of $n_i - 1$ largest items in $G_i$ to the size of the largest item in $G_i$ for $i = 1, \ldots, k$. The procedure satisfies the properties of Lemma 3 with $c = O(\log n_i)$ (left as an exercise to the reader). To prove Theorem 2, it suffices to show that $n_k = O(\text{Size}(I))$. This is done easily by ignoring all items smaller than $1/\text{Size}(I)$ and filling them in only in the end (as in the algorithm of de la Vega and Lueker).

In the case when the item sizes are not too small, the following corollary is obtained.

**Corollary 1** If all the item sizes are at least $\delta$, it is easily seen that $c = O(\log 1/\delta)$, and the above algorithm implies a guarantee of $\text{OPT} + O(\log(1/\delta) \cdot \log \text{OPT})$, which is $\text{OPT} + O(\log \text{OPT})$ if $\delta$ is a constant.}

### Applications

The bin packing problem is directly motivated from practice and has many natural applications such as packing items into boxes subject to weight constraints, packing files into CDs, packing television commercials into station breaks, and so on. It is widely studied in operations research and computer science. Other applications include the so-called cutting-stock problems where some material such as cloth or lumber is given in blocks of standard size from which items of certain specified size must be cut. Several variations of bin packing, such as generalizations to higher dimensions, imposing additional constraints on the algorithm and different optimization criteria, have also been extensively studied. The reader is referred to [1,2] for excellent surveys.

### Open Problems

Except for the NP-hardness, no other hardness results are known and it is possible that a polynomial-time algorithm with guarantee $\text{OPT} + 1$ exists for the problem. Resolving this is a key open question. A promising approach seems to be via the configuration LP (considered above). In fact, no instance is known for which the additive gap between the optimum configuration LP solution and the optimum integral solution is more than 1. It would be very interesting to design an instance that has an additive integrality gap of two or more.

The $\text{OPT} + O(\log^2 \text{OPT})$ guarantee of Karmarkar and Karp has been the best known result for the last 25 years, and any improvement to this would be an extremely interesting result by itself.

### Cross References

- Bin Packing
- Knapsack

### Recommended Reading


### Block Edit Distance

- Edit Distance under Block Operations 2000

### Block-Sorting Data Compression

- Burrows–Wheeler Transform

### Boolean Formulas

- Learning Automata

### Boolean Satisfiability

- Exact Algorithms for General CNF SAT

### Boosting Textual Compression

**2005; Ferragina, Giancarlo, Manzini, Sciortino**

*Paolo Ferragina*, *Giovanni Manzini* 1

1 Department of Computer Science, University of Pisa, Pisa, Italy

*Pietro Sciortino* 2

2 Department of Computer Science, University of Piemonte Orientale, Alessandria, Italy

**Synonyms**

High-order compression models; Context-aware compression

**Problem Definition**

Informally, a boosting technique is a method that, when applied to a particular class of algorithms, yields improved algorithms. The improvement must be provable and well
defined in terms of one or more of the parameters characterizing the algorithmic performance. Examples of boosters can be found in the context of Randomized Algorithms (here, a booster allows one to turn a BPP algorithm into an RP one [6]) and Computational Learning Theory (here, a booster allows one to improve the prediction accuracy of a weak learning algorithm [10]). The problem of Compression Boosting consists of designing a technique that improves the compression performance of a wide class of algorithms. In particular, the results of Ferragina et al. provide a general technique for turning a compressor that uses no context information into one that always uses the best possible context.

The classic Huffman and Arithmetic coding algorithms [1] are examples of statistical compressors which typically encode an input symbol according to its overall frequency in the data to be compressed.1 This approach is efficient and easy to implement but achieves poor compression. The compression performance of statistical compressors can be improved by adopting higher-order models that obtain better estimates for the frequencies of the input symbols. The PPM compressor [9] implements this idea by collecting (the frequency of) all symbols which follow any k-long context, and by compressing them via Arithmetic coding. The length k of the context is a parameter of the algorithm that depends on the data to be compressed: it is different if one is compressing English text, a DNA sequence, or an XML document. There exist other examples of sophisticated compressors that use context information in an implicit way, such as Lempel–Ziv and Burrows–Wheeler compressors [9]. All these context-aware algorithms are effective in terms of compression performance, but are usually rather complex to implement and difficult to analyze.

Applying the boosting technique of Ferragina et al. to Huffman or Arithmetic Coding yields a new compression algorithm with the following features: (i) the new algorithm uses the boosted compressor as a black box, (ii) the new algorithm compresses in a PPM-like style, automatically choosing the optimal value of k, (iii) the new algorithm has essentially the same time/space asymptotic performance of the boosted compressor. The following sections give a precise and formal treatment of the three properties (i)–(iii) outlined above.

---

1In their dynamic versions these algorithms consider the frequency of a symbol in the already scanned portion of the input.

**Key Results**

**Notation: The Empirical Entropy**

Let s be a string over the alphabet \( \Sigma = \{a_1, \ldots, a_k\} \) and, for each \( a_i \in \Sigma \), let \( n_i \) be the number of occurrences of \( a_i \) in \( s \). The 0th order empirical entropy of the string \( s \) is defined as

\[
H_0(s) = -\sum_{i=1}^{k} \frac{n_i}{|s|} \log(\frac{n_i}{|s|})
\]

where it is assumed that all logarithms are taken to the base 2 and \( 0 \log 0 = 0 \). It is well known that \( H_0 \) is the maximum compression one can achieve using a uniquely decodable code in which a fixed codeword is assigned to each alphabet symbol. Greater compression is achievable if the codeword of a symbol depends on the \( k \) symbols following it (namely, its context).2 Let us define \( w_i \) as the string of single symbols immediately preceding the occurrences of \( w \) in \( s \). For example, for \( s = \text{bcabcabdca} \) it is \( \text{ca}_s = \text{bbd} \). The value

\[
H_k(s) = \frac{1}{|s|} \sum_{w \in \Sigma^k} |w| H_0(w_i)
\]

is the \( k \)-th order empirical entropy of \( s \) and is a lower bound to the compression one can achieve using code-words which only depend on the \( k \) symbols immediately following the one to be encoded.

**Example 1**

Let \( s = \text{mississippi} \). For \( k = 1 \) it is \( i_s = \text{msssp}, s_i = \text{isis}, p_i = \text{ip} \). Hence,

\[
H_1(s) = \frac{4}{11} H_0(\text{msssp}) + \frac{4}{11} H_0(\text{isis}) + \frac{2}{11} H_0(\text{ip})
\]

\[
= \frac{6}{11} + \frac{4}{11} + \frac{2}{11} = \frac{12}{11}.
\]

Note that the empirical entropy is defined for any string and can be used to measure the performance of compression algorithms without any assumption on the input source. Unfortunately, for some (highly compressible) strings, the empirical entropy provides a lower bound that is too conservative. For example, for \( s = a^n \) it is \( |s| H_1(s) = 0 \) for any \( k \geq 0 \). To better deal with highly compressible strings [7] introduced the notion of 0th order modified empirical entropy \( H^*_{0}(s) \) whose property is that \( |s| H^*_{0}(s) \) is at least equal to the number of bits needed to write down the length of \( s \) in binary. The \( k \)th order modified empirical entropy \( H^*_{k}(s) \) is then defined in terms of \( H^*_{0}(s) \) as the maximum compression one can achieve by looking at no more than \( k \) symbols following the one to be encoded.

---

2In data compression it is customary to define the context looking at the symbols preceding the one to be encoded. The present entry uses the non-standard “forward” contexts to simplify the notation of the following sections. Note that working with “forward” contexts is equivalent to working with the traditional “backward” contexts on the string reversed (see [3] for details).
Boosting Textual Compression

The Burrows–Wheeler Transform

Given a string $s$, the Burrows–Wheeler transform [2] (BWT) consists of three basic steps: (1) append to the end of $s$ a special symbol $\$ \text{ smaller than any other symbol in } \Sigma$; (2) form a conceptual matrix $M$ whose rows are the cyclic shifts of the string $s\$, sorted in lexicographic order; (3) construct the transformed text $\hat{s} = \text{bwt}(s)$ by taking the last column of $M$ (see Fig. 1). In [2] Burrows and Wheeler proved that $\hat{s}$ is a permutation of $s$, and that from $\hat{s}$ it is possible to recover $s$ in $O(|s|)$ time.

To see the power of the BWT the reader should reason in terms of empirical entropy. Fix a positive integer $k$. The first $k$ columns of the BWT matrix contain, lexicographically ordered, all length-$k$ first substrings of $s$ (and $k$ substrings containing the symbol $\$). For any length-$k$ substring $w$ of $s$, the symbols immediately preceding every occurrence of $w$ in $\hat{s}$ are grouped together in a set of consecutive positions of $\hat{s}$ since they are the last symbols of the rows of $M$ prefixed by $w$. Using the notation introduced for defining $H_k$, it is possible to rephrase this property by saying that the symbols of $w$ are consecutive within $\hat{s}$, or equivalently, that $\hat{s}$ contains, as a substring, a permutation $\pi_w(w)$ of the string $w$.

Example 2 Let $s = \text{mississippi}$ and $k = 1$. Figure 1 shows that $\hat{s}[1,4] = \text{psms}$ is a permutation of $i$, $\text{missi}$. In addition, $\hat{s}[6,7] = \text{pi}$ is a permutation of $p$, $\text{ip}$, and $\hat{s}[8,11] = \text{issi}$ is a permutation of $a$, $\text{issi}$.

Since permuting a string does not change its (modified) $0$th order empirical entropy (that is, $H_0(\pi_w(w)) = H_0(w)$), the Burrows–Wheeler transform can be seen as a tool for reducing the problem of compressing $s$ up to its $k$th order entropy to the problem of compressing distinct portions of $\hat{s}$ up to their $0$th order entropy. To see this, assume partitioning of $\hat{s}$ into the substrings $\pi_w(w)$ by varying $w$ over $\Sigma^k$. It follows that $\hat{s} = \bigcup_{w \in \Sigma^k} \pi_w(w)$ where $\bigcup$ denotes the concatenation operator among strings.\footnote{In addition to $\bigcup_{w \in \Sigma^k} \pi_w(w)$, the string $\hat{s}$ also contains the last $k$ symbols of $s$ (which do not belong to any $w$) and the special symbol $.}$

By (1) it follows that

$$\sum_{w \in \Sigma^k} |\pi_w(w)| H_0(\pi_w(w)) = \sum_{w \in \Sigma^k} |w| H_0(w) = |s| H_k(s).$$

Hence, to compress $s$ up to $|s|$ $H_k(s)$ it suffices to compress each substring $\pi_w(w)$ up to its $0$th order empirical entropy. Note, however, that in the above scheme the parameter $k$ must be chosen in advance. Moreover, a similar scheme cannot be applied to $H_k^* (s)$ which is defined in terms of contexts of length at most $k$. As a result, no efficient procedure is known for computing the partition of $\hat{s}$ corresponding to $H_k^*(s)$. The compression booster [3] is a natural complement to the BWT and allows one to compress any string $s$ up to $H_k(s)$ (or $H_k^*(s)$) simultaneously for all $k \geq 0$.

The Compression Boosting Algorithm

A crucial ingredient of compression boosting is the relationship between the BWT matrix and the suffix tree data structure. Let $T$ denote the suffix tree of the string $s\$. $T$ has $|s| + 1$ leaves, one per suffix of $s\$, and edges labeled
with substrings of $s$ (see Fig. 1). Any node $u$ of $T$ has implicitly associated a substring of $s$, given by the concatenation of the edge labels on the downward path from the root of $T$ to $u$. In that implicit association, the leaves of $T$ correspond to the suffixes of $s$. Assume that the suffix tree edges are sorted lexicographically. Since each row of the bwt matrix is prefixed by one suffix of $s$ and rows are lexicographically sorted, the $i$th leaf (counting from the left) of the suffix tree corresponds to the $i$th row of the bwt matrix. Associate to the $i$th leaf of $T$ the $i$th symbol of $s = \text{bwt}(s)$. In Fig. 1 these symbols are represented inside circles.

For any suffix tree node $u$, let $\hat{s}(u)$ denote the substring of $s$ obtained by concatenating, from left to right, the symbols associated to the leaves descending from node $u$. Of course $\hat{s}(\text{root}(T)) = s$. A subset $L$ of $T$’s nodes is called a leaf cover if every leaf of the suffix tree has a unique ancestor in $L$. Any leaf cover $L = \{u_1, \ldots, u_p\}$ naturally induces a partition of the leaves of $T$. Because of the relationship between $T$ and the bwt matrix this is also a partition of $s$, namely $\{\hat{s}(u_1), \ldots, \hat{s}(u_p)\}$.

Example 3 Consider the suffix tree in Fig. 1. A leaf cover consists of all nodes of depth one. The partition of $s$ induced by this leaf cover is $\{i, psam, $, $, pi, ssii\}$.

Let $C$ denote a function that associates to every string $x$ over $\Sigma \cup \{\}$ a positive real value $C(x)$. For any leaf cover $L$, define its cost as $C(L) = \sum_{u \in L} C(\hat{s}(u))$. In other words, the cost of the leaf cover $L$ is equal to the sum of the costs of the strings in the partition induced by $L$. A leaf cover $L_{\text{min}}$ is called optimal with respect to $C$ if $C(L_{\text{min}}) \leq C(L)$, for any leaf cover $L$.

Let $A$ be a compressor such that, for any string $x$, its output size is bounded by $|x| H_0(x) + \eta |x| + \mu$ bits, where $\eta$ and $\mu$ are constants. Define the cost function $C_A(x) = |x| H_0(x) + \eta |x| + \mu$. In [3] Ferragina et al. exhibit a linear-time greedy algorithm that computes the optimal leaf cover $L_{\text{min}}$ with respect to $C_A$. The authors of [3] also show that, for any $k \geq 0$, there exists a leaf cover $L_k$ of cost $C_A(L_k) = |x| H_k(s) + \eta |s| + O(\Sigma^k)$. These two crucial observations show that, if one uses $A$ to compress each substring in the partition induced by the optimal leaf cover $L_{\text{min}}$, the total output size is bounded in terms of $|x| H_k(s)$, for any $k \geq 0$. In fact,

$$\sum_{u \in L_{\text{min}}} C_A(\hat{s}(u)) = C_A(L_{\text{min}}) \leq C_A(L_k) = |x| H_k(s) + \eta |s| + O(\Sigma^k)$$

In summary, boosting the compressor $A$ over the string $s$ consists of three main steps:

1. Compute $\hat{s} = \text{bwt}(s)$;
2. Compute the optimal leaf cover $L_{\text{min}}$ with respect to $C_A$, and partition $\hat{s}$ according to $L_{\text{min}}$;
3. Compress each substring of the partition using the algorithm $A$.

So the boosting paradigm reduces the design of effective compressors that use context information, to the (usually easier) design of 0th order compressors. The performance of this paradigm is summarized by the following theorem.

Theorem 1 (Ferragina et al. 2005) Let $A$ be a compressor that squeezes any string $x$ in at most $|x| H_0(x) + \eta |x| + \mu$ bits. The compression booster applied to $A$ produces an output whose size is bounded by $|x| H_k(s) + \log |s| + \eta |s| + O(\Sigma^k)$ bits simultaneously for all $k \geq 0$. With respect to $A$, the booster introduces a space overhead of $O(|s| \log |s|)$ bits and no asymptotic time overhead in the compression process.

A similar result holds for the modified entropy $H^*_k$ as well (but it is much harder to prove): Given a compressor $A$ that squeezes any string $x$ in at most $\lambda |x| H^*_k(x) + \mu$ bits, the compression booster produces an output whose size is bounded by $\lambda |x| H^*_k(x) + \log |s| + O(\Sigma^k)$ bits, simultaneously for all $k \geq 0$. In [3] the authors also show that no compression algorithm, satisfying some mild assumptions on its inner working, can achieve a similar bound in which both the multiplicative factor $\lambda$ and the additive logarithmic term are dropped simultaneously. Furthermore [3] proposes an instantiation of the booster which compresses any string $x$ in at most $2.5 |s| H^*_k(s) + \log |s| + O(\Sigma^k)$ bits. This bound is analytically superior to the bounds proven for the best existing compressors including Lempel–Ziv, Burrows–Wheeler, and PPM compressors.

Applications

Apart from the natural application in data compression, compressor boosting has been used also to design Compressed Full-text Indexes [8].

Open Problems

The boosting paradigm may be generalized as follows: Given a compressor $A$, find a permutation $P$ for the symbols of the string $s$ and a partitioning strategy such that the boosting approach, applied to them, minimizes the output size. These pages have provided convincing evidence that the Burrows–Wheeler Transform is an elegant and efficient permutation $P$. Surprisingly enough, other classic Data Compression problems fall into this framework: Shortest Common Superstring (which is MAX-SNP hard), Run Length Encoding for a Set of Strings (which is polynomially solvable), LZ77 and minimum number of phrases...
Bounded Degree Spanning Trees

(which is MAX-SNP-Hard). Therefore, the boosting approach is general enough to deserve further theoretical and practical attention [5].

Experimental Results
An investigation of several compression algorithms based on boosting, and a comparison with other state-of-the-art compressors is presented in [4]. The experiments show that the boosting technique is more robust than other bwt-based approaches, and works well even with less effective 0th order compressors. However, these positive features are achieved using more (time and space) resources.

Data Sets

URL to Code
The Compression Boosting page (http://www.mfn.unipmn.it/~manzini/boosting) contains the source code of all the algorithms tested in [4]. The code is organized in a highly modular library that can be used to boost any compressor even without knowing the bwt or the boosting procedure.

Cross References
- Arithmetic Coding
- Burrows–Wheeler Transform
- Compressed Text Indexing
- Table Compression
- Tree Compression and Indexing

Recommended Reading