On the Usefulness of Backspace

Shmuel T. Klein[1] and Dana Shapira[2]

[1] Department of Computer Science
Bar Ilan University
Ramat Gan, Israel
tomi@cs.biu.ac.il
[2] Department of Computer Science
Ashkelon Acad. College
Ashkelon, Israel
shapird@ash-college.ac.il

Abstract. The usefulness of a backspace character in various applications of Information Retrieval Systems is investigated. While not being a character in the initial sense of the word, a backspace can be defined as being a part of an extended alphabet, thereby enabling the enhancement of various algorithms related to the processing of queries in Information retrieval. We bring examples of three different application areas.

1 Introduction

A large textual database can be made accessible by means of an Information Retrieval System (IRS), a set of procedures which process the given text to find the most relevant passages to a specific information request. This request is usually formulated according to some given rigid query syntax, but in fact the formulation of a query is an art in which the user has to find the right balance between a choice of query terms that may be too broad and others that could be too restrictive.

The present work is an extension of an earlier study of the *negation operator* as it appears in its various forms in Information Retrieval applications [7]. We restrict attention to the Boolean query model, as in Chang et al. [1], though several alternatives are available, like the classical vector space model [9], the probabilistic model [11], and others. The natural approach of most users to query formulation involves the choice of keywords that best describe their information needs. They often overlook the possibility, which sometimes could even be a necessity, of choosing also a *negative* set, that is, a set of keywords which should *not* appear in the vicinity of some others, thereby achieving improved precision. But the use of negation might in certain cases be tricky and is not always symmetrical to the use of positive terms.

We now turn to the usefulness of an element which is intrinsically negative, namely a *backspace* character. A backspace is not really a character in the classical sense, as it is not explicitly written or used to form any words, but keyboards contain a backspace key and standard codes like ASCII assign it a codeword, so programmers, rather than users, consider backspaces just as any other printable character. The purpose of this paper is to show that in spite of it not representing any concrete entity, a backspace can be a useful tool in various different applications related to the implementation of IR systems. The intention is not to present a comprehensive investigation of all the possible applications of backspaces, but rather to emphasize its usefulness by means of some specific examples in different areas of Information Retrieval. The next section deals with the processing of large numbers in an IRS and suggests a solution based on using backspaces as part of the elements that might be retrieved. In Section 3 we

Shmuel T. Klein, Dana Shapira: On the Usefulness of Backspace, pp. 80–89.
Proceedings of PSC 2009, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04403-2

consider compression aspects of the textual databases within an IRS, and show how a model including a backspace may lead to improved savings. As a last example, we show in Section 4 how the use of backspaces may lead to another time/space tradeoff in an application to the fast decoding of Huffman encoded texts.

2 Dealing with large numbers

2.1 Syntax definition

To enable the subsequent discussion, one has first to define a query language syntax. Most search engines allow simple queries, consisting just of a set of keywords, such as

$$A_1 \quad A_2 \ \ldots \ A_m, \tag{1}$$

which should retrieve all the documents in the underlying textual database in which all the terms A_i occur at least once. Often, some kind of stemming is automatically performed on all the terms of the text during the construction of the database, as well as online on the query terms [5]. *Negating* one or more keywords in the query means that one is interested in prohibiting the occurrence of the negated terms in the retrieved documents. A further extension of the query syntax accommodates also tools for *proximity* searches. The idea is that a user may wish to limit the location of possible occurrences of the query terms to be, if not adjacent, then at least quite close to each other. Many query languages support, in addition to the loose formulations of (1), also an *exact phrase* option. This should, however, be used with care, as one has to guess all possible occurrence patterns of the query terms, and failing to do so may yield reduced recall.

This leads to the following generalization. Consider a query containing only positive terms as consisting of m keywords and $m-1$ binary distance constraints, as in

$$A_1 \ \ (l_1 : u_1) \ \ A_2 \ \ (l_2 : u_2) \ \ \ldots \ \ A_{m-1} \ \ (l_{m-1} : u_{m-1}) \ \ A_m. \tag{2}$$

This is a conjunctive query, requiring all the keywords A_i to occur within the given metrical constraints specified by l_i, u_i, which are integers satisfying $-\infty < l_i \leq u_i < \infty$ for $1 \leq i < m$, with the couple $(l_i : u_i)$ imposing a lower and upper limit on the distance from an occurrence of A_i to one of A_{i+1}. The distance is measured in words, and usually restricts, in addition to the specific constraints imposed by the $(l_i : u_i)$ pairs, all the terms to appear within some predefined textual unit, like the same sentence. Negative distance means that A_i may follow A_{i+1} rather than precede it. In the presence of negated keywords, association of keywords to metrical constraints should be to the left, unless there is no such option, that is, all the keywords to the left of the leftmost non-negated one will be right associated (each query must have at least one non-negated keyword). An example of left association could be A (1:3) −B (1:5) C, meaning that we seek an occurrence of C following an occurrence of A by 1 to 5 words, but such that there is no occurrence of B in the range of 1 to 3 words after A. In the query −D (1:1) E, an occurrence of E should not be preceded immediately by an occurrence of D.

Such a query language is used for over thirty years at the *Responsa Retrieval Project* [4,2]. Even more extended features, mixing Boolean operators with proximity constraints between certain keywords can be found in the *word pattern* models for Boolean Information Retrieval $\mathcal{WP}$ and $\mathcal{AWP}$ [12].

2.2 The use of backspace for large numbers

The problem with large numbers in an IRS is that there are potentially too many of them. If we store 20,000 pages of a running text, including also the page numbers, at least all the numbers up to 20,000 will appear, which can be an increasingly large part of the dictionary. A real-life application of this problem is mentioned in [8]. A possible solution is to break long strings of digits into blocks of at most k digits each. For $k = 4$, the number `1234567` would thus be stored as 2 consecutive items: `1234` and `567`. We have thus bounded the number of possible numbers by 10^k, independently of the length of the text.

There is however a new problem, namely one of precision. If the query asks for the location of a number such as `5678`, the system would also retrieve certain occurrences of numbers having `5678` as substring, like `123456789`. We therefore need some indicator, telling if the string is surrounded by blanks or not. The two obvious solutions, of storing indices of all the blanks, or using special treatment on all queries involving numbers, have to be ruled out, the first because of the increased space requirements, the second on the basis of the additional processing time that would be required to check the vicinity of each occurrence of a number.

A possible solution to the problem can be based on the fact that we are not restricted to deal only with standard tokens such as characters or character strings. In fact, we need some mechanism to overcome our implicit assumption that all the words in the text, and therefore also in the query, are separated by blanks. Let us thus formalize the setup.

We assume that the words stored in the dictionary are full words as they appear in the text, *each followed by a blank*. For example, we might find there the words `house`␣, the␣, etc, where the ␣ is used to visualize the terminating blank. Of course, these blanks are not actually stored, but they are conceptually present. A query asking for `House of Lords` will thus generate 3 accesses to the dictionary, with items `House`␣, `of`␣ and `Lords`␣, and one would check if there is an occurrence of these three items in the same sentence, having consecutive relative indices.

A similar treatment will also be given to numbers of up to k digits in length. For a longer number, the fact that there are no spaces within it will be stored by means of a *back-space* item, BS. For example, the number `1234567890` will be stored as a sequence of the following items: `1234`␣, BS, `5678`␣, BS, `90`␣. The backspace will be treated like any other word: it will have a consecutive numbering, and all its occurrences will be referenced in the concordance. For example, consider the phrase:

`I declared an income of 1000000 on my last 10 1040 forms.`

When inverting the text to build the dictionary and the concordance, this will be parsed as `I declared an income of 1000 BS 000 on my last 10 1040 forms`, with the words numbered 1 to 14, respectively. In particular, the BS has index 7. Note the absence of a backspace between 10 and 1040, since these are indeed two separate numbers, and the space between them is a part of the original text.

Punctuation signs are traditionally attached to the preceding word, which should therefore loose its trailing blank. This can be implemented by considering each punctuation sign as if it were preceded by BS. The phrase `Mr. Jones` would thus be encoded by `Mr`␣ `BS . Jones`.

It is true that having the backspaces numbered just like words may disrupt proximity searches which do not require adjacency. In the query `solve` $(-10 : 10)$

`differential`, the request is to find the query terms at a distance from up to 10 words from each other, without caring which term precedes the other. The occurrence of a backspace between the occurrences of these terms in the text may lead to the wrong numbering, and therefore cause retrieval of passages which would not have complied with the strict original definition of the distances, or it may imply the non-retrieval of other passages which should have been retrieved. However, the same is true already if large numbers are split, even when no backspaces are used. The current proposition is thus only valid if either:

- no proximity searches other than immediate adjacency are supported;
- or the software is adapted to deal with the correct numbering also in the presence of backspaces and number splits;
- or that strict adherence to the exact metrical constraints is not deemed critical. In most queries using large distances, one can hardly justify the use of $(-10 : 10)$ rather than $(-11 : 11)$ or $(-9 : 9)$, so even if backspaces or number splits effectively reduce the range of the query, this does not necessarily lead to a worse recall/precision tradeoff. In other words, the fact that a text passage does not strictly obey to the constraints imposed by the query does not yet mean that the passage is absolutely non-relevant.

Table 1 brings a few examples of queries including large numbers, and how they can be processed by means of the backspace item.

Searching for	Submit query	Comments
`234`	−BS `234`	the negated backspace to avoid retrieval of, e,g, `8888234`
`2000 1040`	−BS `2000 1040` −BS	
`12345678`	−BS `1234` BS `5678` −BS	
`1234567`	−BS `1234` BS `567`	Note that since the last part of the number has less than $k = 4$ digits, it is not necessary to add the −BS, which was used in the previous example to prevent the retrieval of `123456789` for example
`user@address.com`	`user @` BS `address` . BS `com`	Note the absence of BS preceding `@` and the dot, since these are punctuation signs

Table 1. Examples of the use of a backspace character

3 Compressing the text of an IR System

At the heart of any Information Retrieval system is the raw text, which is usually stored in some compressed form. A myriad of different text compression schemes has been suggested, but a full description is beyond the scope of this work, and the reader is referred to [14] for a description of many of these methods.

One class of compression techniques, often called *statistical*, uses variable length codewords to encode the different characters. Compression is achieved by assigning

shorter codewords to characters occurring at higher frequency, and an optimal assignment of codewords lengths, once the character frequencies are known, is given by Huffman's algorithm [6].

But a Huffman code, applied on individual characters, does not achieve a good compression ratio, because adjacent characters are encoded as if they were independent, which is not the case for natural language texts. In order to exploit also inter-character correlations, one may extend the set of elements to be encoded, to include character pairs, triplets, etc., or even entire words. In the latter case, the Huffman tree could be huge, with a leaf for each of the different words in the text, but this overhead may be acceptable as the list of different words, also known as the *dictionary* of the Information Retrieval system, is needed anyway. The compression obtained by the word oriented Huffman code is excellent and competes with that of the best methods.

A text consists, however, not just of a sequence of juxtaposed words, but these words are separated by blanks and other punctuation signs, which have also to be encoded. One possibility is to use two, rather than a single Huffman code [8], one for the words, and another for the *non-words* separating them, keeping strict alternation to avoid having to encode an indicator of which code is being used. This method is referred to as the *Huffword* scheme [14].

But the overwhelming majority of the non-words are blanks, so their encoding would be wasteful. As alternative, one may use the idea of the backspace as above. A single Huffman code can be used, for which the set of elements to be encoded consists of:

1. the words including a trailing blank. This is the same idea as in the definition of the dictionary in the IR application of Section 1, in which the blank following a word is considered an integral part of the word itself;
2. punctuation signs, also including trailing blanks, but being preceded by a conceptual backspace to attach them to the word they follow;
3. the backspace character, to deal also with the exceptions of the attachment rules for punctuation signs.

Every text can be parsed into a sequence of such elements, and a *single* Huffman code can be built on the basis of the frequencies obtained from this parsing.

We tested the approach on two textual databases of different sizes and languages: the Bible (King James version) in English, and the French version of the European Union's JOC corpus, a collection of pairs of questions and answers on various topics used in the ARCADE evaluation project [13]. Table 2 reports on the basic statistics and on the compression results.

File	Size	Huffword	BShuff	# exceptions	gzip	bzip
English	3.1 MB	3.91	3.97	2006	3.28	4.41
French	7.1 MB	3.98	4.03	24430	3.27	4.63

Table 2. Comparing backspace based compression

The values give the compression ratio, which is the size of the original file divided by the size of the compressed file. We see that in both cases, the approach using a backspace (BShuff) is slightly better than what can be obtained by Huffword. The table also contains data for compression by gzip (with parameter -9 for maximal

compression) and bzip2. Both Huffword and BShuff are preferable to gzip, but bzip is considerably better. It should however be noted that the comparison to the adaptive Lempel Ziv based gzip or Burrows-Wheeler based bzip is not necessarily meaningful: adaptive methods require the whole file to be decompressed sequentially and do not allow partial decoding of selected sub-parts, as can be obtained by the static Huffman based methods. In certain applications, the zip methods are thus not plausible alternatives to those treated here, even if their compression is better.

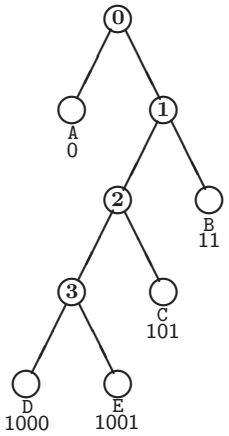
4 Blockwise decoding of Huffman codes

We now turn to a last example, also connected to the compression of large texts in an IRS, but concentrating on the decoding. Indeed, decoding might be of higher importance than encoding, since the latter is only done once, when the system is built, while efficient decoding is critical for getting a good response time each time a query is being submitted. However, the decoding of variable length codes, and in particular Huffman codes, can be slow, because the end of each codeword has to be detected by the decoding algorithm itself, and the implied manipulations of the encoded text at the bit level can have a negative impact on the decoding speed. But efficient decoding of k bits in every iteration, for $k > 1$, rather than only a single one, is made possible by using a set of m auxiliary tables, which are prepared in advance for every given prefix code. The method has first been mentioned in [3], and has since been reinvented several times, for example in [10].

4.1 Basic decoding scheme

The basic scheme is as follows. The number of entries in each table is 2^k, corresponding to the 2^k possible values of the k-bit patterns. Each entry is of the form (W, j), where W is a sequence of characters and j $(0 \leq j < m)$ is the index of the next table to be used. The idea is that entry i, $0 \leq i < 2^k$, of table number 0 contains, first, the longest possible decoded sequence W of characters from the k-bit block representing the integer i (W may be empty when there are codewords of more than k bits); usually some of the last bits of the block will not be decipherable, being the prefix P of more than one codeword; j will then be the index of the table corresponding to that prefix (if $P = \Lambda$, where Λ denotes the empty string, then $j = 0$). Table number j is constructed in a similar way except for the fact that entry i will contain the analysis of the bit pattern formed by the prefixing of P to the binary representation of i. We thus need a table for every possible proper prefix of the given codewords; the number of these prefixes is obviously equal to the number of internal nodes of the appropriate Huffman-tree (the root corresponding to the empty string and the leaves corresponding to the codewords), so that $m = N - 1$, where N is the size of the alphabet.

More formally, let P_j, $0 \leq j < N-1$, be an enumeration of all the proper prefixes of the codewords (no special relationship needs to exist between j and P_j, except for the fact that $P_0 = \Lambda$). In table j corresponding to P_j, the i-th entry, $T(j, i)$, is defined as follows: let B be the bit-string composed of the juxtaposition of P_j to the left of the k-bit binary representation of i. Let W be the (possibly empty) longest sequence of characters that can be decoded from B, and P_ℓ the remaining undecipherable bits of B; then $T(j, i) = (W, \ell)$.



Entry	Pattern for Table 0	Table 0		Table 1		Table 2		Table 3	
		W	ℓ	W	ℓ	W	ℓ	W	ℓ
0	000	AAA	0	D	0	DA	0	DAA	0
1	001	AA	1	E	0	D	1	DA	1
2	010	A	2	CA	0	EA	0	D	2
3	011	AB	0	C	1	E	1	DB	0
4	100	—	3	BAA	0	CAA	0	EAA	0
5	101	C	0	BA	1	CA	1	EA	1
6	110	BA	0	B	2	C	2	E	2
7	111	B	1	BB	0	CB	0	EB	0

Figure 1. Huffman tree and partial decoding tables

As an example, consider the alphabet {A,B,C,D,E}, with codewords {0, 11, 101, 1000, 1001} respectively, and choose $k = 3$. There are 4 possible proper prefixes: Λ,1,10,100, hence 4 corresponding tables indexed 0,1,2,3 respectively, and these are given in Figure 1, along with the corresponding Huffman tree that has its internal nodes numbered accordingly. The column headed 'Pattern' contains for every entry the binary string which is decoded in Table 0; the binary strings which are decoded by Tables 1, 2 and 3 are obtained by prefixing '1', '10' or '100', respectively, to the strings in 'Pattern'. If the encoded text, which serves as input string to this decoding routine, consists of 100 101 110 000 101, we access sequentially Table 0 at entry 4, Table 3 at entry 5, Table 1 at entry 6, Table 2 at entry 0 and Table 0 at entry 5, yielding the output strings EA B DA C.

The general decoding routine is thus extremely simple. Let $M[f;t]$ denote the substring of the encoded string serving as input stream to the decoding, that starts at bit number f and extends up to and including bit number t; let j be the index of the currently used table and $T(j,\ell)$ the ℓ-th entry of table j:

$j \leftarrow 0$
for $f \leftarrow 1$ to length of input do
$(\mathsf{output}, j) \leftarrow T(j, M[f; f+k-1])$
$f \leftarrow f + k$

The larger is k, the greater is the number of characters that can be decoded in a single iteration, thus transferring a substantial part of the decoding time to the preprocessing stage. The size of the tables, however, is $\Omega(N2^k)$, so it grows exponentially with k, and may become prohibitive for large alphabets and even moderately large k. For example, if $N = 30000$, k is chosen as 16 and every table entry requires 6 bytes, the tables, which should be stored in RAM, would need about 11 GB! Even if such amounts of memory were available, the number of cache misses and page faults would void a significant part of the benefits in time savings incurred by the reduced number of processing steps.

4.2 Using backspaces to get another time/space tradeoff

The number of tables, and thus the storage requirements, can be reduced, by conceptually modifying the text with the introduction of backspace characters at certain locations. Particularly in the case of a large alphabet, the blocksize k could be chosen smaller than the longest codeword, and tables would be constructed not for all the internal nodes, but only for those on levels that are multiples of k, that is for the root (level 0), and all the internal nodes on levels k, $2k$, $3k$, etc. There is an obvious gain in the number of tables, which comes at the price of a slower decoding pace: as before, the table entries consist first of the decoding W of a bit string B obtained by concatenating some prefix to the binary representation of the entry index. If B is not completely decipherable, the remainder P_j is used in the previous setting as index to the next table. For the new variant, if $|P_j|$, the length of the remainder, is smaller than k, then no corresponding table has been stored, so these $|P_j|$ bits have to be reread in the next iteration. This is enforced by adding, after the remainder, $|P_j|$ backspaces into the text.

Entry	Pattern for Table 0	Table 0 W	ℓ	b	Table 3 W	ℓ	b
0	000	AAA	0	0	DAA	0	0
1	001	AA	0	1	DA	0	1
2	010	A	0	2	D	0	2
3	011	AB	0	0	DB	0	0
4	100	—	3	0	EAA	0	0
5	101	C	0	0	EA	0	1
6	110	BA	0	0	E	0	2
7	111	B	0	1	EB	0	0

Figure 2. Reduced partial decoding tables

It should however be noted, that the modification of the text is only conceptual, and will manifest itself only in the modified decoding routine; the encoded text itself need not to be touched, so there is no loss in compression efficiency, only in decoding speed. The table entries for the new algorithm are thus extended to include a third component: a back skip b, indicating how many backspaces should have been introduced at that particular point, which is equivalent to the number of bits the pointer into the input string should be moved back. Using the above notations, $T(j,i)$ will consist of the triplet (W,ℓ,b), and the decoding routine is given by

$j \leftarrow 0 \qquad back \leftarrow 0$
for $f \leftarrow 1$ to length of input do
$(\mathsf{output}, j, back) \leftarrow T(j, M[f; f+k-1])$
$f \leftarrow f + k - back$

As example, consider the same Huffman tree and the same input string as above with $k = 3$. Only two tables remain, Table 0 and Table 3, given in Figure 2. Decoding is now performed by six table accesses rather than only 5 with the original tables,

using the sequence of blocks 100, 101, BS11, BS00, 001, BS01 to access tables 0, 3, 0, 0, 3, 0, respectively, where the backspace BS indicates that the preceding bit is read again, resulting in an overlap of the currently decoded block with the preceding one.

Figure 3 shows the input string and above it its parsing into codewords using the standard Huffman decoding. Below appear first the parsing into consecutive k-bit blocks using the original tables, then the parsing into partially overlapping k-bit blocks with the tables relying on the backspaces. Note that for simplicity, we do not deal here with the case that the last block may be shorter than k bits.

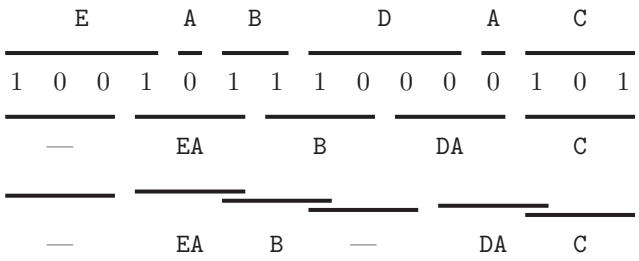


Figure 3. Example using original and reduced tables

To get some experimental results, we used the King James Bible (KJB) as above, and Wall Street Journal (WSJ) issues that appeared in 1989. Huffman codes were generated for large "alphabets", consisting of entire words. Some of the relevant statistics are given in Table 3, and the results are presented in Table 4. The row headed Bit corresponds to the regular bit per bit Huffman decoding. The next row brings the values of the Partial decoding Tables of [3] described in Section 4.1 above, and the row with title reduced corresponds to the variant with the backspaces.

For each of the test databases, the first column brings the maximal size k of the block of bits that is decoded as one unit. The next column, headed bpa is in fact the average value of k used during the decoding. It is the average number of decoded **b**its **p**er table **a**ccess, evaluated as the size of the compressed file in bits divided by the total number of such accesses. The next column brings timing results, in terms of the number of MB that can be processed per second on our test machine. The time for the variant with the full tables for WSJ could not be evaluated, due to exceeding RAM requirements. The last column, headed RAM, gives the size of the required auxiliary storage in MB. We see that the Reduced Tables saved 50 to 80 % of the space required by the partial decoding tables, while using the same k, reducing the decoding rate only by about 20 %.

	full size	compression ratio	number of words	average word length
KJB	3.1 MB	5.15 MB	11669	8.8 bit
WSJ	36.5 MB	5.05 MB	115136	11.2 bit

Table 3. Statistical data on test files

	KJB				WSJ			
	k	bpa	MB/sec	RAM	k	bpa	MB/sec	RAM
Bit	1	1	10.1	0.21	1	1	6,6	2.1
Tables	8	8	0.4	17	8	8	—	197
reduced	8	6.37	13.7	8.7	8	6.35	7.6	34.1

Table 4. Experimental comparison of decoding

5 Conclusion

We presented three examples of applications in various areas of Information Retrieval Systems, for which the inclusion of a backspace character in the alphabet may lead to improved performance. The main message we hoped to convey in this study, is that the definition of alphabets in the broadest sense as used in IR systems, does not have to be restricted to collections of classical items such as letters, words or strings, but may be extended to include also conceptual elements such as a backspace, which, even if not materialized as the other elements, may at times have helpful usages.

References

1. K. C.-C. Chang, H. Garcia-Molina, and A. Paepcke: *Predicate rewriting for translating Boolean queries in a heterogeneous information system.* ACM Trans. on Information Systems, 17(1) 1999, pp. 1–39.
2. Y. Choueka: *Responsa: A full-text retrieval system with linguistic processing for a 65-million word corpus of jewish heritage in Hebrew.* IEEE Data Eng. Bull., 14(4) 1989, pp. 22–31.
3. Y. Choueka, S. T. Klein, and Y. Perl: *Efficient variants of Huffman codes in high level languages*, in Proc. 8-th ACM-SIGIR Conference, Montreal, Canada, 1985, pp. 122–131.
4. A. S. Fraenkel: *All about the Responsa Retrieval Project you always wanted to know but were afraid to ask, Expanded Summary.* Jurimetrics J., 16 1976, pp. 149–156.
5. W. B. Frakes: *Stemming algorithms*, in Information Retrieval, Data Structures and Algorithms, W. B. Frakes and R. Baeza-Yates, eds., Prentice Hall, NJ, 1992, pp. 131–160.
6. D. Huffman: *A method for the construction of minimum redundancy codes*, in Proc. of the IRE, vol. 40, 1952, pp. 1098–1101.
7. S. T. Klein: *On the use of negation in Boolean IR queries.* Information Processing & Management, 45 2009, pp. 298–311.
8. A. Moffat and Z. J.: *Adding compression to a full-text retrieval system.* Software — Practice & Experience, 25(8) 1995, pp. 891–903.
9. G. Salton, A. Wong, and C. S. Yang: *A vector space model for automatic indexing.* Communications of the ACM, 18(11) 1975, pp. 613–620.
10. A. Sieminski: *Fast decoding of Huffman codes.* Information Processing Letters, 26 1998, pp. 237–241.
11. K. Sparck Jones, S. Walker, and S. E. Robertson: *A probabilistic model of information retrieval: development and comparative experiments – parts 1 and 2.* Information Processing & Management, 36(6) 2000, pp. 779–840.
12. C. Tryfonopoulos, M. Koubarakis, and Y. Drougas: *Filtering algorithms for information retrieval models with named attributes and proximity operators*, in Proc. SIGIR Conf., Sheffield, UK, 2004, pp. 313–320.
13. J. Véronis and P. Langlais: *Evaluation of parallel text alignment systems: The ARCADE project*, in Parallel Text Processing, J. Véronis, ed., Kluwer Academic Publishers, Dordrecht, 2000, pp. 369–388.
14. I. H. Witten, A. Moffat, and T. C. Bell: *Managing Gigabytes: Compressing and Indexing Documents and Images*, Van Nostrand Reinhold, New York, 1994.