Musings about Text Redundancy and Text Compression

Stefan Gruner
Dept. of Comp. Science
Universiteit van Pretoria
stefan@cs.up.ac.za

Abstract
I muse about some "stringological" questions: Is it possible to encode and compress any given string in such a way that *all* redundancy is removed? And, if this not possible: How closely could we approximate the ideal aim? My little essay is *naive* in the sense that I have never studied "stringology" and coding theory properly – therefore also no literature references at the end of this paper. The sole purpose of this little sketch is to *entertain* my colleague *Derrick Kourie* at the occasion of his 60th birthday, for which I wish him all the best of happiness as well as many further years to come.

1. Preliminaries

Let us try to compress strings of text in a fully reversible way, such that the result *S'* = compress(*S*) can be undone to *S* = uncompress(*S'*) without loss of information and without any ambiguity.

We need a text alphabet *T*, which is here (for the sake of intuition) the small Roman alphabet {*a, b, ..., y, z*}.

To separate words and sentences from each other within a string, we need a blank space symbol ε = " " as well as a period symbol π = "." – these control tags however do not belong to the texts which we want to encode and compress, and they also occur in the code.

Finally we also need a code alphabet *C*, which is here (for the sake of intuition) the set of natural numbers ***N*** {1, 2, 3, ...} – though any other alphabet, even *T* itself could be chosen.

Definition. Let *S* be a text string consisting of a finite number of words (*w* ϵ *T**) which are separated from each other by occurrences of ε or π. The ***weight*** of a word *w* (in *S*) is its length |*w*| multiplied with the number #(*w*) of its occurrences (i.e. repetitions, or frequency) in *S*. The weights of the word separator symbols ε and π are defined as Null. ■

2. Naive Method

In the first phase of the procedure, we build a list of all words occurring in a given text, and we sort this list in a descending order by weight of the words. Searching and sorting all those words is certainly an expensive task from the perspective of complexity theory, however let us naively assume for now that we would get all such searching and sorting for free, as a birthday present from a friendly oracle demon :-)

Example.

S = "ich gehe spazieren. ich gehe in diese richtung."

The weights of the words in this text are calculated as follows: 2|ich|=6, 2|gehe|=8, 1|spazieren|=9, 1|in|=2, 1|diese|=5, 1|richtung|=8. Sorting the words from the highest weight to the lowest we get the following list:

1)	spazieren	// weight 9
2)	richtung	// weight 8
3)	gehe	// weight 8
4)	ich	// weight 6
5)	diese	// weight 5
6)	in	// weight 2 ■

At this point of the elaboration we can see why the set ***N*** was chosen as our code alphabet in the introductory section: The code "word" which we choose to encode a text word is simply the *index* number of this word in the weighted list. Thus, "spazieren" will be replaced by "**1**", "richtung" will be encoded by "**2**", and so on, yielding *S'* = "**4 3 1**. **4 3 6 5 2**."

Obviously it is necessary to keep the coding table as a separate data structure in addition to the encoded text string *S'*, otherwise no decoding would be possible any more. The length of the coding table itself –in addition to the length of the compressed string *S'*– puts a heavy tax-burden onto our total compression gain; we will come back to this problem in one of the following paragraphs.

In a second phase of the procedure we can try to compress the coding table itself! The method for this is the same as the method for the compression of the original text. In our example of above we detect –with help of our friendly search demon– that the little word "ich" (index 4) occurs as a sub string within the longer word "richtung" (index 2). Consequently we can replace the initial coding table with the following:

1) spazieren
2) r**4**tung
3) gehe
4) ich
5) diese
6) in

In a further phase of the recursion our friendly search demon could detect that the compressed string *S'* (see above) contains *two* occurrences of the code phrase "**4 3**": this is redundancy which must vanish! Finally our friendly birthday demon would also detect that the tiny string "ie" occurs in the word "diese" as well as in the word "spazieren". Consequently we can construct yet another –better– coding table, namely:

1) spaz**8**ren
2) r**4**tung
3) gehe
4) ich
5) d**8**se
6) in
7) **4_3** // auxiliary binding-symbol "_"
8) ie

together with the encoded string *S''* = "**7** 1. **7** 6 5 2."

Here we have reached the *fixpoint* of the iteration: no further compressions are possible (based on *T* and ***N***), and all redundancies are eliminated.

The resulting *file F* **[...]**, which can obviously be un-compressed again without any ambiguity or loss of information, has then the following contents:

[7 1. 7 6 5 2 | spaz8ren r4tung gehe ich d8se in 4_3 ie**]**

Note that the index numbers <1, 2, ..., 8> of the code list do not need to be written explicitly into the file as they are represented *implicitly* by the order of the un-coding words following *S''* after the separation bar |. Further note that the auxiliary binding-symbol "_" is necessary such that the code word "4_3" can be identified as *one* item, namely the 7th item in the code list, after the list *index* numbers were removed from the result file.

Question: *What have we won?* Counting only those characters belonging to the text and code alphabets *T* and *C* (but *not* the auxiliary symbols "ε", "_", ".", "|") we find:

|*S*| = |"ichgehespazierenichgeheindieserichtung"| = **38**.

|*F*| = |"717652spaz8renr4tunggeheichd8sein43ie"|=**37**.

In this tiny example, the compression gain on the original text *S* is almost completely absorbed by the tax which must be paid in form of the coding table sitting behind *S''* and the separation bar "|" in the file.

For our example we calculate our total compression gain as: *G* = (1 – 37/38) ≈ **0.026** – only 2.6% percent, though the *G* would have been bigger for longer texts which are sufficiently large in comparison to the size of the coding table.

3. Searching Phrases instead of Words

Reflecting our little example critically, we must ask the question: *Could we have won more than that?* Indeed, the answer is: "yes"! Remember how late in the process our helpful search demon has detected the double occurrence of the sub string "4_3" in the intermediate string *S'*. Code "4_3", however, represented the sub-string "ich_gehe", which indeed occurs twice in the original text *S*. The sub-string "ich_gehe", however, consisting of 3 + 4 = 7 characters from *T*, has a weight of 2 · 7 = 14 in *S*! This means that we had too naively constructed our initial table by looking only for repeated occurrences of *single* words – instead of looking for repetitions of longer *phrases*. Had our friendly search demon found this longer phrase in the first place, then our coding table would have been much more concise: Lines (3) and (4) *together* would have been at the first position (due to the highest weight, 14, in *S*), and line (7) would not have been needed at all – thus only five lines are needed instead of our initial six, namely:

1)	*ich_gehe*	// weight 14
2)	*spazieren*	// weight 9
3)	*richtung*	// weight 8
4)	*diese*	// weight 5
5)	*in*	// weight 2

Again we apply the coding procedure recursively on the table itself, thereby reducing the redundancies of each two occurrences of sub-strings "ich" and "ie". The resulting coding table has the form:

1) **6**_gehe
2) spaz**7**ren
3) r**6**tung
4) d**7**se
5) in
6) ich
7) ie

The sub-string "ich_gehe" will now get code word "1", "spazieren" will get code word "2", and so on, and "ie" will finally get code word "7". Consequently the resulting file *F* contains:

[1 2. 1 5 4 3 | 6_gehe spaz7ren r6tung d7se in ich ie**]**

Needless to say that the total weight of the original text is still |*S*| = 38, but the total weight of the result *F* (without counting auxiliary symbols which are neither in alphabet *T* nor in alphabet *C*) is now only |*F*| = **36**. In comparison with the previous arrangement (where |*F*| = 37) we have saved 1 weight-unit, and the total compression gain is now *G* = (1 – 36/38) ≈ **0.055**, or 5.5% percent. This is a nice difference in comparison to our previous 2.6% percent gain, and it shows the worthiness of searching for longer phrases (and not only the individual words) in the original text string *S*. However, the computational complexity of the task *find all repetitions of any multi-word-phrase* (which our friendly birthday-demon is supposed carry to out) is certainly larger than the computational complexity of the rather simple task of finding all repetitions of single words.

4. Compression Conflicts

As soon as we have issued the recommendation to search for reoccurrences of multi-word-phrases (instead of reoccurrences of single words only) the problem arises that two or more of such long compressible phrases can *overlap*. This leads to what I would call a *compression conflict*.

Example. Let us look at the following sentences (in German language, again, just for the fun of it):

a) "ich habe doch (nie gewonnen)."
b) "ich habe aber (schon oft gespielt)."
c) "er habe aber (bereits gewonnen, sagte er)."

(We ignore the endings of these sentences in the brackets as they are not important to the problem which we now want to address.) We notice that there are two occurrences of the phrase "ich habe", namely in sentences a) and b), as well as two occurrences of the phrase "habe aber", namely in sentences b) and c). Moreover we notice that these phrases are *not* disjoint in sentence b) – unfortunately they overlap.

Definition. In the context of the example of above, the set **{** ich habe aber **}** is the *conflict **set***. Generally speaking, the conflict set is the union-set of all words which participate in the same conflict. Moreover, in the example of above, the set **{** habe **}** is the *conflict **case***. Generally, the conflict case is the intersection of the overlapping phrases which participate in the same conflict. Finally, in our example, the set **{** ich aber **}** is the *conflict **context***. Generally speaking, the conflict context is the difference-set between the conflict set and the conflict case. ■

How shall we deal with such conflicts when our goal is *optimal* compression? It does not seem very daring to conjecture that a *greedy* compression algorithm, which would always choose the first alternative in the case of a conflict, would generally not lead to the best solution. In the case of a coding conflict, the optimal compression algorithm should either:

- predict (oracle) the optimal choice, or:
- branch out in two sub-processes, compute both alternatives, compare the results, output the best and discard the worse.

Needless to assert that, whereas the second option might "only" be extremely expensive, the first option might possibly not be computable at all. To illustrate the scenario, let us build the two alternative coding tables for the example of above, (not taking the parts between brackets into account). A first alternative is:

1)	ich_habe	// weight **14**
2)	aber	// weight 8
3)	doch	// weight 4
4)	habe	// weight 4
5)	er	// weight 2

Proceeding as explained above we can further encode the compression table itself, yielding:

1) i**6_4**
2) **7_5**
3) do**6**
4) h**7**e
5) er
6) ch
7) ab

Remarks. We wrote "7_5" in the second line (and not "75") in order not to get confused with code word "75" which also exists in the code alphabet, though it is not applied in this example; ditto for "6_4" (instead of "64" in the first line). More important: There was another coding conflict *within* the table, namely with the words "aber" (line 2) and "habe" (line 4), which nicely demonstrates the ubiquitous character of this phenomenon. In the micro-conflict within the table, the conflict case (in the sense of the definition above) is **{** abe **}**, the conflict context is **{** h r **}** and the whole conflict set is **{** h abe r **}**. In the 7-line table of above I have "just somehow" done the micro-coding without any deeper considerations about the micro-conflict on the level of single characters within those words. The string *S* ="ich habe doch. ich habe aber. er habe aber" will thus be compressed to "1 3. 1 2. 5 4 2" and the result file *F* (with all decompression information) is **[**1 3. 1 2. 5 4 2 | i6_4 7_5 do6 h7e er ch ab**]**, whereby |*F*| = **24**.

For comparison let us now try the second alternative of our example conflict set **{** ich habe aber **}** (whereby the *micro*-conflict **{** h abe r **}** *within* the table must of course be treated in the same way as above; otherwise our comparison would be methodologically invalid). For the second alternative we start with this initial table:

1)	habe_aber	// weight **16**
2)	ich	// weight 6
3)	doch	// weight 4
4)	habe	// weight 4
5)	er	// weight 2

The words "doch", "habe" and "er" are placed at the same positions (3, 4, 5) as before, yielding now:

1) **4_7_5**
2) *i***6**
3) do**6**
4) h**7**e
5) er
6) ch
7) ab

The two tables to be compared are placed exactly opposite of each other in the two columns of this page such that direct comparison is as easy as possible by simply looking left and right :-)

String *S* ="ich habe doch. ich habe aber. er habe aber" will now be compressed to "2 4 3. 2 1. 5 1", and the result file *F* (with all decompression information) is **[**2 4 3. 2 1. 5 1 | 4_7_5 i6 do6 h7e er ch ab**]**, whereby |*F*| = **24**, as in the first alternative!

Here we have thus found a nice little example where it does not matter whether we solve the compression conflict in this way or in the other way. However:

- Without having done this comparison step by step we would have hardly been able to predict (foresee, oracle) this result.
- From this special little example we may of course *not* conclude that the two (or more) alternative solutions to a compression conflict would *always* lead to results of exactly the same quality as far as the total weight of the output file *F* is concerned.

Anyway we have learned that *optimal* compression in all its recursive branches and choice-alternatives with all the searching of all the reoccurring phrases both at macro- and at micro-level is not at all an easy task. A text is optimally compressed when *redundancy-free*.

5. Implications of Code Word Lengths

Text compression, as shown above, would not make any sense if the code-words would be longer than the words to be encoded. The choice of the code alphabet *C* thus imposes a theoretical upper-bound on the total compression gain which can be achieved for texts of the original alphabet *T*. For example, if C = ***N***, the set of natural numbers, then it would not make sense to encode the little word "ich" with code-word "7126", because |7126| = 4 > 3 = |ich|; the compression gain would be negative – a loss rather than a gain. Thus we can state that compression reaches its limits when the source language consists of huge numbers of tiny words for which no target language with even more and even tinier words can be found.

Yet there is room for further tinkering and tampering! **Example.** Imagine that for whatever string *S*, the following word (or phrase) weight table had been built:

1) ...
2) ...
3) ...
4) ...
5) ...
6) ...
7) ...
8) ...
9) you
10) me
11) ...

For some reason, the word "you" had an altogether higher weight in text S than the word "me", therefore, according to our weight-based algorithm, "you" was put at a lower position than "me" in this index table, which means that "You" would have to be encoded by "9" whereas "I" would have to be encoded by "10" though |me| = |10| = 2 (lexically speaking) – no gain! In cases it might be wise to swap positions of words in the index table before continuing the procedure of compression. However, such word position swapping should be applied wisely rather than blindly; the total win must be calculated in terms of the *weights* of the words involved. In continuation of the example let us consider the following two example cases:

a) Both "you" and "me" occur exactly once in *S*, thus their weights are 1|you|=**3** and 1|me|=**2**, we see that 3>2, therefore "you" has a lower list position (9) than "me" (with list position 10).
b) Whereas "me" occurs now 4 times in *S*, "you" will occur now thrice, such that the weight of "you" in *S* is now 3|you| = **9**, whereas 4|me| = **8**.

We calculate the total gains of swapping "you" and "me" in the index list.

a) In the first case, the total weight of "you" and "me" together –in the uncompressed text– is 3 + 2 = 5.
   i. **Swapping Index Positions.** "you" is encoded by "10" and "me" by "9". |10|=2, |9|=1, both occur each once, thus the compression result has the weight 2 + 1 = **3**.
   ii. **Keeping Positions as are.** "you" is encoded by "9" and "me" by "10". |9|=1, |10|=2, both occur each once, thus the compression result has the weight 1 + 2 = **3**.
b) In the second case, the total weight of "you" and "me" together in the uncompressed text is 9 + 8 = 17.
   i. **Swapping Index Positions.** "you" is encoded by "10", "me" by "9". With their different occurrence frequency we have 3|10|=6, 4|9|=4, and the result of the compression has the weight 6 + 4 = **10**.
   ii. **Keeping Positions as are.** "you" is encoded by "9", "me" by "10". With their different occurrence frequency we have 3|9|=3, 4|10|=8, and the result of the compression has the weight 3 + 8 = **11**.

Thus, in case a) it does not matter whether or not we swap the positions of "you" and "me" in the coding table; in either way the total compression gain is *G* = (1 – 3/5) = 0.4 (i.e. 40% percent). In case b), however we gain *G* = (1 – 10/17) ≈ **0.41** *with* position swapping, but only *G* = (1 – 11/17) ≈ **0.35** without. The result should not be too surprising, as the whole compression system is based on the principle of replacing the heaviest text phrase by the lightest code word.

6. Outlook

Already 40 years ago (in the same year in which the term *Software Engineering* was coined at the now much commemorated NATO Science Conference of Garmisch, Germany, 1968, when *Derrick* was just 20 years of age), a book was published by *Harold Borko* in which the possibilities of *automated text-abstraction* and *automated word-index generation* were discussed.[*] The dream was to input a text in electronic form into some computing machinery, and out would come a little abstract-summary of the text. Ditto for the generation of index-appendices: input would be a book in electronic form, and out would come a list of the most important keywords or phrases (with page numbers attached) which could help the reader-in-hurry to look up certain pages and navigate through the book. I believe that my naive little word-weight-definition (on the first page of this paper) can serve as a good starting-point to such an endeavour, because the "importance" of a word or phrase in a text S is likely related to its numeric frequency. Additionally needed, of course, would be a filter-list of frequently occurring structure-words (like "the", "and", "but", etc.) which do not carry much meaning in themselves but rather perform a grammatical role in the structuring of meaningful sentences. The number of those grammatical structure-words in the lexicon of a language is usually very small compared to the size of the entire lexicon such that an according filter-table should be easy to implement.

Moreover, the word-weight-method might perhaps also be useful for text comparison (such as plagiarism detection, etc.) whereby the –let's say– hundred first index positions of the weighted-phrase lists of the two texts could be analysed for differences or similarities.

As mentioned above in the abstract, the sole purpose of my *musings* was the *entertainment* of their readers at the occasion of Derrick's 60th birthday. Neither did I look into the up-to-date literature on coding and file compression theory, nor do I claim any originality for my simplistic considerations. In fact I am quite sure that much more sophisticated solutions must be long known to all the problems which I have sketched in this paper.

[*] Harold Borko (Ed.): ***Automated Language Processing***. John Wiley, New York, **1968**. US Library of Congress No. 66-26735.