On the Use of Negation in Boolean IR Queries

Shmuel T. Klein
Department of Computer Science
Bar Ilan University, Ramat-Gan 52900, Israel
Tel: (972–3) 531 8865 Fax: (972–3) 736 0498
tomi@cs.biu.ac.il

Keywords: Boolean queries, negated keywords, distance constraints, concordance, query processing

Abstract: The negation operator, in various forms in which it appears in Information Retrieval queries, is investigated. The applications include negated terms in Boolean queries, more specifically in the presence of metrical constraints, but also negated characters used in the definition of extended keywords by means of regular expressions. Exact definitions are suggested and their usefulness is shown on several examples. Finally, some implementation issues are discussed, in particular as to the order in which the terms of long queries, with or without negated keywords, should be processed, and efficient heuristics for choosing a good order are suggested.

1. Introduction

Formulating a query in an Information Retrieval (IR) System requires an effort as to the correct choice of the query terms. Finding the right balance between terms that may be too broad and others that are overly restrictive is crucial to assure good retrieval performance, as can be measured by recall and precision. In fact, the formulation of queries is an art, and to be successful, one needs, in addition to mastering the query language syntax, also knowledge about the underlying textual database, its language and peculiarities.

It might be objected that in a time where powerful search engines can freely be used by everybody to access an ever growing pool of information on the internet, the usage of complex query languages, requiring a quite sophisticated user, will be less and less frequent. Even the already existing *advanced search* functions, provided by practically all search engines, are rarely used in practice. But this view of the potential set of people seeking some information is very much biased toward users of the internet, which enjoys growing popularity because it is cheap and easily accessible. In particular, a large part of the users access the internet only occasionally and with very simple queries (Spink et al. 2001). On the other hand, there are entire communities of users of Information Retrieval systems that are often focused on specific topics. Examples are lawyers and judges wishing to access juridical databases (such as Lexis), physicians and other health professionals interested in various collections of medical information (such as Medline), researchers in the Humanities studying classical texts in different languages (such as the ARTFL project on the *Trésor de la Langue Française*, Bookstein et al., 1992), etc. Taken

as a part of the full set of search engine users on the internet, these communities might seem relatively small, but in fact they include many thousands of well educated users which are not reluctant to use more sophisticated tools than the most basic queries. Research to provide good query languages can thus be justified.

Indeed, correct query formulation has been a prominent subject in the Information Retrieval literature, and various languages with different application areas have been suggested and studied, see, e.g., Sormunen (2000), Cafarella and Etzioni (2005), or Koubarakis et al. (2006) to cite a few recent ones. Studies relating to specific smaller communities can be found in Mason, 2006, for Lexis, and in Yoo and Choi, 2007, for Medline. The languages used for Database systems (DBS) are usually more involved, permitting a precise description of what is being looked for, but the task solved by a DBS is different from the classical IR task: the underlying text is structured, and meaning is conveyed not just by the words but also by their appearance in specific locations in well defined fields, whereas IR deals with free, unstructured text, and some information need has to be translated into queries which are generally quite fuzzy. XML files have common features with both database and IR systems, and languages have been adapted to treat XML files, ranging from the simplest that could possibly work (O'Keefe and Trotman 2003) to IR inspired languages, as in (Fuhr and Großjohann 2004). A recent study of the processing of metrical constraints for XML files can be found in (Klein 2008).

The present work is a systematic study of the *negation operator* as it appears in its various forms in Information Retrieval applications. In fact, we restrict attention to the Boolean query model, as in Chang et al. (1999), though several alternatives are available, like the classical vector space model (Salton et al. 1975), the probabilistic model, and others. By dealing with the impact of negation on the formulation of Boolean queries, this paper complements the work in Widdows (2003), which considers negation in the context of the vector space model: a negative term is implemented as a vector which is orthogonal to the vectors of the positive terms.

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 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 sometimes be tricky and is not always symmetrical to the use of positive terms. To bring an example from another IR connected application, user queries are often improved by using *relevance feedback*, adding to the query typical terms that appear in documents that have been judged relevant; the negative counterpart, adding typical terms of documents that have been judged non-relevant as *negated* terms, can not always be justified, as reported by Dunlop (1997). The meaning of negated terms in Boolean queries therefore needs precise definitions.

Negative keyword sets are, however, not the only application of negation. Distances can also be negative in proximity searches, and individual characters can be excluded when defining query terms using regular expressions. In the next section, several applications of a negation operator are investigated and exact definitions are proposed. Section 3 then deals with the adaptations to the software which are necessary to support the use of negation. In particular, new heuristics are suggested by which the time necessary to evaluate an efficient order of processing the different query terms is reduced from exponential to polynomial.

2. Using negation operators

In this section, the usefulness of negation operators to various applications in different areas of Information Retrieval is studied.

2.1 Negating keywords in a query

To enable a discussion on possible operators in a query language, one has first to define the query language syntax, which we shall do incrementally.

2.1.1 Simplest syntax

Most search engines allow simple queries, consisting just of a set of keywords, such as

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

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 (Frakes 1992, Goldsmith et al. 2001), and if the text has not been pre-processed, the system has to replace each A_i by a set $\cup_{j=1}^{n_i} A_{ij}$, where each A_{ij} is a grammatical variant of A_i, and n_i is the number of such variants for keyword A_i; generating these variants is in fact the inverse process of stemming. For example, a typical query could be

`solve differential equation`,

seeking documents in which all these terms appear, but instead of `solve`, one could also accept an occurrence of `solving`, `solves`, `solution`, `solved`, etc, and `equation` could as well appear in plural form.

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. The query is thus extended to the form

$$[-/]A_1 \quad [-/]A_2 \cdots [-/]A_m, \tag{2}$$

where we use the standard notation $[-/]$ to indicate the occurrence of either a minus sign or an empty string, that is, each keyword is preceded by an optional minus sign. For example, an auto mechanic could be interested in solving problems related to a differential gear, and thus submit a query of the form

`solve differential` $-$ `equation problem`,

to avoid documents treating differential equations. A major restriction has to be imposed for practical reasons on queries of the type of equation (2), namely that while more than one of the keywords may be negated, not *all* of the keywords can, since the set of all the documents *not* containing certain keywords will most often consist of almost the entire database. Non-negated keywords will be referred to in the sequel as *positive* terms.

2.1.2 Adding global proximity restrictions

While most of the user queries might be described by the simple forms of equations (1) or (2), the possibility of using negated terms may raise certain problems. Suppose one looks for news related to Orléans, a town 100 km south of Paris in France, but wants to avoid texts dealing with New Orleans, which appears much more frequently in the news, especially in the Katrina aftermath. A simple query could thus be

$$-\texttt{New Orleans},$$

but this would be much too restrictive, since all documents containing the term `New` would be rejected, even if the term is not immediately preceding the term `Orleans`. Note that the simple solution of using also the term `France` in the query could badly affect recall, as there might be many text passages mentioning the french town without explicitly adding to each occurrence that the town is in France.

This leads to a further extension of the query syntax, accommodating 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) or (2), 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. Returning to the above query example on differential equations, submitting it with an exact phrase option would retrieve passages containing ...`solving differential equations`..., but not text parts that include ...`solving these differential equations`... or ...`set of differential equations that could not be solved`....

The exact phrase option is therefore also too restrictive and should be relaxed. One possible alternative is to define a bound on what could be called a global *diameter* of the set of occurrences of the query terms: let D be a positive integer limiting the diameter, formally defined as the maximal distance between any pair of the occurrences of the A_i in the text. The query would be rephrased as

$$D: \quad [-/]A_1 \quad [-/]A_2 \cdots [-/]A_m. \tag{3}$$

For example, reusing the last example, if the text contains

...`set of` <u>`differential`</u> <u>`equations`</u> `that could not be` <u>`solved`</u>...,

where the keywords have been underlined, then the diameter is 6, since this is the distance in words from `differential` to `solved`. Thus only if in the query one chooses $D \geq 6$ would this passage be retrieved.

Choosing the right diameter is not a simple task. Obviously, if D is chosen too small, possibly relevant occurrences will not be retrieved, lowering recall; a larger D, on the other hand, might also lead to the retrieval of many non-relevant passages, lowering precision. Moreover, the global nature of the diameter bound is not always appropriate. In a query with keywords `New`, `Orleans` and `Katrina`, one would like to restrict to 1 the distance between the first two terms, but the name `Katrina` could appear several words before or after the occurrence of the pair `New Orleans`.

2.1.3 Adding local proximity restrictions

One therefore needs a syntax allowing more flexibility, putting constraints not just on the global diameter, but on individual pairs of keywords. Consider thus a query containing only positive terms as consisting of m keywords and $m-1$ binary distance constraints, as in

$$A_1 \ \ (l_1 : u_1) \ \ A_2 \ \ (l_2 : u_2) \ \ \cdots \ \ A_{m-1} \ \ (l_{m-1} : u_{m-1}) \ \ A_m. \tag{4}$$

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 $l_i \leq u_i$ 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, some small number of adjacent sentences or the same paragraph.

Such a query language is used for over thirty years at the *Responsa Retrieval Project* (Fraenkel 1976, Choueka 1989). 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 $\mathcal{AWP}$ (Tryfonopoulos et al. 2004). Note that similarly to database queries, the metrical constraints allow a precise description of the required expressions. The usual fuzziness of the Information Retrieval approach is deferred here to user feedback: if the number of retrieved items is too large or too small or the items themselves are not satisfactory, the user can broaden or restrict the query iteratively by changing the constraints and/or the keywords.

In a more general setting, one could also consider extended queries, consisting of several disjuncts, each having a form similar to (4). The requested set of locations to be retrieved is then simply the union of the sets of locations to be retrieved for each of the disjuncts. We may therefore restrict our attention to queries of the form (4).

2.1.4 Formal definition of the set of retrieved locations: only positive terms

For every word W, let $\mathcal{C}(W)$ be the ordered list of the *coordinates* of all its occurrences in the text. For a given textual corpus, the coordinates of a word W are the ordered list of descriptions of the locations of W in the text. These descriptions can have several forms, ranging from a simple (*doc, offset*) pair, where each occurrence is given by the document number and an internal offset in number of words from the start of the document, to more sophisticated hierarchical coordinates, for example, using a 4-level hierarchy to describe a location, a coordinate could consist of a 4-tuple (d, p, s, w), where d is the document number, p is the number of the paragraph within the document, s is the number of the sentence within the paragraph and w is the index of the given word within the sentence. The problem of processing a query of the form (4) consists then, in its most general form, of finding all the m-tuples $\langle a_1, \ldots, a_m \rangle$ of coordinates $a_i = (d_i, p_i, s_i, w_i)$ satisfying

$$\forall i \in \{1, \ldots, m\} \quad \exists j \in \{1, \ldots, n_i\} \qquad \text{with} \quad a_i \in \mathcal{C}(A_{ij})$$

and

$$l_i \leq d(a_i, a_{i+1}) \leq u_i \qquad \text{for } 1 \leq i < m,$$

where $d(x, y)$ denotes the distance in words from x to y, i.e., if $w(x)$ denotes the index of the word x in its sentence, that is, the w-component of the coordinate x is $w(x)$, then

$$d(x,y) = \begin{cases} w(y) - w(x) & \text{if } x \text{ and } y \text{ are in the same sentence} \\ \infty & \text{otherwise.} \end{cases} \tag{5}$$

Every m-tuple satisfying these constraints will be retrieved and the corresponding locations in the text are presented to the user (Choueka et al. 1987). Some reasonable defaults can be chosen when the distance constraints are omitted, for example $(l_i : u_i) = (1 : 1)$, so that a query consisting only of a sequence of keywords without $(l_i : u_i)$ pairs as in (1) is interpreted as a query requesting an exact phrase.

2.1.5 Adding negated keywords

Coupling the use of metrical constraints as in (4) with the possibility of negating some keywords raises the question of an exact definition of what the query stands for in case of ambiguity. If a query contains no negated terms, then processing it from left to right or vice versa will yield the same result. A query

$$A \quad (1:5) \quad B \quad (1:3) \quad C$$

can be interpreted as B following A at distance up to 5, and being itself followed by C at distance up to 3, but equivalently, the query could be thought of as C being preceded at distance up to 3 by B, which itself is preceded by A at distance up to 5. This equivalence is important when one has to decide about the order of processing the terms based on the number of their occurrences, see Section 3 below. However, in the presence of negated terms, the above symmetry is broken: reading from left to right,

$$A \quad (1:5) \quad -B \quad (1:3) \quad C$$

stands for A not being followed, at distance up to 5, by B, but being followed by C at distance up to 3. This is not the same as the interpretation one gets by processing the query right to left, namely C not being preceded at distance up to 3 by B, but being preceded by A at distance up to 5. Suppose a text passage contains the sequence of terms `x x A x x x C x x`, where `A` and `C` are occurrences of the terms A and C, respectively, and `x` stands for occurrences of other terms, which are different from B, then with the former, left to right, interpretation, this passage should not be retrieved, but with the latter, right to left one, it should.

The problem is that the treatment of negated keywords is not symmetrical to that of their non-negated counterparts. In the query $A \ (1:1) \ -B$ one looks for occurrences of A that are not followed by an occurrence of B, but if the query were $-B \ (1:1) \ A$, we are obviously not looking for every non-occurrence of B to be followed by an occurrence of A, but rather for occurrences of A not preceded by B. Mathematically, the result is of course the same, but algorithmically, it makes no sense to check for every word that is not B, so for almost every word in the text, whether it is followed by A. The correct processing retrieves first the coordinates of A and then filters those out that are preceded by B.

Therefore one has to decide whether a keyword is connected to the $(l_i : u_i)$ pair preceding it or to that following it, so for the above example, whether it should be read

$$A \quad [(1:5) \quad -B] \quad (1:3) \quad C \qquad \text{or} \qquad A \quad (1:5) \quad [-B \quad (1:3)] \quad C.$$

This is an arbitrary decision and has to be defined as part of the syntax. So let us define that association of keywords to metrical constraints should be to the left, as in the first alternative, unless there is no such option, that is, all the keywords to the left of the leftmost non-negated one will be right associated (recall that each query must have at least one non-negated keyword). For example, the query

$$-A \quad (1:3) \quad -B \quad (1:2) \quad C \quad (3:6) \quad -D \quad (2:4) \quad -E \quad (1:5) \quad F \qquad (6)$$

will be interpreted as

$$[-A \quad (1:3)] \quad [-B \quad (1:2)] \quad C \quad [(3:6) \quad -D] \quad [(2:4) \quad -E] \quad (1:5) \quad F,$$

thus seeking occurrences of C followed at distance up to 5 by occurrences of F, but these occurrences of C should not be preceded by occurrences of A or B or followed by occurrences of D or E at the given distances. Note also that if several negated terms are adjacent, the associated distance pair refers to the closest preceding (or following for right association) non-negated term. In the example, occurrences of E should not appear at distance 2 to 4 from occurrences of C; interpreting this part of the query as prohibiting the occurrence of E at distance 2 to 4 from a non-occurrence of D makes obviously no sense, since almost every word pair in the text would match that definition.

To give some examples using real English terms:

$$-\texttt{research} \ (1:2) \ \texttt{development} \ (1:1) \ -\texttt{fund} \ (1:9) \ \texttt{European} \ (1:1) \ \texttt{countries}$$

looks for documents related to the development of (or in) European countries, but wants to avoid documents containing the frequent phrases `research and development` or `development fund`, which are considered as noise and would reduce precision.

To get information about United Airlines, an appropriate query seems to be simply `United Airlines`. But in many contexts, the name `United` by itself refers to the airline as in `I took a United flight`; requiring the appearance of the term `Airlines` would thus have a negative impact on the recall. The same is true for the auto mechanic example in Section 2.1.1 above, where the term `gear` did not appear in the query: a differential, in the automobile industry jargon, generally refers to a gear rather than to an equation, so there is no need to mention this explicitly. On the other hand, using `United` alone as query term would produce much noise, reducing precision. A better query could thus be

$$\texttt{United} \ (1:1) \ -\texttt{States} \ (1:1) \ -\texttt{Kingdom} \ (1:1) \ -\texttt{Nations} \ (2:2) \ -\texttt{Emirates}.$$

One could abbreviate the query by using the $\vee$ (OR) operator, thereby defining the set $\cup_{j=1}^{n_i} A_{ij}$ explicitly, but if not all the negated terms require the same distance constraints, as in this example, then using several negated terms allows more flexibility. The reason for using here another distance for `Emirates` is that the term usually appears in the phrase `United Arab Emirates`, so that the query could be rewritten as

$$\texttt{United} \ (1:1) \ -[\texttt{States} \vee \texttt{Kingdom} \vee \texttt{Nations}] \ (2:2) \ -\texttt{Emirates}.$$

2.1.6 Adding negated distances

If in the sentence `x x A x x x B x x`, the distance from `A` to `B` is 4, then the distance from `B` to `A` is -4, in accordance with the definition in (5). In the query syntax, the bounds on the distance constraints can be relaxed to $-\infty < l_i \leq u_i < \infty$, that is, the lower and upper bounds can be negative, as long as l_i is smaller or equal to u_i.

Such an extension is natural and often needed, as a user may not always know the order in which the keywords appear in the text, and in fact all possible orders might be plausible. We saw in the differential equation example above that the term `solve` can possibly appear both before and after the others, so a good query could be

$$\texttt{solve} \quad (-10:10) \quad \texttt{differential} \quad (1:1) \quad \texttt{equation},$$

referring to the fact that `solve` can appear in the range from at most 10 words preceding up to 10 words following the term `differential`. But the negative and positive parts of the range need not be symmetrical: a family name sometimes follows the first name, and sometimes precedes it, so a correct query could be

$$\texttt{Edgar} \quad (-1:2) \quad \texttt{Poe},$$

matching both Edgar Allan Poe and Poe, Edgar Allan.

A further use of negative distance is to extend slightly the set of possible queries. The syntax implies that distance constraints relate to terms that are adjacent in the query, so that in the query $A\ (1:3)\ B\ (2:7)\ C$, the constraint $(2:7)$ is on the distance between B and C; if one wishes to limit both the distance between A and B and also that between A and C, one could reformulate the query as

$$C\ (-7:-2)\ A\ (1:3)\ B,$$

but this method can not be extended to three or more constraints on distances from the same keyword, because at most two keywords can be adjacent to any given one and by the definition of our syntax, metrical constraints operate between codewords that are adjacent in the query.

The use of negative distances also permits to assume without loss of generality that in the general definition of a query as in eq. (7), the leftmost keyword is a positive one, so that association of keywords with $(l_i : u_i)$ constraints is always to the left; this will lead to simplified processing in Section 3 below. The reason for the lack of loss of generality is that negated terms to the left of the leftmost positive term can be moved to its right by reversing the distances. For example, the query in eq. (6) can equivalently be reformulated as

$$C \quad (-3:-1) \quad -A \quad (-2:-1) \quad -B \quad (3:6) \quad -D \quad (2:4) \quad -E \quad (1:5) \quad F.$$

2.1.7 Formal definition of the set of retrieved locations: including negated terms

To formally redefine equation (5) also for the case of negated terms, we need to distinguish between positive and negated keywords. The extended query is given by

$$[-/]A_1 \quad (l_1:u_1) \quad [-/]A_2 \quad (l_2:u_2) \quad \cdots \quad [-/]A_{m-1} \quad (l_{m-1}:u_{m-1}) \quad [-/]A_m. \qquad (7)$$

Define $P(1), P(2), \ldots, P(p)$, with $1 \leq p \leq m$, as the ordered sequence of indices of the positive keywords in (7), and let $N(1), \ldots, N(m-p)$ be the (possibly empty) ordered sequence of the indices of the negated keywords. Define also the function Q such that $Q(N(i)) = P(j)$ if the i-th negated term is attached by its distance constraint to the j-th positive term, that is

$$Q(N(i)) = \begin{cases} P(1) & \text{if } N(i) < P(1) \\ \max\{P(j) \mid P(j) < N(i)\} & \text{otherwise.} \end{cases}$$

For the example of eq. (6), $(P(1), P(2)) = (3, 6)$, $(N(1), \ldots, N(4)) = (1, 2, 4, 5)$ and $Q(1) = Q(2) = Q(4) = Q(5) = 3$. A solution to a query of type (7) is then any p-tuple $\langle a_1, \ldots, a_p \rangle$ of coordinates $a_i = (d_i, p_i, s_i, w_i)$ satisfying

$$\forall i \in \{1, \ldots, p\} \quad \exists j \in \{1, \ldots, n_{P(i)}\} \qquad \text{with} \quad a_i \in \mathcal{C}(A_{P(i)j})$$

and

$$l_i \leq d(a_i, a_{i+1}) \leq u_i \qquad \text{for } 1 \leq i < p,$$

and satisfying the additional constraint on the negative terms, namely that

$$\forall i \in \{1..m-p\} \quad \forall j \in \{1..n_{N(i)}\} \quad \forall b \in \mathcal{C}(A_{N(i)j}) \quad \forall j' \in \{1..n_{Q(N(i))}\} \quad \forall a \in \mathcal{C}(A_{Q(N(i))j'})$$

$$\begin{cases} d(b,a) < l_{N(i)} \quad \vee \quad d(b,a) > u_{N(i)} & \text{if } N(i) < P(1) \\ d(a,b) < l_{N(i)-1} \quad \vee \quad d(a,b) > u_{N(i)-1} & \text{if } N(i) > P(1), \end{cases}$$

where the distance d is defined by eq. (5).

2.2 Negating characters in the definition of keywords

The definition of a query as given in equation (7), with its positive or negated keywords and with metrical constraints, can be regarded as a generalization to a larger scale of a similar process that can be applied to define a single keyword. Indeed, in all of the examples above, and possibly in most real life user queries, the keywords are either given explicitly, or one relies on some automated thesauri to generate, for a given keyword, a set of related morphological variants. There are however many instances in which we may want to define the set represented by a given keyword on our owns.

Again, such tools might be much too involved for the occasional user of a search engine on the internet, but for the sophisticated researcher accessing a large database seeking information on a specific topic, the possibility of precisely defining the set of keywords to be used may be valuable. This is especially true for historical texts or texts in other languages, in which the keywords may appear in a variety of not always predictable spelling variants.

One of the useful tools for such definitions is a *wild-card*, often denoted by *, and matching a variable length (possibly empty) don't care sub-string, which allows the use of truncated terms. For example, `comput*` could stand for any of `computer`, `computers`, `computation`, `computability`, and many other variants of this root; `*mycin` could retrieve a large class of antibiotics; infix truncation can help in case of foreign names that are not always transliterated in the same way, for example, in `Ba*rain`, the `*` could match `h`, `kh`, `ch`, `ha`, and possibly other strings.

2.2.1 Adding constraints to strings matched by wildcards

A variable length wild card character is sometimes not precise enough. In the last example, one would also like to match other spelling variants, such as `Bahrein` with `e`, but using here another wildcard as in `Ba*r*in` could already produce too much noise: that would also match the word `Bargain`. The solution is to restrict the patterns to be matched by means of a regular expression, using for example a syntax adapted from the Unix *gawk* command:

$$\texttt{ba} * \texttt{r}\ [+\texttt{a,e}]\ \texttt{in}$$

would match strings starting with `ba`, including then an `r` which is followed by one of the two letters `a` or `e`, followed by `in`. Note that we chose a syntax separating the optional elements by commas, writing [+`a,b,c`] rather than just [+`abc`], which allows more flexibility, such as the use of variable length strings, for example `Ba`[+`kh,ch,h,ha`]`rain`.

Similar to the + operator, one can also define a minus operator to describe a set of characters to be *avoided* at the given position. Suppose we are looking for documents related to the city of Venice in a multilingual database. Using `Venic*` alone as keyword would probably yield poor results. The problem is that Venice is spelled in a variety of ways in the different languages: `Venezia`, `Venecia`, `Venetia`, `Venetsia`, `Venice`, `Venise`, `Venedig`, `Velence`, `Benétke`, `Wenecja`, `Benátky`, etc. Note that the last two variants have only two letters in common, though `W` and `B` are obviously related, just as `c` and `k`. Trying to match all these by

$$[+\texttt{V,B,W}] * \texttt{en} * [+\texttt{t,s,z,c}] * [+\texttt{a,e,y}] *$$

would again yield unwanted words, like `Vendetta`, `Ventilate` or `Venezuela`, so one may wish to restrict the scope by using negated characters, as in

$$[+\texttt{V,B,W}] * \texttt{en}\ [-\texttt{d,t}]\ * [+\texttt{t,s,z,c}] * [+\texttt{a,e,y}]\ [-\texttt{l}]\ *.$$

It should be noted that the use of negated characters in a keyword is not symmetrical to the use of required characters. The expression `xy*[+z,w]` requires the string `xy`, followed by some arbitrary, possibly empty string, to be followed by `z` or `w`. But the corresponding `xy*[-z,w]` does not mean that we seek for the string `xy`, followed by some arbitrary string S, which is not to be followed by `z` or `w`. Such an interpretation would allow `z` or `w` to appear in S, which in fact voids the negation of its sense. Similarly `xy[+z,w]t` looks for `xy`, followed by `z` or `w`, and then followed by `t`, but the negated version `xy[-z,w]t` does not mean that we want `xy`, not followed by either `z` or `w`, but followed by `t`; to get the latter interpretation, the [−`z,w`] is superfluous, since if `xy` is followed by `t`, it is clearly not followed by any different letter. Moreover, the minus operator can be applied to a single character as in `xy*[-z]t`, whereas this makes no sense with the plus operator, since `xy*[+z]t` is equivalent to just `xy*zt`.

2.2.2 Formal definition of the minus operator

The definition of the minus operator should therefore be amended as follows: a negated component is not the counterpart of a component preceded by a plus sign, but should

be understood as a *modifier* of the wild-card character to which it is adjacent. Thus in the above example, `xy*[−z,w]` matches a string starting with `xy` and followed by some arbitrary string S, but S is not allowed to contain either `z` or `w`. It follows that unlike *positive* terms (those preceded by the + operator), the negated terms must appear in combination with some wild-card. This wild-card could be the `*`, matching an arbitrary, possibly empty, variable length string, but one could envisage other alternatives, such as non-empty strings, strings whose lengths equal to or are bounded by some imposed integer, strings restricted to a certain type (upper case, digits, non-alphanumeric), etc. In any case, for `*` there is no need to require left or right association, because `x * * y` is equivalent to `x * y`, so in

$$\texttt{x} * [-\texttt{z,w}] * \texttt{y}$$

one of the `*`'s is redundant and can be deleted. However, for fixed length or type restricted don't care characters, there could be a confusion, so we decide that left association is used wherever the meaning is not obvious. Using $\#_n$ to stand for an arbitrary string of fixed length n, the interpretation of

$$[-\texttt{a}] * \texttt{xy} \ \#_4 \ [-\texttt{z,w}] \ \#_2 \ [-\texttt{qu}] \ \texttt{t}$$

would thus be: the string `xy`, not preceded by `a`, but followed by some 4-character string that does not contain `z` or `w`, further followed by some 2-character string which is not `qu` and finally followed by `t`.

An interesting alternative for defining the sets A_i of equation (7) by using regular expressions is to combine features from the preceding sections, coupling negated terms with the special distance constraint (0:0). For example

$$\texttt{comput*} \ (0:0) \ - \texttt{computer*}$$

could be used to match words like `computed`, `computation` or `computing` but not `computer` or `computerize`.

2.2.3 Extended example

While for English the possibility of using regular expressions seems to be a nice tool to have, but not absolutely necessary, this is not the case for French with its complicated conjugations, and even less so for German, which allows an almost unlimited series of word concatenations. For certain non-European languages, and especially those using other scripts than the Latin one, this tool is definitely a must. In highly inflected languages like Hebrew and Arabic, single words often reflect what in English would be translated into a multi-word phrase, as various prepositions and articles can be prefixed and pronouns can be suffixed to most of the terms. Moreover, foreign names have to be transliterated, and since there are usually several options to render the same sound, the number of spelling variants of a single term can be large enough to make an explicit enumeration impossible.

Though this work is intended to be mainly theoretical, we now give an extended example showing the usefulness of regular expressions with negation operators for the precise definition of a set of keywords that are supposed to be homonymous on a Hebrew database. The underlying IR system is that of the Responsa Project (Fraenkel 1976, Choueka 1989)

and contains 369 books of Responsa, written by rabbinical authorities all over the world from the 8th century till these very days. The text is mainly in Hebrew and Aramaic, but contains many passages in other languages like German or English, all written using the Hebrew alphabet. The size of the database is about 60 million words.

The example chosen deals again with different spelling variants for the city of *Venice*. The problem above was that the city is named and spelled differently in various languages. The additional complication in the Responsa literature is that, depending on the origin of the author, the transliteration used a variety of alternatives. One of the reasons for the large number of possibilities is the lack of vowels in Hebrew. Certain consonants may be substituted for some of the vowels, but there are no strict rules so that hundreds of spelling possibilities for the same word are not uncommon, as can be seen below. We shall use {`ABGDHWZXtYKLMNSaPCQR$T`} as transliteration of {*aleph, beth, . . . , tav*} respectively.

Table 1: Searching for Venice in the Responsa database

Query	Retrieved occs terms	Relevant occs terms	recall	prec
`#[+W,WW]*N*CY*`	2026 352	1232 159	0.998	0.608
`#[+W,WW][+Y,a]*N*[+C,S]*[+Y,H,A,']*`	1437 234	1227 149	0.994	0.854
`#[+W,WW][+Y,a]*N*[+C,S]*[-N,Q][+Y,H,A,']*`	1316 192	1227 149	0.994	0.932
`#[+W,WW][+Y,a]*[-K,$]N*[+C,S]*[-N,Q][+Y,H,A,']*`	1287 184	1227 149	0.994	0.953

Table 1 lists the various queries and for each, the number of retrieved terms and total number of their occurrences. The next column shows how many of the retrieved terms and occurrences were relevant, that is, indeed dealing with the city of Venice. The last two columns bring the corresponding *recall* and *precision*. Note that these terms are not used here in the usual document retrieval sense;no documents are retrieved and there is no relevance assessment. The elements produced by the regular expression are the term themselves, so one can extend the notions as follows: *precision* will denote the fraction of the retrieved relevant terms among all retrieved terms, that is, those generated by the query (in our case, relevance means referring to the city of Venice), whereas *recall* is the fraction of the retrieved relevant terms among all the relevant terms in the text. One thus needs some knowledge about other occurrences of relevant terms in the text, if any, that might not have been retrieved. Indeed, we found two more relevant occurrences, which explains why recall is not 1.0 in our example.

Recall is based on the fact that 2 more relevant occurrences have been found, which were not matched by the given queries. The `#` stands for a wild-card matching only grammatical prefixes, which, in Hebrew, include prepositions, conjunctions and articles. Note also the use of `[+W,WW]` in the queries: the letter *vav* (`W`) is one of the consonants rendering the *V*-sound in Venice, but `W` serves also, and mostly, as a replacement for some vowels (the *O*- and *OO*-sounds), so that one of the rules calls for doubling the `W` to get the consonant, but this rule is only partially adhered to. Using `W` alone would produce too much noise, as this single letter corresponds to the conjunction *and*.

The first two lines show queries retrieving almost all the relevant variants but producing many irrelevant terms; using negated characters, precision can be significantly improved, without in fact affecting recall. The third and fourth queries are in fact restrictions of the

second one, so no more relevant items are added and recall stays constant; but the second query is not a restriction of the first one, so it should not be surprising that in the passage from the first to the second query, both recall and precision may change.

3. Implementation issues

While Section 2 dealt mainly with the definition of negation operators in various settings, we now turn to implementation problems caused by the presence of negation, and in particular try to set the proper order in which the terms should be processed.

3.1 Processing negated keywords in a query

The way to process queries depends on the algorithmic approach chosen for the Information Retrieval system at hand. For small texts, a direct approach, using *pattern matching* techniques may be feasible. A text of length n can be preprocessed in time $O(n)$ by building a *suffix tree* (Grossi and Italiano 1993), by means of which patterns of length m can subsequently be located in time $O(m)$. But varying distances are hard to deal with, so that in most cases, the processing relies on *inverted files* (Zobel and Moffat 2006).

In the latter approach, the text is scanned and a *dictionary* is produced, including the complete list of all the different words in the database. This list may be organized as a hash table for fast access, but for certain applications, a lexicographically ordered set is preferable, e.g., when several consecutive entries of the dictionary are requested. The dictionary also contains for each word a pointer to the *concordance*, which is often called postings file or simply index. The concordance contains, for each word W, the ordered list $\mathcal{C}(W)$ of its coordinates, which in fact are pointers to its different locations and can have several forms, as discussed above. The order within the lists is usually induced by some global order defined on the documents of the database. This order may be chronological, referring to the time the documents have been created, or it may group documents into classes (by author, topic or some other criterion) and arrange the classes in alphabetical order of their names, etc.

Processing a query of the form $A\,(l:u)\,B$ consists of accessing the dictionary for entries A and B, getting from there pointers to their coordinates in the concordance, and intersecting these lists in the sense that for every pair $(a,b) \in \mathcal{C}(A) \times \mathcal{C}(B)$ one checks whether $l \le d(a,b) \le u$. Since the lists are ordered, the intersection takes only time $O\left(|\mathcal{C}(A)| + |\mathcal{C}(B)|\right))$. For example, if the query is $A\,(-2:8)\,B$ and the text contains the passage

$$\ldots \texttt{x}\ \ a_1\ \ \texttt{x}\ \texttt{x}\ \ a_2\ \ \texttt{x}\ \ b_1\ \ \texttt{x}\ \ a_3\ \ \texttt{x}\ \texttt{x}\ \ b_2\ \ \texttt{x}\ \ldots,$$

where x stands for an arbitrary word and a_i and b_j are occurrences of A and B, respectively, then the requested output of the query is the set of pairs $\{(a_1,b_1),(a_2,b_1),(a_2,b_2),(a_3,b_1),(a_3,b_2)\}$.

If one of the keywords is negated, say the query is $A\,(l:u)\,-B$, then the expected result is the list $\mathcal{C}(A)$ from which all elements a have been purged for which there exists a $b \in \mathcal{C}(B)$ such that $l \le d(a,b) \le u$. Using the above passage with query $A\,(-2:2)\,-B$

would retrieve only the singleton $\{(a_1)\}$; note that the occurrence a_3 was invalidated by b_1 and not by b_2.

3.2 Evaluating the processing order

When longer queries are to be processed, the question of the order in which the keywords should be dealt with arises. For the discussion below, we define the *size* of keyword A_i, denoted by $s(A_i)$, as the total number of occurrences of all its variants, that is, $s(A_i) = \sum_{j=1}^{n_i} |\mathcal{C}(A_{ij})|$. If the sizes of all the keywords are small, the order of processing may not be important, but if there are some keywords with very large sizes, a more careful approach is needed. Intuitively, one would suggest to start then with the "smallest" keyword, i.e., one with minimal $s(A_i)$, and continue processing by order of non-decreasing size. The reason for such intuition is that, for large enough queries and when large sizes are involved, it makes no sense to retrieve all the coordinates of all the terms in the query and storing them temporarily, only to throw away most of them after the intersection. Rather, only the list $\mathcal{L}$ of the coordinates of the smallest keyword is stored, and the lists of the other keywords are processed on the fly, by performing their intersection with $\mathcal{L}$ and gradually removing those parts of $\mathcal{L}$ that did not have matching elements in the other lists. By using *bitmaps* (Choueka et al. 1987), the expected range of relevant coordinates can be dynamically reduced during the processing stages, so that parts of the lists of the larger keywords, which have been deferred to later processing, may not have to be read at all.

There are, however, two problems with that intuition. First, ordering the keywords by size is not always possible, because varying distance constraints may force a more restricted order, and second, even if processing by non-decreasing order is feasible, it is not necessarily the best choice.

If all the constraints are fixed, that is $l_i = u_i$ for $1 \le i \le m$, then any order can be chosen. For example, if the query is $A\,(3:3)\,B\,(2:2)\,C$, then the distance between A and C must be 5; if the sizes of A, B and C are 100, 10000 and 100, respectively, one would first intersect A with C at distance 5 — which will most often result in a list of (a, c) pairs much shorter than 100 — and then check for occurrences of B. But if the query were $A\,(1:3)\,B\,(2:4)\,C$, it would be wrong to try to intersect first A with C, assuming that they must appear at a distance of $1+2=3$ to $3+4=7$ words from each other, and then look for occurrences of B either up to 3 words after A or 2 to 4 words before C. Consider the text passages ... `x` *a* `x` *b* `x x x x` *c* `x` ... and ... `x` *a* `x x x` *b* `x x` *c* `x` ...; the first would be retrieved if the intersection order is (A, C) followed by (A, B), and the second would be retrieved if the intersection order is (A, C) followed by (B, C); however, both passages do not satisfy the query, the first because $d(b, c) = 5 > 4$, the second because $d(a, b) = 4 > 3$.

The processing order is therefore restricted to dealing at every step with one of the keywords that is adjacent to one of those already processed earlier. For example, if the keywords of a query are, from left to right, A, B and C, then the possible processing orders are (A, B, C), (B, A, C), (B, C, A) and (C, B, A). In general, if $T(m)$ denotes the number of possible orders for m adjacent keywords, we have that the last keyword to be dealt with must be either the leftmost or the rightmost, so that $T(m) = 2T(m-1)$; since $T(1) = 1$ we get that $T(m) = 2^{m-1}$.

The fact that the keywords have to be processed by adjacency order also provides an example of the non-optimality of the heuristic suggesting to start with the smallest keyword. Consider the query $A\ (1:5)\ B\ (3:7)\ C\ (1:5)\ D$, and suppose the sizes of A, B, C and D are 100, 10000, 102 and 103, respectively. Starting with A would imply that the next keyword is B, requiring an intersection of 10000 elements with 100, whereas if one intersects first C with D, the number of coordinates left after the intersection will probably be much lower than 100, so the intersection with the 10000 elements of B will be more efficient.

It is true that for many queries, the number m of keywords will be small enough to justify an exhaustive search through the 2^{m-1} options. But this approach is not scalable, and theoretically, the size of the queries should not be bounded. In future applications, with much larger databases and possibly automatically generated query terms, an exponential number of possibilities might be prohibitive.

To choose a priori the best among the 2^{m-1} orders, we suggest the following heuristic, trying to estimate the expected number of comparisons needed for the sequence of intersections induced by each of the orderings of the keywords, and reducing the number of the orderings to be inspected from exponential to polynomial. Since this paper is intended for experts in IR which might not be specialists in algorithms, some necessary background will now be given.

3.2.1 Algorithmic background

One of the major problems dealt with by research in algorithms relates to their *time complexity*, which measures the time required by a specific algorithm, in terms of number of operations rather than in seconds, as a function of the length of the input. It is generally accepted that exponential complexities, like in the above example a number of operations of the order of 2^{m-1} for an input of m query terms, are prohibitive, because even for moderately large m such algorithms cannot be performed in reasonable time. On the other hand, a polynomial complexity is usually considered acceptable.

There are many optimization problems which at first sight, using a straightforward exhaustive search through all the possibilities, seem to require exponential time, but a more careful processing might reduce this to polynomial. One of the standard techniques achieving this is known as *dynamic programming* (Cormen et al. 1990). It is generally based on defining the solution of a problem recursively as a combination of the solutions of some of its subproblems, but the exponential blowup of the recursive procedure — which is often due to the frequent reevaluation of the same subproblems — may be avoided by cleverly choosing the order in which all the subproblems are solved, thereby solving each of them only once. Classical examples of dynamic programming include Floyd's algorithm (Floyd 1967) for finding a shortest path in a graph from every origin node to every target node, finding longest common subsequences of two given input strings, finding the smallest number of edit operations (insert, delete, replace) needed to transform one string into another, and many others.

In our application, we have to compare 2^{m-1} possible orderings of the m keywords of the query and choose one with lowest processing cost according to some pricing heuristic

defined below. Defining the subproblems in the proper way then yields a polynomial time dynamic programming solution. We first deal with queries containing only positive terms, and discuss adaptations to negated terms later.

3.2.2 Processing order with positive keywords only

Let σ be a permutation of the numbers $\{1, 2, \ldots, m\}$ corresponding to one of the 2^{m-1} possible processing orders of the keywords, that is $A_{\sigma(1)}$ is the first element to be chosen, and it is intersected with $A_{\sigma(2)}$, which is either $A_{\sigma(1)-1}$ or $A_{\sigma(1)+1}$ (unless $\sigma(1) = 1$ or m), etc.

The number of comparisons requested by each of the possible orderings depends on the sizes of the lists to be intersected in each iteration, but these sizes change dynamically, since after each intersection, the number of coordinates that may still be relevant is possibly reduced. The first step is to intersect $\mathcal{C}(A_{\sigma(1)})$ with $\mathcal{C}(A_{\sigma(2)})$, requiring $s(A_{\sigma(1)}) + s(A_{\sigma(2)})$ steps. In fact, the processing time for a query $A\ (l : u)\ B$ should also be proportional to $u - l + 1$; if $u = l$, then each coordinate of A can match only one coordinate of B, but for a larger range, the merging process may have to look at certain elements more than once. Nevertheless, this will happen only rarely, even for a large $(l : u)$ range, because the number of occurrences of a given word in the same sentence, or even within the same paragraph, is almost always very low (recall that according to our syntax as defined in Section 2.1.3, the terms of a query are bound to appear in the same or at least in close sentences). We shall therefore ignore the range in our estimates.

The expected size of the coordinates list after the first intersection will be proportional to the sizes of both lists, but smaller or equal to the shorter one. We estimate this size as

$$\frac{s(A_{\sigma(1)}) \times s(A_{\sigma(2)})}{C \times M,} \tag{8}$$

where $M = \max\{s(A_i) \mid 1 \leq i \leq m\}$ is the size of the longest list, and $C \geq 1$ is some constant proportionality factor. It is of course easy to come up with artificial counterexamples, showing cases in which smaller lists gave larger intersection. For the estimations below, we choose $C = 1$, implicitly assuming that the intersection with the largest list does not reduce the list of relevant coordinates. But the bias introduced by the estimates in eq. (8) is the same for all the permutations, and since we need the sizes of the lists only to compare between alternative processing orders, the heuristic can be justified. The estimated cost implied by permutation σ is thus

$$\sum_{i=1}^{m-1} \left[\frac{\prod_{j=1}^{i} s(A_{\sigma(j)})}{M^{j-1}} + s(A_{\sigma(i+1)}) \right]. \tag{9}$$

The time for evaluating (9) for the 2^{m-1} possible σs can be reduced if one realizes that for the given restricted permutations, $\prod_{j=1}^{i} s(A_{\sigma(j)})$ must be in fact $\prod_{j=k}^{\ell} s(A_j)$ for some $1 \leq k < \ell \leq m$. This suggests a dynamic programming approach: let $V(i, j)$ be the minimal expected cost of processing the query terms $A_i \cdots A_j$, for $1 \leq i < j \leq m$. The last keyword to be processed in this subsequence is either A_i or A_j, so that $V(i, j)$ is the

smaller of the two values L and R defined by

$$L = V(i+1, j) + s(A_i) + \frac{1}{M^{j-i-2}} \prod_{k=i+1}^{j} s(A_k) \tag{10}$$

$$R = V(i, j-1) + \frac{1}{M^{j-i-2}} \prod_{k=i}^{j-1} s(A_k) + s(A_j). \tag{11}$$

This yields the program in Figure 1, in which the matrix $D(i, j)$ is used to reconstruct the optimal permutation σ after the minimal cost has been evaluated in $V(1, n)$. $D(i, j)$ is the index of the last element to be processed in the optimal arrangement of $A_i \cdots A_j$, so it is either i or j.

```
for i ⟵ 1 to m − 1
        V(i, i + 1) ⟵ s(A_i) + s(A_{i+1})
for diff ⟵ 2 to m − 1
        for i ⟵ 1 to m− diff
                j ⟵ i+ diff
                evaluate L and R using eq. (10) and (11)
                if L < R     then  { V(i, j) ⟵ L;   D(i, j) ⟵ i}
                else               { V(i, j) ⟵ R;   D(i, j) ⟵ j}
        end for i
end for diff
```

FIGURE 1: *Evaluating the optimal cost (only positive terms)*

```
Fill-sigma(i, j, k)
        if k > 0 then
                if D(i, j) = i then
                        σ(k) ⟵ i
                        Fill-sigma(i + 1, j, k − 1)
                else   /* D(i, j) = j */
                        σ(k) ⟵ j
                        Fill-sigma(i, j − 1, k − 1)
```

FIGURE 2: *Finding the optimal permutation σ (only positive terms)*

The complexity has been reduced to $O(m^2)$. The recursive procedure Fill-sigma in Figure 2 can then be called with parameters $(1, m, m)$ to fill, in time $O(m)$, the optimal permutation σ backwards from the end.

3.2.3 Processing order allowing negative keywords

Several adaptations are necessary to extend the above algorithm also to cases including negated keywords. For the positive terms, there was a clear priority for those with smaller

size, since they both required fewer comparisons and also had a tendency to better reduce the remaining set of coordinates. The reason for not using the order implied by non-decreasing sizes was that varying distance constraints forced us to proceed by adjacency only. For negated terms, a decision to make the order of processing dependent on their size seems not so clear cut: on the one hand, the smaller the list of coordinates, the less comparisons are needed, but on the other hand, since it is the *non*-occurrence we are looking for, the smaller the list, the *more* coordinates are generally left after the intersection.

A further difference is the possible order of processing. For positive terms, this order was restricted, because, e.g., in the query $A\,(1:3)\,B\,(2:5)\,C$, there are limits both on the distances from A to B, and from B to C. But as explained earlier in Section 2.1, a negated term is only associated with one of the positive ones, so in fact, the negated terms can be processed in any order. Using again the example of eq. (6), if C is the first keyword chosen, the subsequent intersections with $-A$, $-B$, $-D$, $-E$ and F can be performed in any of the $5! = 120$ possible orderings.

To derive an upper bound on the time needed to evaluate the optimal processing order, recall that we defined $P(1), \ldots, P(p)$ as the ordered sequence of indices of the positive keywords in the query. The relative order of the positive keywords is still restricted as before, thus their number is 2^{p-1}; there are at most $m - p$ negative keywords, and these can be processed in any order; and there are at most $\binom{m}{p}$ possible orders to merge the set of p positive with the set of $m - p$ negated keywords. The total number of possibilities is therefore bounded by $\binom{m}{p}(m-p)!2^{p-1} = \frac{m!}{p!}2^{p-1}$. This bound is not very tight, for example, in the query of eq. (6), the order F, A, B, C, D, E is not possible: as a matter of fact at each step of the processing, only negated terms relating to positive terms that have already been handled can be adjoined. There are, however, cases, for which the bound is not exaggerated: consider a query with a single positive and $m - 1$ negated keywords; the number of possible processing orders is then $(m-1)!$.

The following observation permits a dynamic programming solution as above, reducing the processing time to polynomial. The intersection process is done by pairs, and when keyword A_i is adjoined to the process, it contributes $s(A_i)$ comparisons to the total cost, regardless of it being negated or not. Negation only affects the size of the remaining intersected coordinate list, whether it should be proportional to $s(A_i)$ or to $M' - s(A_i)$, where M' is some constant larger than the longest list of a negated keyword. Thus when computing the total cost of a given processing order, all the values $s(A_i)$ will ultimately appear exactly once in the sum, so that in fact this part of the sum is equal for any processing order. But this implies that the optimal way to process the negated keywords is to choose, every time there is a choice, the one with the largest $s(A_i)$, since this will best reduce the size of the remaining coordinate list.

The problem is therefore to find the best order, where that of the positive keywords is restricted by adjacency, and that of the negated terms is given, at each stage, by the order induced by $s(A_i)$. Of course, for each partial set of keywords there is only a subset of the negated keywords that is relevant.

Let CNI(i, j) (Corresponding Negative Indices) be the set of the indices of those negated terms that are neighbors to one of the (positive) keywords in the range from $P(i)$ to $P(j)$,

that is, if the currently processed positive keywords are $A_{P(i)} \cdots A_{P(j)}$, then each of the negated keywords A_w, with $w \in \mathrm{CNI}(i,j)$ can be processed next. We assume that $\mathrm{CNI}(i,j)$ is given as a list according to the non-increasing order of the corresponding keyword sizes. The following example should clarify these definitions. Consider the query given in the top line of the left table in Figure 3 (metrical constraints were omitted for clarity), and suppose the sizes of the negated terms are as given in the row $s(A_i)$. The left table also shows the corresponding values of $P(i)$ and $N(i)$, and the matrix to the right gives the sets $\mathrm{CNI}(i,j)$ for $1 \le i \le j \le 4$.

Query	A	$-B$	C	$-D$	$-E$	F	G	$-H$
i	1	2	3	4	5	6	7	8
$s(A_i)$		500		200	10000			1000
P	$P(1)$		$P(2)$			$P(3)$	$P(4)$	
N		$N(1)$		$N(2)$	$N(3)$			$N(4)$

CNI	1	2	3	4
1	(2)	(5,2,4)	(5,2,4)	(5,8,2,4)
2		(5,4)	(5,4)	(5,8,4)
3			$\emptyset$	(8)
4				(8)

FIGURE 3: *Example of sets of indices of negated keywords*

When dealing with positive keywords, we estimated the effect of intersecting the current list of coordinates with the list $\mathcal{C}(A_i)$ as reducing the size of the current set by a factor of $s(A_i)/M$ (see eq. (8) and (9)). For negated keywords, we assume the reduction is by a factor of $1 - s(A_i)/2M'$, where M' has been defined above as $M' = \max\{s(A_{N(j)}), j = 1, \ldots, m-p\}$. The idea is that the larger the set, the lower the probability of its elements to be avoided, but the denominator here is chosen as $2M'$ and not just M' as in the positive case, reflecting our assumption that even when dealing with the largest negated keyword, the set of remaining coordinated should not be empty.

To extend the above dynamic programming procedure, define $V(i,j,\ell)$ as the minimum expected processing cost of the sub-query $A_{P(i)} \cdots A_{P(j)}$ in which ℓ negated keywords are allowed to participate, $0 \le \ell \le |\mathrm{CNI}(i,j)|$. To define $V(i,j,\ell)$, the following notations are useful: let $C(i,j,\ell)$ be the expected size of the coordinates list after intersecting $A_{P(i)} \cdots A_{P(j)}$ and the ℓ first values in $\mathrm{CNI}(i,j)$, and denote by $F(\ell, X)$ the subset including the first ℓ elements of the ordered sequence X. We then have

$$C(i,j,\ell) = \frac{1}{M^{j-i-1}} \prod_{k=i}^{j} s(A_{P(k)}) \prod_{k \in F(\ell, CNI(i,j))} \left(1 - \frac{s(A_k)}{2M'}\right).$$

The value of $V(i,j,\ell)$ can now be recursively defined by observing that the last element to be processed is either a positive one, in which case it must be $A_{P(i)}$ or $A_{P(j)}$, or a negative one, in which case it must be $A_{w(i,j,\ell)}$, where $w(i,j,\ell)$ denotes the ℓ-th element of the list $\mathrm{CNI}(i,j,\ell)$. This yields the following recursion:

$$V(i,j,\ell) = \min \left\{ \begin{array}{l} V(i+1,j,\ell) + C(i+1,j,\ell) + s(A_{P(i)}), \\ V(i,j-1,\ell) + C(i,j-1,\ell) + s(A_{P(j)}), \\ V(i,j,\ell-1) + C(i,j,\ell-1) + s(A_{w(i,j,\ell)}) \end{array} \right\}, \tag{12}$$

where the minimum should be understood as being applied only to the first two lines for the special case $\ell = 0$.

The corresponding program is given in Figure 4. The complexity is clearly bounded by $O(m^3)$, but the number of iterations is in fact at most $\frac{1}{2}(m-p)p^2$, which is at most $\frac{2}{27}m^3$

```
for i ⟵ 1 to m − 1
        V(i, i + 1, 0) ⟵ s(A_i) + s(A_{i+1})
for ℓ ⟵ 0 to m − p
        for diff ⟵ 2 to p − 1
                for i ⟵ 1 to p − diff
                        j ⟵ i+ diff
                        if ℓ ≤ |CNI(i, j)| then
                                evaluate V(i, j, ℓ) according to eq. (12)
                        D(i, j, ℓ) ⟵ index of last element adjoined (P(i), P(j) or w(i, j, ℓ))
                end for i
        end for diff
end for ℓ
```

FIGURE 4: *Evaluating the optimal cost (including negative terms)*

```
Fill-sigma(i, j, ℓ, k)
        if k > 0 then
                if D(i, j, ℓ) = P(i) then
                        σ(k) ⟵ P(i)
                        Fill-sigma(i + 1, j, ℓ, k − 1)
                elseif D(i, j, ℓ) = P(j) then
                        σ(k) ⟵ P(j)
                        Fill-sigma(i, j − 1, ℓ, k − 1)
                else   /* D(i, j, ℓ) = w(i, j, ℓ) */
                        σ(k) ⟵ w(i, j, ℓ)
                        Fill-sigma(i, j, ℓ − 1, k − 1)
```

FIGURE 5: *Finding the optimal permutation σ (including negative terms)*

(for $p = \frac{2}{3}m$). To recover the optimal permutation, the recursive procedure Fill-sigma of Figure 5 is called with parameters $(1, p, m-p, m)$ and works in time $O(m)$.

4. Conclusion

Negation has been part of most retrieval systems before and the purpose of this work was not to reinvent some unknown features. The aim was rather to try to give exact definitions in ambiguous cases and increase the popularity of using different forms of negation in IR queries by showing how to use such queries in many examples. We conclude that in spite of its connotation, *negation* can have some very *positive* aspects.

References

Bookstein A., Klein S.T., Ziff D.A. (1992), A systematic approach to compressing a full text retrieval system, *Information Processing & Management* **28**, 795–806.

Cafarella M.J., Etzioni O. (2005), A search engine for natural language applications, *Intern. World Wide Web Conf.*, Chiba, Japan, 442–452.

Chang K.C.-C, Garcia-Molina H., Paepcke A. (1999), Predicate rewriting for translating Boolean queries in a heterogeneous information system, *ACM Tarns. on Information Systems* **17**(1), 1–39.

Choueka Y. (1989), 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) 22–31.

Choueka Y., Fraenkel A.S., Klein S.T., Segal E. (1987), Improved Techniques for Processing Queries in Full-Text Systems, *Proc. 10-th ACM-SIGIR Conf.,* New Orleans, 306–315.

Cormen T.H., Leiserson C.E., Rivest R.L. (1990), *Introduction to Algorithms*, MIT Press.

Dunlop M.D. (1997), The effect of accessing nonmatching documents on relevance feedback, *ACM Trans. on Information Systems* **15**(2) 137–153.

Floyd R.W. (1962), Algorithm 97: Shortest path, *Communications of the ACM* **5**(6), 345.

Fraenkel A.S. (1976), All about the Responsa Retrieval Project you always wanted to know but were afraid to ask, Expanded Summary, *Jurimetrics J.* **16**, 149–156.

Frakes W.B. (1992), Stemming algorithms, in Frakes W.B., Baeza-Yates R. (eds), *Information Retrieval Data Structures and Algorithms*, Prentice Hall, NJ, 131–160.

Fuhr N., Grossjohann K. (2004), XIRQL: an XML query language based on information retrieval concepts, *ACM Transactions on Inf. Systems* **22**, 313–356.

Goldsmith J.A., Higgins D., Soglasnova S. (2001), Automatic language specific stemming in Information Retrieval, *Proc. Workshop of Cross-Language Evaluation Forum,* Lisbon 2000, *LNCS* **2069**, 273–284.

Grossi R., Italiano G.F. (1993), Suffix trees and their applications in string algorithms. *Proc. 1st South American Workshop on String Processing (WSP 1993)*, 57-76.

Klein S.T. (2008), Processing queries with metrical constraints in XML based IR systems, *Journal of the American Society for Information Science and Technology* **59**(1), 86–97.

Koubarakis M., Skiadopoulos S., Tryfonopoulos C. (2006), Logic and computational complexity for Boolean information retrieval, *IEEE Trans. on Knowledge and Data Engineering* **18**(12), 1659–1666.

Mason D. (2006), Legal Information Retrieval study – Lexis Professional and Westlaw UK, *Legal Information Management* **6** 246–250.

O'Keefe R.A., Trotman A. (2003), The simplest query language that could possibly work, *Proc. second INEX Workshop*, 117–124.

Salton G., Wong A., Yang C.S. (1975), A vector space model for automatic indexing, *Communications of the ACM* **18**(11) 613–620.

Spink A., Wolfram D., Jansen B.J., Saracevic T. (2001), Searching the Web: the public and their queries, *Journal of the American Society for Information Science* **53**(2), 226–234.

Sormunen E. (2000), A novel method for the evaluation of Boolean query effectiveness across a wide operational range, *Proc. SIGIR Conf.*, Athens, Greece, 25–32.

Tryfonopoulos C., Koubarakis M., Drougas Y. (2004), Filtering algorithms for information retrieval models with named attributes and proximity operators, *Proc. SIGIR Conf.*, Sheffield, UK, 313–320.

Widdows D. (2003), Orthogonal negation in vector spaces for modelling word-meanings and document retrieval, *Proc. Conf. Assoc. for Computational Linguistics*, Sapporo, Japan, 136–143.

Yoo S., Choi J. (2007), Improving Medline document retrieval using automatic query expansion, *Proc. ICADL Conf.,* Hanoi, Vietnam, 241–249.

Zobel J., Moffat A. (2006), Inverted files for text search engines, *ACM Computing Surveys* **38**(2), 1–56.