Improved Antidictionary Based Compression*

Maxime Crochemore
Institut Gaspard-Monge, France; King's College, London.
mac@univ-mlv.fr

Gonzalo Navarro
Dept. of Computer Science, University of Chile.
gnavarro@dcc.uchile.cl

Abstract

The compression of binary texts using antidictionaries is a novel technique based on the fact that some substrings (called "antifactors") never appear in the text. Let sb be an antifactor, where b is its last bit. Every time s appears in the text we know that the next bit is $\overline{b}$ and hence omit its representation. Since building the set of all antifactors is space consuming at compression time, it is customary to limit the maximum length of antifactors considered up to a constant k. Larger k yields better compression of the text but requires more space at compression time.

In this paper we introduce the notion of almost antifactors*, which are strings that rarely appear in the text. More formally, almost antifactors are strings that, if we consider them as antifactors and separately code their occurrences as exceptions, the compression ratio improves. We show that almost antifactors permit improving compression with a limited amount of main memory to compress. Our experiments show that they obtain the same compression of the classical algorithm using only 30%–55% of its memory space.*

1. Introduction

Text compression aims at representing a text using less space [3]. The compression of binary texts using antidictionaries [4, 5] is a novel technique based on the fact that some substrings (called "antifactors") never appear in the text. Let sb be an antifactor, where b is its last bit. Every time s appears in the text we know that the next bit is $\overline{b}$ and hence omit its representation. The set of antifactors used to compress a text is called the *antidictionary*.

In a semi-static setup, a suitable antidictionary is obtained from the text in a first pass and the text is compressed using the antidictionary in a second pass. The compressed file contains the antidictionary plus the compressed text. Adding an antifactor to the antidictionary reduces the size of the compressed text but increases the size of the antidictionary and hence its representation in the compressed file. A simple algorithm to compute the net gain obtained by each antifactor and to choose the useful ones (those with positive gain) exists [5], thus yielding the "optimal" antidictionary.

In general, the method is slow to compress and fast to decompress, which is good because a text is usually compressed once and decompressed many times. So we do not care much about compression performance. However, a more important disadvantage of the method is the amount of memory used at compression time. Despite that the final set of useful antifactors is normally small, we have to find and store all the antifactors first and then choose those with positive net gain. A way to reduce the amount of memory used is to limit the length of the antifactors considered up to a constant k. A larger k yields better compression of the text but needs more memory to compress the text.

In this paper we introduce the notion of *almost antifactors*, which are strings that rarely appear in the text. More formally, almost antifactors are strings (either normal antifactors or text factors) that, if we consider them as antifactors and separately code their occurrences as exceptions, we obtain improved compression ratios.

Our experiments show that almost antifactors are especially useful to improve compression when there is a limited amount of main memory to compress, that is, when low k values have to be used. In these cases we obtain the same compression of the classical algorithm using 30%–55% of the memory space.

*Partially supported by ECOS/Conicyt Project C99E04 and Fondecyt grant 1-020831.

2. Compression Using Antidictionaries

In this section we briefly explain the algorithm to compress a binary text using antidictionaries. For a full description see [4, 5].

We represent the set of text factors using a binary trie data structure [7, 2]. The value k is then the maximum height allowed for the trie. This trie is built in a single pass over the text using standard techniques, which in particular involve storing a *suffix link* on each node. A suffix link connects the node that represents bs to the node that represents s, where b is a single bit. For example, a simple $O(kn)$ time construction algorithm traverses the text and keeps a cursor at the trie node that represents the last length-k suffix read, $b_1 \ldots b_k$. When a new text character b is read, it uses the suffix link to reach the node that represents $b_2 \ldots b_k$ and then creates its child $b_2 \ldots b_k b$, if it does not already exist (this is the new cursor position). If the node did not exist already, it repeats the process creating $b_3 \ldots b_k b$, $b_4 \ldots b_k b$, and so on, until the corresponding node exists (and hence all its remaining suffixes already exist) or until it creates the node representing just b.

Once this trie is built, we have *internal* and *external* nodes. The former correspond to nodes actually represented in the tree, that is, to factors of the text. The external nodes correspond to antifactors, and they are implicitly represented in the tree by the null pointers that are children of internal nodes. The exception are the (forcedly) external nodes at depth $k+1$, that are children of internal nodes at the maximum depth k, which may or may not be antifactors.

We transform this trie of text factors into a *trie of antifactors*: Each external node that surely corresponds to an antifactor (i.e., at depth $\leq k$) is converted into an internal (leaf) node. These new internal nodes are called *terminal*. Note that not all leaves are terminal, as some leaves at depth k are not antifactors.

For compression, this trie is later turned into an automaton [1] that recognizes antifactors: basically, every time we reach a node with a terminal child, this child represents an antifactor, so we can omit the bit in the compressed text representation. Similarly, for decompression, the trie is converted into a transducer that completes the missing bits. For this paper it is enough to think in terms of the trie of antifactors. In the compressed file we store the trie of antifactors and the compressed text. The text is given as a pair (s, n), where s is the text with some bits omitted as explained and n is the uncompressed text length. The value n is necessary for technical reasons, for example the text 10^{n-1} with $k = 2$ can be represented as the antifactor set $\{01, 11\}$ and the pair $(1, n)$.

An important consideration when converting external nodes into antifactors is to avoid representing antifactors that are suffixes of others, as the shorter is enough to omit all the corresponding bits, and storing the longer one unnecessarily increases the antidictionary size. Hence, the process of converting external nodes into antifactors is done in breath-first top-down order in the tree, so as to get the shorter antifactors first. Before converting an external node bs into a terminal node, we use its suffix link to check that s is an internal node and is not terminal. This ensures that, given external nodes representing strings $s,\ b_1 s,\ b_2 b_1 s,\ \ldots$, only s will be converted into terminal: $b_1 s$ will not because its suffix link points to the terminal node s, $b_2 b_1 s$ will not because its suffix link points to an external node (since $b_1 s$ was left as an external node), and so on.

3. Optimal Antidictionaries

As we let k grow, more and more antifactors are found. This allows us to omit more and more bits in the compressed representation of the text. On the other hand, we need to code the new antifactors in the compressed file. Therefore the challenge of finding an optimal subset of antifactors is raised.

In this section we present an algorithm [5] that, given a binary tree of antifactors, obtains an optimal subset of it, in the sense that no other subset will obtain better compression when both the antidictionary representation and the compressed text are considered. Combined with the previous technique that first finds all the antifactors of length up to k, this scheme improves monotonically as k grows, because it takes the optimal subset of larger and larger sets.

The main concept is that it is possible to exactly predict the number of bits that are saved in the compressed text thanks to a given antifactor sb. This is precisely the number of occurrences of its longest proper prefix s in the original text, because each time this prefix appears the compressor avoids coding the next bit $\overline{b}$ thanks to the existence of the antifactor sb.

On the other hand, we need to know how many bits we need in order to code the antifactor in the compressed file, for which we have to fix a coding scheme. We code the antidictionary as follows: we traverse the tree in depth-first order and write a 1 bit each time we find an internal node and a 0 bit each time we find an external node. Therefore, a binary tree of m internal nodes will need exactly $2m+1$ bits. This is almost identical to the scheme used in [5].

The algorithm works recursively on the binary tree of antifactors. The goal is to compute the *gain* that is obtained by storing each subtree. If the *gain* is negative then the subtree is pruned, otherwise it is kept. The final non-pruned subtree is the optimal set of antifactors.

To compute the *gain* of a node, we need to have computed the *gains* of its two children. Only the terminal nodes of the tree (the antifactors) add a positive term to the *gain*,

namely the number of occurrences of their longest proper prefix. Being leaves, they also decrement their *gain* in 3 bits. The *gain* of an internal node is the sum of the *gains* of its two children minus one bit to represent the internal node. Note that it is possible that an antifactor that at first seems interesting turns out to be unconvenient when higher nodes of the tree are considered.

Figure 1 depicts the algorithm, which takes time linear in the size of the input. The algorithm removes unconvenient subtrees on the fly and leaves the optimal subtree. It also returns the number of bits that will be saved thanks to the compression. In the worst case it can return -1, which means that an empty antidictionary is the best choice. In that case the text will not be altered but an extra bit to code the empty antidictionary will be necessary.

```
Optimize (A)

if A is an external node then return −1
if A is terminal
   then gain ← occParent(A) − 3
   else gain ←  Optimize (child_0(A))
               + Optimize (child_1(A)) − 1
if gain > −1 then return gain
remove the whole tree rooted at A
return −1
```

Figure 1. The algorithm that leaves an optimal antidictionary from an initial set of candidates. The children of A are referred to as $child_0$ and $child_1$. Function $occParent$ gives the number of times that the factor represented by the parent node of A appears in the text. The algorithm returns the number of bits that the compression will save.

What is left is the algorithm to compute $occParent$. This is the number of occurrences of the parent node of A, so it is enough to be able to keep count of the number of times every trie node has appeared in the text. This is obtained as a modification of the algorithm that obtains all the text factors of length up to k. Each time a new text bit is read we add 1 to the number of occurrences of the cursor node. At the end, each text position has been counted exactly once and added to the number of occurrences of the longest text factor ending at that position. The number of occurrences of the factors of length k is already correct, but shorter factors s still need that we add the occurrences of other factors $s's$ they are suffixes of.

This is solved using the suffix links. These induce a topological ordering in the trie nodes, where the leaves at depth k are the first nodes that can be processed because they receive no suffix links. We traverse the nodes in this topological order and, for every suffix link $bs \to s$, we add the occurrences of node bs to those of s. This algorithm correctly computes the number of occurrences of all the text factors in time proportional to the size of the tree.

In the rest of the paper this will be called the "classical" algorithm.

4. Almost Antifactors

Let us consider this odd behavior of antidictionaries: if we try to compress the string 10^{n-1} with $k \geq 2$, then the result is very good because we can use, for example, $\{01, 11\}$ as our antidictionary. This permits compressing the string to $(1, n)$ plus the small antidictionary. However, if we reverse the string to $0^{n-1}1$, then for any $k < n$ the antidictionary contains $\{10, 11\}$, which indeed does not yield any compression. For example, the classical algorithm yields an empty antidictionary. Yet, both strings have the same entropy.

The main problem is that a single occurrence of a string in the text (in our second example the string 01) outrules it as an antifactor. In a less extreme case, it may be possible that a string sb appears just a few times in the text, but its prefix s appears so many times that it is better to consider sb as an antifactor. Of course, to be able to recover the original text, we need to code somehow those text positions where the bit predicted by taking the string as an antifactor is wrong. We call *exceptions* the positions in the original text where this happens, that is, the final positions of the occurrences of sb in the text.

Let us assume that a given string s appears m times in the text, and that $s0$ and $s1$ appear m_0 and m_1 times, respectively, so that $m = m_0 + m_1$ (except if s is at the end of the text, where $m = m_0 + m_1 + 1$). Let us assume that we need e bits to code an exception. Hence, if $m > e \times m_0$, then we improve the compression by considering $s0$ as an antifactor (similarly with $s1$). Note that in the computation we have used the fact that we omit the bit even when we are wrong, although in this case we pay e bits for reversing the mistake. We call *almost antifactors* those strings that improve compression if taken as antifactors. This, of course, includes true antifactors that have no other antifactor as a suffix.

In this scheme we generate two files: one is the same as before (antidictionary plus compressed text) and the other is the exceptions file, which will be read in parallel with the classical file. Many different coding formats are possible for the exceptions file. We have chosen to code the differences between consecutive positions of exceptions using the 8:1 format. This format codes a number as a sequence of bytes, from which 7 bits are used to code data and the eighth signals the end of the number.

We build on top of the previous algorithm. The idea is that we can compute a *gain* for all internal nodes as if they were terminal. If a given node A is taken as an antifactor (and hence converted into a leaf node), then the gain that it will produce is

$$occParent(A) - e \times occ(A) \quad (1)$$

where $occ(A)$ is the number of times A occurs in the text. The first term corresponds to the number of times a (correct or not) bit prediction will be made[1], while the second term is the penalty to code the exceptions. Note that this gain is positive only if A appears much fewer times than its parent node.

Now, for each internal node, we have the choice of leaving it as is or converting it into terminal (and hence deleting its subtree). To make this decision, we have to compare the gain it produces as an internal node versus the gain it produces as a terminal node (Eq. (1)). On top of this, we apply the normal optimization algorithm that discards subtrees whose overall gain is not positive. Just like the optimization algorithm, all this process has to be done bottom-up, so that the gains of the subtrees are already computed when we make the decision about the current node.

The problem is that we also have to guarantee that none of these newly created terminal nodes are suffixes of others (they could be suffixes of real antifactors as well). Otherwise, our *gain* computations of Eq. (1) would be optimistic, since they will add up text occurrences that are not disjoint. When we deal with real antifactors, we can detect at creation time whether we are trying to create an antifactor whose suffix is already an antifactor. Moreover, in that case it is clear that the shorter antifactor is the best choice[2]. This is independent on the optimization of leaving the antifactors of nonnegative gain, which is done later.

For this sake, we would like to process the tree top down, so as to decide first which of the shorter substrings will be almost antifactors, and later see if a node has a suffix that is already terminal prior to considering converting it into terminal as well. But even this is not a complete solution now: given two terminal nodes, one a suffix of the other, it is not immediate which is better: the longer one could produce a larger gain, but whether it will be finally left in the tree or not will be known later, when we process the higher nodes (as an ancestor could have negative gain or be converted into terminal).

The key problem is that the decision of what is an almost antifactor depends in turn on the gains produced, so we cannot separate the process of creating the almost antifactors and computing their gains: creating an almost antifactor changes the gains upwards in the tree, as well as the gains downwards via (reversed) suffix links. So there seems to be no suitable traversal order. It is not possible either to do a first pass computing gains and then a second pass deciding which will be terminals, because if one converts a node into terminal its gain changes and modifies those of all the ancestors in the tree. It is not possible to leave the removal of redundant terminals for later because the removal can also change previous decisions on ancestors of the removed node.

We have resorted to heuristic solutions, some more sophisticated (and costly in compression time) than others. We present now the two most successful ones.

4.1. A One-Pass Heuristic

A simple heuristic solution is to (1) determine terminal nodes based solely on the gains of Eq. (1), without considering their possible gains as internal nodes; and (2) give preference always to the shorter almost antifactor, which is a good guess. That is, if a node produces a positive gain when converted into terminal, it is converted into terminal unless it has already a terminal suffix. So we first perform a breadth-first top-down traversal determining which nodes will be terminal, and then apply the normal bottom-up optimization algorithm to compute gains and decide which nodes deserve belonging to the antidictionary.

Note that, unlike the case of exact antifactors, using the suffix link is not enough to determine whether a suffix of the current node is already terminal. Since antifactors were external nodes, the first extension bs of a terminal node s was left as an external node because of its suffix link to s. The next extension $b'bs$ and the others had suffix links pointing to external nodes and knew that they could not be terminals either. However, with almost antifactors, an internal node may become terminal, and a node may have to traverse several suffix links through internal nodes in order to reach a suffix terminal node.

Hence we use a transitive mechanism: Every time we create a terminal node, we mark it as transitively-terminal too. Every time we process a node, if its suffix link points to a transitively-terminal node, we mark it as transitively-terminal as well and forbid making the node terminal.

An advantage of this heuristic is that it produces reasonable solutions by performing the same number of passes over the tree as the classical method. It also needs the same memory space.

We also tried the alternative of computing gains without paying attention to suffixes dependencies, but the results were worse. This suggests that the errors for ignoring these dependencies are significative, and that it is better to give at

[1] This may be off by 1 if A appears at the end of the text, but the effect of this case is negligible and we ignore it here for simplicity.

[2] In fact, in pathological cases more compression could be obtained by leaving the longer antifactor, if it turns out to need less bits to be represented. This has not been considered up to now.

least a simplified treatment to the problem.

4.2. A Multi-Pass Heuristic

A more sophisticated heuristic takes into account the facts that (1) it may be a bad decision to convert into terminal a node that turns out to have a subtree with a large gain, hence losing it; and (2) giving the preference to the highest node is not necessarily the best choice.

The idea is to perform several passes over the trie, refining the gain estimations until we converge (that is, the last pass makes no changes). In a first pass we compute gains using the classical antifactor mechanism. In successive passes, we use the gains computed in the previous pass to determine the gain an internal node produces, so as to compare that gain against that of converting the node into terminal. If the node gives a higher gain as a terminal node, it is converted into terminal and its gain is recomputed (its subtree is not physically deleted because the decision can be reverted in further passes). All the passes are breadth first top-down, since the gains are already computed in the previous pass.

We use the same transitive mechanism to avoid making terminal nodes with a suffix that is already terminal in the current pass. Note that since we recompute the gains in the wrong order (top-down), these are not consistent when we finish. Hence, after each pass, we perform a bottom-up recomputation of the gains, given the nodes that have been declared as terminal in the current pass.

With respect to giving the preference always to the highest node, we keep this decision but permit correcting it in further passes. Every time a node $s's$ wants to become terminal and cannot because s is already terminal, it compares its gain as a terminal node against that of s. If the gain $s's$ could obtain is higher, it annotates this gain in a "*suffix-gain*" field of s. In the next pass, if s wants to become terminal but its *suffix-gain* value is larger than the potential gain of s, this means that s is preventing a more productive node from being terminal, so s does not become terminal. This opens the door for $s's$ to become terminal in this pass.

Note that this is just a heuristic, because $s's$ could later be removed at a higher level, and then we lose both gains: those of s and $s's$. However, this has worked better than giving preference always to the lower node.

The method has the disadvantage of requiring several passes over the trie until converging, but it does improve the compression obtained. With respect to memory usage, it is the same as for the classical algorithm. The new fields introduced can share space with existing values that are used in different moments in the algorithm.

5. Experimental Results

We have implemented the classical algorithm and both heuristics using almost antifactors, together with their decompressors. We store both the antidictionary (using the storage scheme of Section 3) and the compressed text into the compressed file. The exceptions file is separated because it has to be accessed in parallel with the classical file. Our implementation, in C under Linux, is careful with the space usage. We have applied the algorithms to 100 Kbytes (i.e., 800 Kbits) of articles extracted from The Wall Street Journal 1989 [6].

Figure 2 shows the behavior of the classical algorithm as k grows. On top, we show the sizes of both parts of the compressed file (the antidictionary representation and the compressed text). On the bottom we show the amount of main memory needed to compress the file. As it can be seen, reaching $k = 40$ is necessary to achieve good compression (so the compressed file reaches 40% of the original file). However, this requires a lot of main memory (more than 70 Mbytes), if we consider that we are compressing just 100 Kbytes. The algorithm has naturally low locality of reference when manipulating the set of antidictionaries, so this means that we need 70 Mbytes of main memory in order to have reasonable compression time. Note that this figure would worsen with larger files.

We consider now the effect of almost antifactors. As the expected length of an exception has been left as a parameter, we have manually found the best value for each k. The more bits we assume are necessary to code an exception, the less almost antifactors are used. Ideally, the best performance would be achieved when the expected length we give and the resulting length matched, but since the heuristic decisions taken are not perfect and tend to produce more terminal nodes than the ideal, the optimum compression is achieved with larger e values. For example, the typical length of an exception is around 9–14 bits, but the optima are reached with values that are typically in the range 9–65 for the one-pass heuristic and 9–30 in the multi-pass. This shows that the latter makes better decisions.

Figure 3 shows the behavior of the different parts of the compressed representation. The maximum size of the exceptions file does never reach 15% of the compressed text using the one-pass heuristic, while it has a more erratic behavior, surpassing 20% at times, on the multi-pass. In both cases, there is no monotonic behavior in the optimal size of the exceptions file. After reaching a maximum, its size reduces as k grows. This means that the importance of almost antifactors is higher for small k, or which is the same, when the resources for compression are limited.

Figure 4 (top) compares the compression ratios obtained with the classical algorithm against those obtained with the two almost antifactor heuristics. The one-pass heuristic is

moderately successful: in the best case it improves the classical compression by less than 8%. However, it becomes worse than the classical compression for $k \geq 33$.

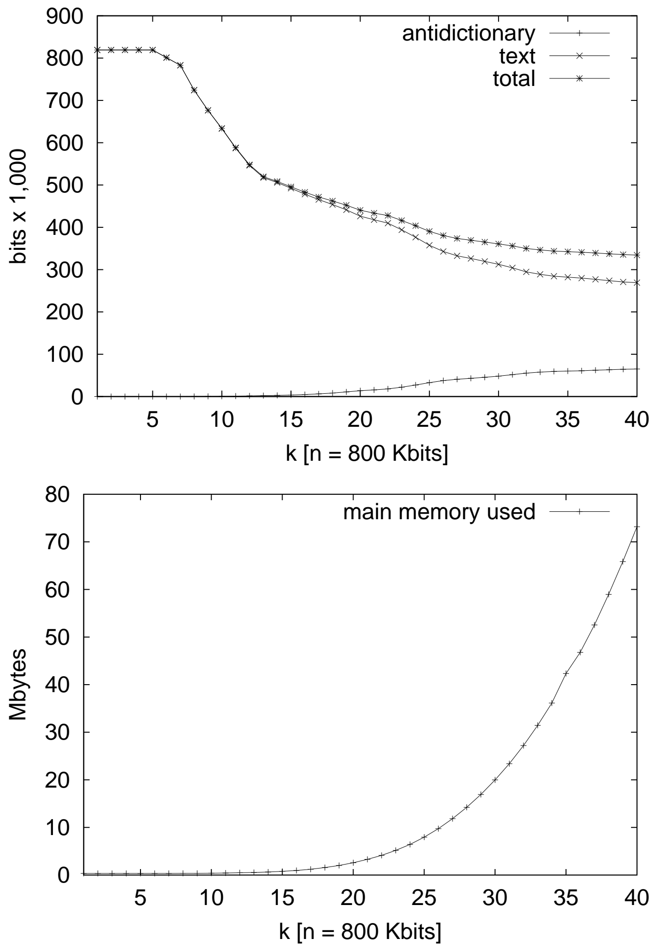

Figure 2. *Classical method:* **On top, the size of the different parts of the compressed file as k grows. On the bottom, amount of memory required.**

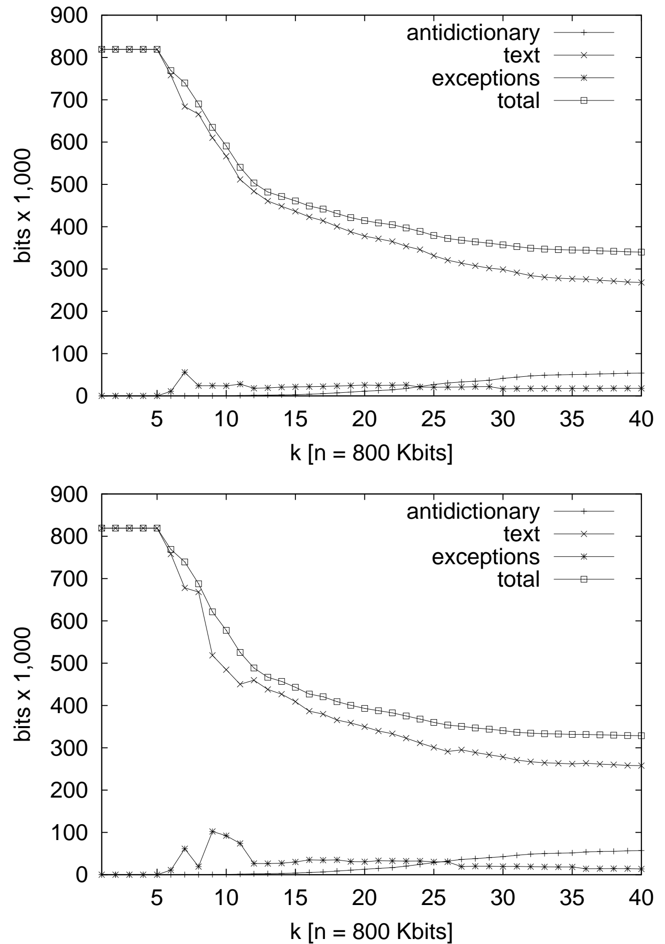

Figure 3. *Almost antifactors:* **The sizes of the antidictionary, the exceptions file, and the compressed text. On top the one-pass and on the bottom the multi-pass heuristics.**

The multi-pass heuristic is much more successful, although in many cases it needs as many as 20 passes over the trie (the number of passes is about $k/2 \dots 2k/3$). It obtains up to 12% improvement in the compression ratio and it is never worse than the classical method. The improvement, however, declines as k grows, being negligible for $k = 40$.

Another way to see this effect is to consider how much less memory we need to obtain the same compression of the classical algorithm. For example, with the multi-pass heuristic we obtain with $k = 32$ the same compression obtained with the classical algorithm using $k = 40$, which means that we can obtain the same result using 37% of the memory space.

We show this memory comparison in Figure 4 (bottom). Since we never obtain exactly the same compression ratio, we can speak about an optimistic and a pessimistic estimation of the savings in memory, knowing that the true value is somewhere in between. As it can be seen in the figure, for $k \geq 15$ we need 30%–55% of the memory of the classical algorithm in order to obtain the same compression performance.

This shows that almost antifactors are interesting when there is limited space for compression, since they permit obtaining the compression ratios that, using the classical algorithm, would demand much more main memory.

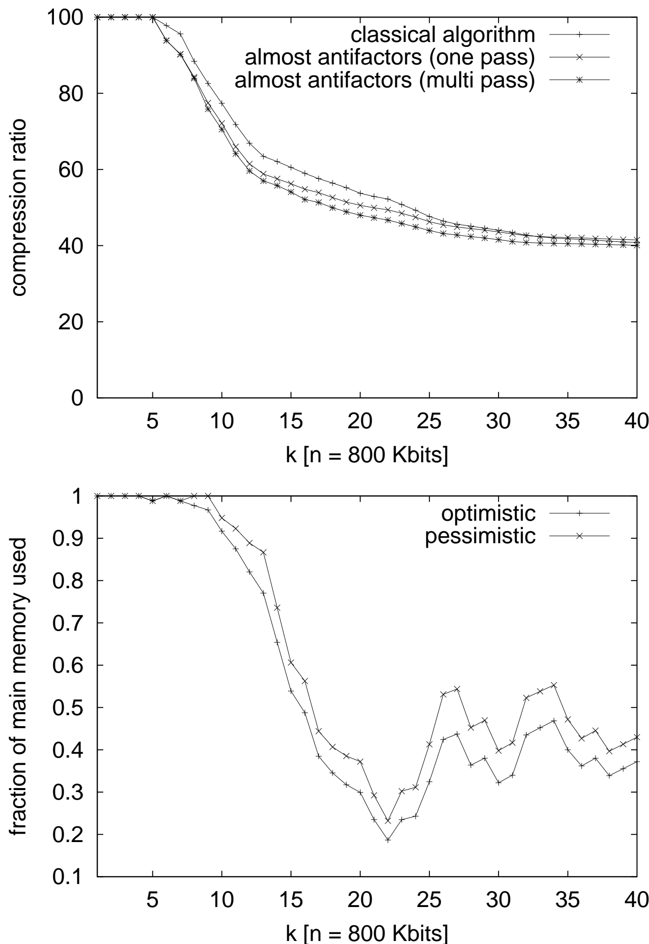

Figure 4. On top, the compression ratio obtained for the classical algorithm and for both almost antifactors heuristics. On the bottom, fraction of the memory needed by the multi-pass heuristic to obtain the same compression of the classical algorithm, when the latter uses the k values shown in the x axis.

6. Conclusions

We have presented an improvement on antidictionary based compression, by introducing a statistical technique that adds antifactors which in fact do appear in the text, but so few times that we improve compression by considering that they do not appear and coding the exceptions separately. We call these "almost antifactors". It is experimentally shown that these improve compression by up to 12%, or alternatively obtains the same compression using 30%-55% of the space. This mechanism is useful especially when there is a limited amount of space to compress, as it obtains results that would demand much more memory space on the classical algorithm.

There are several directions for future work: (1) Better ways to code the exceptions file can have an impact on the result. (2) We have assumed that terminal nodes have to be leaves, but this is not really necessary. More compression can be obtained by letting internal nodes be terminal. On the other hand, we have to devise a mechanism to represent an antidictionary where internal nodes can be terminal too. (3) We have proposed a heuristic to find good almost antifactors that performs satisfactorily but does not guarantee optimality. An interesting issue is whether an algorithm to obtain the optimal set of almost antifactors can be designed, or at least improve the current heuristics. (4) Finally, an interesting problem is whether the set of useful antifactors can be obtained without first building the set of all the antifactors, which is much larger, and seeing how can this be extended to almost antifactors. This may permit using much less memory space at compression time.

References

[1] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[2] A. Apostolico and Z. Galil. *Combinatorial Algorithms on Words*. Springer-Verlag, New York, 1985.

[3] T. Bell, J. Cleary, and I. Witten. *Text Compression*. Prentice Hall, New Jersey, 1990.

[4] M. Crochemore, F. Mignosi, A. Restivo, and S. Salemi. Text compression using antidictionaries. In J. Gruska, L. Brim, and J. Slatuška, editors, *Proc. ICALP'99*, LNCS 1664, 1999.

[5] M. Crochemore, F. Mignosi, A. Restivo, and S. Salemi. Data compression using antidictonaries. *Proceedings of the IEEE*, 88(11):1756–1768, 2000. Special issue *Lossless data compression* edited by J. Storer.

[6] D. Harman. Overview of the Third Text REtrieval Conference. In *Proc. Third Text REtrieval Conference (TREC-3)*, pages 1–19, 1995. NIST Special Publication 500-207.

[7] D. Knuth. *The Art of Computer Programming*, volume 3: Sorting and Searching. Addison-Wesley, 1973.