemblance detection to find similar files [4, 9] or blocks [6].

I make two observations about these techniques. First, they are generally done in isolation. A system that does block-level duplicate detection may not simultaneously perform compression at the level of entire files, even though whole-file compression could exploit inter-block redundancy and dramatically reduce overall storage consumption [6]. Second, the techniques are applied all-or-none: a system that does one type of compression most likely always does that type.

Where We Should Go

Not all data are created equal. Some are accessed often; some are write-once, read-rarely-if-ever. Some compress well with "traditional" compressors; some contain duplication with other pieces of data. The same is true of computing environments, where the speed of processing, accessing a disk, or communicating over a network can be extremely variable.

It is the combination of all these factors—data contents, access patterns, and execution environment—that determines how compression should best be applied, if at all. One size does not fit all.

Ideally, one can give systems (and applications) the ability to pick and choose automatically among a suite of data reduction techniques. With my colleagues at IBM Research, I have explored and quantified the benefits of a number of techniques across a variety of data sets [6]. Ultimately, one would offer systems the ability to select compression techniques dynamically as a function of the data and the execution environment. Eventually, we may enable compression and other data reduction techniques to fade into the background, just like Weiser's ubiquitous computers.

Acknowledgments

Jason LaVoie and John Tracey provided helpful comments on this paper.

References

[1] William J. Bolosky, Scott Corbin, David Goebel, and John R. Douceur. Single instance storage in windows 2000. In *Proceedings of the 4th USENIX Windows Systems Symposium*, August 2000.

[2] Timothy E. Denehy and Windsor W. Hsu. Reliable and efficient storage of reference data. Technical Report RJ10305, IBM Research, October 2003.

[3] Fred Douglis. On the role of compression in distributed systems. In *Proceedings of the Fifth ACM SIGOPS European Workshop*. ACM, September 1992. Also appears in *ACM Operating Systems Review*, 27(2):88–93, April 1993.

[4] Fred Douglis and Arun Iyengar. Application-specific delta-encoding via resemblance detection. In *Proceedings of 2003 USENIX Technical Conference*, June 2003.

[5] David G. Korn and Kiem-Phong Vo. Engineering a differencing and compression data format. In *Proceedings of the 2002 Usenix Conference*. USENIX Association, June 2002.

[6] Purushottam Kulkarni, Fred Douglis, Jason LaVoie, and John M. Tracey. Redundancy elimination within large collections of files. In *Proceedings of the 2004 Usenix Conference*, June 2004. To appear.

[7] D. A. Lelewer and D. S. Hirschberg. Data compression. *ACM Computing, Springer Verlag (Heidelberg, FRG and NewYork NY, USA)-Verlag Surveys, ; ACM CR 8902-0069*, 19(3), 1987.

[8] Athicha Muthitacharoen, Benjie Chen, and David Mazieres. A low-bandwidth network file system. In *Symposium on Operating Systems Principles*, pages 174–187, 2001.

[9] Zan Ouyang, Nasir Memon, Torsten Suel, and Dimitre Trendafilov. Cluster-based delta compression of a collection of files. In *International Conference on Web Information Systems Engineering (WISE)*, December 2002.

[10] Calicrates Policroniades and Ian Pratt. Alternatives for detecting redundancy in storage systems data. In *Proceedings of the 2004 Usenix Conference*, June 2004.

[11] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In *Proceedings of the First USENIX Conference on File and Storage Technologies*, Monterey,CA, 2002.

[12] Neil T. Spring and David Wetherall. A protocol-independent technique for eliminating redundant network traffic. In *Proceedings of ACM SIGCOMM*, August 2000.

[13] Andrew Tridgell. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, Australian National University, 1999.

[14] Mark Weiser. Some computer science issues in ubiquitous computing. *Communications of the ACM*, 36(7):74–84, July 1993.

Block Sorting Lossless Delta Compression Algorithms

James J. Hunt

The ideas behind block sorting lossless data compression algorithm have influence not only file compression but also delta compression. Modern delta compression algorithms based on Ziv-Lempel techniques significantly outperform *diff*, a popular but older delta compressor, in terms of compression ratio. The modern compressors also correlate better with the actual difference between files without sacrificing performance.

Two different delta compression algorithms will be presented. The algorithms used different strategies for finding common blocks. These strategies will also be compared and their relationship to the Burrows-Wheeler Transform will be presented. Performance issues will also be addressed.

The performance of a delta algorithm is dependent upon the size of the difference between pairs of files. A metric using the Longest Common Subsequence (LCS) as the reference against which to measure compression will be presented. This metric applies mainly to one-dimensional data, but it may also apply to higher dimensional data that can be linearized without fragmenting typical changes.

Delta algorithms are also related to software differencing and merging. By using tokens instead of byte, one can use delta algorithms to improve differencing and merging. In order to attain accurate results, differencing and merging must respect the block structure of programs. A method to extend delta algorithms to this domain will also be presented.

Comparing Sequences with Segment Rearrangements

S. Cenk Sahinalp
Simon Fraser University

Computational genomics involves comparing DNA sequences based on similarity/distance computations for detecting evolutionary and functional relationships. Until recently available portions of published genome sequences were fairly short and sparse and the similarity between such sequences was measured based on character level differences. With the advent of whole genome sequencing technology there is emerging consensus that the measure of similarity between long DNA sequences must capture segmental rearragements found in abundance in the human genome. In this talk we will focus on block edit distance, which is defined as the smallest number of character edits (replacement and indel) and block edits (segmental duplication, deletion and translocation) to transform one sequence into another. Although it is NP hard to compute the block edit distance between two sequences, we show that it can be approximated within a constant factor in linear time via a simple one pass algorithm. The approximation method, based on the Lempel-Ziv'77 compressibility of one sequence when concatenated with the other, enhances the long suspected link between mutual compressibility and evolutionary relationship between genomic sequences.

Grammar-based Compression of DNA Sequences

Neva Cherniavsky
Department of Computer Science and Engineering
University of Washington

Richard Ladner
Department of Computer Science and Engineering
University of Washington

Grammar-based compression methods have shown success for many different types of data. The central idea behind these methods is to use a context-free grammar to represent the input text. Grammars can capture repetitions occurring far apart in the data. This is a limitation on sliding window or block sorting algorithms, such as LZ77 or bzip2.

Compression of DNA sequences is a notoriously hard problem. DNA contains only four symbols, and so can be represented by two bits per symbol. It is very hard to beat the bound of two bits per symbol. However, DNA is also known to have long repetitions that a compressor could hope to capture. Furthermore, DNA has a unique kind of repetition, because it contains both exact matches and *reverse complement* matches.

We explore the effectiveness of grammar-based compression on DNA sequences. We exploit the different types of repetition in DNA by modifying a successful grammar inference algorithm, Sequitur [3]. We then endeavor to maximize our gain in the next two stages of grammar compression: grammar encoding and entropy encoding. We present several different techniques for grammar encoding, ranging from very simple methods to more complicated pointer-based methods. All skew the data in some way to make the entropy encoding step more effective. We implement a custom arithmetic coder as the entropy coder, which is specifically designed to work well with our symbol stream. After evaluating our algorithm, we return to the grammar inference portion of the algorithm and improve the grammar by quantifying the efficiency of the rules.

After striving to optimize each stage of our compression algorithm, we conclude that grammar-based compression does not work well on DNA sequences. The best compressors for DNA are GenCompress [2] and DNACompress [1]. These algorithms, while achieving a bit rate smaller than two bits per symbol, still do not compress much more than a standard first order entropy coder. We pose this as a challenge to the compression community: is there a clever algorithm which can compress DNA significantly better than a simple first order arithmetic coder?

References

[1] Chen, X., Li, M., Ma, B., and Tromp, J. DNACompress: Fast and effective DNA sequence compression. *Bioinformatics 18*, 12 (Dec. 2002), 1696–1698.

[2] Grumbach, S., and Tahi, F. A new challenge for compression algorithms: genetic sequences. *Information Processing and Management 30*, 6 (1994), 875–886.

[3] Nevill-Manning, C. G., and Witten, I. H. Identifying hierarchical structure in sequences: A linear-time algorithm. *Journal of Artificial Intelligence Research 7* (Sept. 01 1997), 67–82.

Compression of Words over a Partially Commutative Alphabet

Serap A. Savari[†]
Department of Electrical Engineering and Computer Science
University of Michigan, Ann Arbor

Introduction

Multiprocessor configurations, distributed systems and communication networks are examples are systems consisting of a collection of distinct processes which communicate with one another but are also partly autonomous. Here certain events are allowed to occur independently while others must happen in a predetermined order. In other words, there is a *partial ordering* of events rather than a total ordering [1].

The events of sequential processes are well modeled by a string of events. A recent award-winning paper in the computer architecture literature [2] applies a grammar-based data compression scheme to the sequence of events that occur while a computer program runs and uses the hierarchical structure inferred by the algorithm to better understand the program's dynamic behavior and improve its performance. The implicit assumption in using lossless data compression for this application is that there is a well-defined total ordering of event occurrences. Trace theory [3] is one way to generalize the notion of a string in order to model the executions of concurrent processes. The sequential observer of a concurrent system is provided with a set of atomic actions together with a labeled and undirected *dependence relation* or *noncommutation graph* indicating which actions can be performed independently or concurrently. For a noncommutation graph G with vertex set V two words are *congruent* if one can be transformed into the other by a sequence of steps each of which consists of interchanging two consecutive letters that are nonadjacent vertices in G. For example, if the noncommutation graph G is given by a—b—c—d, then the two words $dabac$ and $abdca$ are equivalent since $dabac \equiv_G adbac \equiv_G adbca \equiv_G abdca$. To generalize lossless data compression to concurrent systems, [4] introduces a compression problem where it is only necessary to reproduce a string which is equivalent to the original string and provides some heuristic compression schemes. We mention in passing that this compression problem also arises in the compression of executable code [5]. In [6], we initiate a study of this compression problem from an information theoretic perspective.

Let us assume we are given a discrete, memoryless source that emits symbols belonging to the vertex set V, and let $P(v)$ denote the probability of $v \in V$. Let G denote a graph on vertex set V. Our goal is to minimize the average number of bits per symbol needed to represent the congruence class containing a word emitted from the source as the word length approaches infinity. We call the limit of the best achievable rate as L approaches

[†]This work was done while the author was with the Computing Sciences Research Center, Bell Labs, Lucent Technologies.

infinity the *interchange entropy* of the source, which exists and which we will denote by $H_\iota(G,P)$.

Basic Properties

Let E denote the edge set of a graph.

Proposition 1 (Monotonicity) *If F and G are two graphs on the same vertex set and $E(F) \subseteq E(G)$, then for any probability distribution P we have $H_\iota(F,P) \leq H_\iota(G,P)$.*

Proposition 2 (Subadditivity) *Let F and G be two graphs on the same vertex set V and define $F \cup G$ to be the graph on V with edge set $E(F) \cup E(G)$. For any fixed probability distribution P we have $H_\iota(F \cup G, P) \leq H_\iota(F,P) + H_\iota(G,P)$.*

Proposition 3 (Disjoint Components) *Let the subgraphs G_j denote the connected components of the graph G. For a probability distribution P on $V(G)$ define the probability distributions $P_j(x) = P(x)[P(V(G_j))]^{-1},\ x \in V(G_j)$. Then $H_\iota(G,P) = \sum_j P(V(G_j)) H_\iota(G_j, P_j)$.*

Theorem 6 *Suppose V is of the form $V = V_1 \cup V_2 \cup \ldots \cup V_k$ with $|V_i| = m_i,\ i \in \{1,2,\ldots,k\}$ and label the elements of V_i as $v_{i,j},\ i \in \{1,2,\ldots,k\},\ j \in \{1,2,\ldots,m_i\}$. For the complete k-partite graph $K_{m_1,m_2,\ldots,m_k}$ there is an edge corresponding to every pair of vertices $\{v_{i,j}, v_{l,n}\}, v_{i,j} \in V_i, v_{l,n} \in V_l,\ l \neq i$, and no two vertices from the same subset V_i are adjacent for any $i \in \{1,2,\ldots,k\}$. Define $Q_i = \sum_{j=1}^{m_i} P(v_{i,j}),\ i \in \{1,2,\ldots,k\}$. Then*
$H(P) - H_\iota(K_{m_1,m_2,\ldots,m_k}, P) =$

$$\sum_{S=2}^{\infty} \log_2(S) \sum_{i:m_i \geq 2} (1 - Q_i) \left({Q_i}^S - \sum_{j=1}^{m_i} \left(\frac{P(v_{i,j})}{1 - Q_i + P(v_{i,j})} \right)^S \right).$$

References

[1] L. Lamport, Time, clocks, and the ordering of events in a distributed system, *Comm. ACM*, 21, 558–565, 1978.

[2] J. Larus, Whole program paths, *ACM SIGPLAN Conf. Prog. Lang. Des. Implem.* 259–269, Atlanta, GA, May 1999.

[3] A. Mazurkiewicz, Trace theory, in W. Brauer et. al., ed., *Petri Nets, Applications and Relationship to other Models of Concurrency*, Lecture Notes in Computer Science 255, 279–324, Springer, Berlin, 1987.

[4] R. Alur, S. Chaudhuri, K. Etessami, S. Guha and M. Yannakakis, Compression of partially ordered strings, *Proc. CONCUR 2003* (14th International Conference on Concurrency Theory), Marseille, France, September 2003.

[5] M. Drinić and D. Kirovski, PPMexe: PPM for compressing software, *Proc. 1997 I.E.E.E. Data Comp. Conf.* 192–201, Snowbird, UT, March 2002.

[6] S. A. Savari, Compression of words over a partially commutative alphabet, *IEEE Transactions on Information Theory,* 50(7):1425–1441, July 2004.

Delayed-Dictionary Compression for Packet Networks

Yossi Matias
Tel Aviv University

Raanan Refua
Tel Aviv University

We consider data compression in packet networks, in which data is transmitted by partitioning it into packets. Packet compression allows better bandwidth utilization of a communication line resulting in much smaller amounts of packet drops, more simultaneous sessions, and a smooth and fast behavior of applications.

Packet compression can be obtained by a combination of *Header compression* and *payload compression*, which are complementary methods. In this work we focus on payload compression only. We are particularly interested in dictionary-based compression. Many dictionary compression algorithms were developed, following the seminal papers of Lempel and Ziv.

In dictionary compression, an input sequence is encoded based on a dictionary that is constructed dynamically according to the given text. The compression is done in a streaming fashion, enabling to leverage on redundancy in the input sequence.

In many packet networks, including ATM, Frame Relay, Wireless, and others, packets are sent via different routes, and may arrive reordered, due to different network characteristics, or due to retransmissions in case of dropped packets. Since streaming compression assumes that the compressed sequence arrives at the decoder at the order in which it was sent by the encoder, the decoder must hold packets in a buffer until all preceding packets arrive. This causes *decoding latency*, which may be unacceptable in some applications.

To alleviate decoding latency, standard packet compression techniques are based on a packet-by-packet compression. For each packet, its payload is compressed using a dictionary compression algorithm, independently to other packets. While the decoding latency is addressed properly, this may often result with poor compression quality, since the inherent redundancy within a packet is significantly smaller than the redundancy over many packets in the stream.

We introduce a novel compression algorithm suitable for packet networks: the *delayed-dictionary compression* (DDC). The DDC is a general framework that applies to any dictionary algorithm; it considers the dictionary construction and the dictionary-based parsing of the input text as separate processes, and it imposes a delay Δ in the dictionary construction. As a result, when decoding a packet, the decoder does not depend on any of the preceding Δ packets, eliminating or diminishing the problems of out-of-order packets and packet drops compared to streaming compression, still with a good traffic compression ratio.

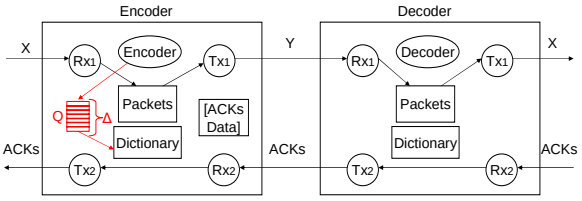

Figure 1: Internal structure of the Encoder and the Decoder: The encoder task transfers phrases to the dictionary task by using a FIFO queue.

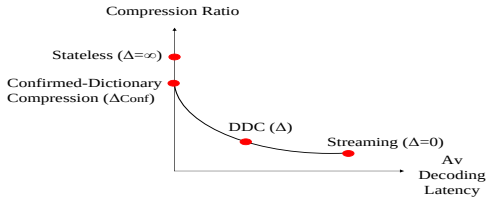

Figure 2: A tradeoff between the compression ratio and the average decoding latency. Streaming has the best compression ratio and the worst decoding latency. DDC has compression ratio close to that of streaming compression, and also has average decoding latency which is close to that of stateless. The *confirmed-dictionary compression* algorithm ensures a zero decoding latency.

The internal architecture of the encoder and the decoder combined with the DDC method is depicted in Fig. 1. The dictionary delay is implemented by a FIFO queue which is initialized to Δ dummy packets.

We focus on two alternative encoding methods for the DDC algorithm. The first adapts to the network propagation delay and the probability for packet loss. The second, called *confirmed-dictionary compression*, ensures zero decoding latency. The DDC at its most powerful version ensures that the compression ratio will be at least as good as that of stateless compression, and quite close to that of the streaming compression, with decoding latency close to or equal to that of stateless compression.

A full tradeoff between compression ratio and decoding latency can be obtained using the DDC algorithms, as illustrated in Fig. 2. On one extreme of the tradeoff is the streaming compression, which is DDC with $\Delta = 0$, it has the best compression ratio and the worst decoding latency. On the other extreme we have the DDC with confirmed dictionary, which has zero decoding latency - as in stateless compression; its compression ratio is the worst among the DDC algorithms, but is still better than that of stateless compression. Thus, the DDC has the benefits of both stateless compression and streaming compression.

With the right choices of the dictionary delay parameter, it may have a decoding latency which is close to that of stateless compression, and with compression ratio which is close to that of streaming compression. For example, for a concatenation of the Calgary corpus files, fragmented into packets with a payload of 125 bytes, in streaming the compression ratio is 0.52 with an average decoding latency of 62 packets, while in DDC with a large dictionary delay of 200 packets we obtain a compression ratio of 0.68 and only an average decoding latency of 14.3 packets.

The DDC method is particularly good for low to medium speed communication links. Its advantage is most significant for applications in which the latency is important, and in which the order of decoded packets is not important.

We conducted various experiments to establish the potential benefit of DDC, by comparing the compression ratios of streaming compression versus stateless compression, and

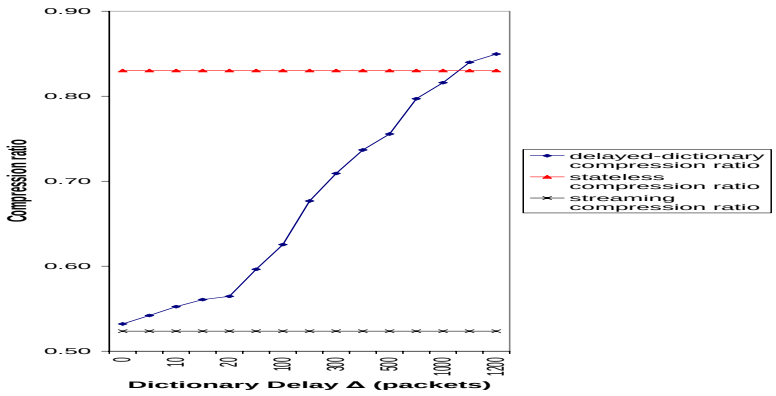

Figure 3: Compression ratio of DDC as a function of the dictionary delay in packets, compared to stateless compression ratio and to streaming compression ratio. The ratio for streaming compression is very close to the DDC ratio with zero dictionary delay. The data file in use is the concatenation of 18 Calgary corpus files, $|Header| = 20, |Payload| = 125$. DDC obtains a good compression ratio even for large dictionary delays.

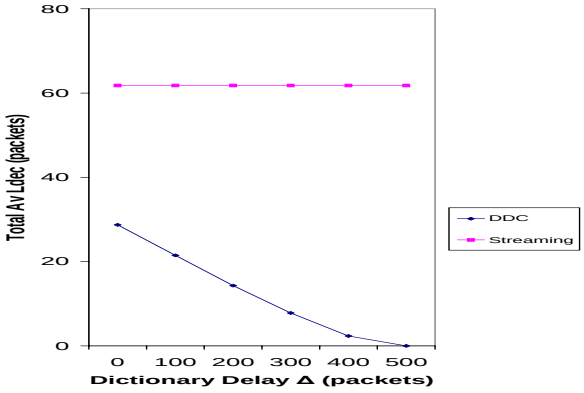

Figure 4: The Effect of the Dictionary Delay on the Decoding Latency: Increasing the dictionary delay will cause a decrease of the decoding latency L_{dec}. As can be seen, the average decoding latency in DDC is smaller than that of streaming. In particular when $\Delta = 500$ packets in DDC do not wait at all while packets in streaming wait on average for 62 packets. $RTT = 5000$msec.

by measuring the decoding latency of streaming compression. We used the Flexible Parsing version of LZW as the compression algorithm for testing purposes. For network related experiments we used the Planet Lab project over the Internet.

Using experimentation, we study the dependency of compression quality and the imposed dictionary delay, showing that the improvement in compression over stateless compression could be significant even for a relatively large dictionary delay. The compression ratios of stateless compression, streaming compression, and DDC, for small packets, are depicted in Fig. 3.

We also consider the effect of the dictionary delay on the performance in terms of the decoding latency. This effect is depicted in Fig. 4. A sufficiently large dictionary delay will practically provide a practical zero decoding latency. We compare the decoding latencies of streaming compression vs. DDC, showing that the latter is indeed considerably better. For streaming compression, the maximal decoding latency is $4RTT$ (e.g., for $RTT = 5000$msec and a payload size of 125 bytes the decoding latency is 963 packets) while in DDC we can control it to be zero.