Computación y Sistemas Vol. 6 No. 3 pp. 167 - 182
© 2003, CIC - IPN. *ISSN 1405-5546* Impreso en México

Exact and Approximate Prefix Search under Access Locality Requirements for Morphological Analysis and Spelling Correction

La Búsqueda Exacta y Aproximada de Prefijos Bajo los Requerimientos del Acceso Local, para el Análisis Morfológico y Corrección de Ortografía

Alexander Gelbukh
Centro de Investigación en Computación-IPN
Av. Juan de Dios Bátiz s/n esq. Miguel Othón de Mendizabal,
Unidad Porfesional Adolfo López Mateos, Col. Sn Pedro Zacatenco
Del. Gustavo A. Madero, México D.F. C.P. 07738
E-mail: gelbukh@cic.ipn.mx, gelbukh@gelbukh.com

Article received on December 12, 2000; accepted on March 18, 2003

Abstract

A data structure useful for prefix search in a very large dictionary with an unlimited query string is discussed. This problem is important for morphological analysis of inflective languages, including particularly difficult cases such as German word concatenation or Japanese writing system that does not use spaces; similar tasks arise in DNA computing. The data structure is optimized for locality of access: to find all necessary records, access to only one block (page) of the main data storage is guaranteed, which significantly improves performance. To illustrate its usefulness, the algorithms of exact and approximate search are described, with application to morphological analysis and spelling correction. The algorithms for building, exporting, and updating the data structure are explained.

Keywords: prefix search, approximate prefix search, approximate string matching, morphological analysis, spelling correction, natural language processing, DNA computing.

Resumen

Se presenta una estructura de datos que es útil para la búsqueda de prefijos en un diccionario muy grande con una petición de entrada no limitada. Este problema es importante para el análisis morfológico de los lenguajes flexivos, incluyendo los casos particularmente difíciles tales como encadenamiento de palabras en el alemán o el sistema de la escritura japonés que no utiliza espacios; las tareas similares se presentan en el procesamiento computacional de ADN. La estructura de datos es optimizada para el acceso local: para encontrar todos los registros necesarios, se garantiza el acceso a sólo un bloque (página) del dispositivo principal de almacenamiento de datos, lo que significadamente mejora el rendimiento. Para ilustrar su utilidad, se describen los algoritmos de la búsqueda exacta y aproximada, aplicados al análisis morfológico y la corrección de ortografía. Se explican los algoritmos para la construcción, exportación y actualización de la estructura de datos.

Palabras clave: búsqueda de prefijos, búsqueda aproximada de prefijos, comparación aproximada de cadenas, análisis morfológico, corrección de ortografía, procesamiento de lenguaje natural, computación de ADN.

1 Introduction

A typical database system is, basically, a device that can answer a simple question: Which records have the key x exactly equal to the given query string s? In some cases we are interested in another question: Which records have the key x that is a *prefix* of the given string s? By prefix, we mean an initial substring.

Definition **1**. By $x \prec s$, we denote the fact that the string x is a prefix (initial substring) of the string s, i.e., $\exists$ a string y (possibly empty) such that $s = x\,y$. Here x is a finite-length string and s, y are finite or infinite strings.

Thus, given a query string s, finite or infinite, and the database D, we can distinguish the following tasks of finding all records with the keys x such that:

Task **1**. Exact search: $\{x \in D \mid x = s\}$; s is finite.

Task **2**. Approximate search: $\{x \in D \mid x \approx s\}$ for some criterion of similarity $\approx$; s is finite.

Task **3**. Prefix search: $\{x \in D \mid x \prec s\}$.

Task **4**. Approximate prefix search: $\{x \in D \mid \exists\, x' \approx x\colon x' \prec s\}$ for some criterion of similarity $\approx$.

In this paper, we are interested in the latter two cases, prefix and approximate prefix search, and specifically when the dictionary D is very large in terms of the number of records.

Note that unlike classical database search—Task 1, in approximate prefix search the length of the key x looked for is not known a priori. As we will see below, for a long enough query string s even its length needs not to be specified in the query, i.e., s can be considered infinite. This task has some specificity and raises some technical issues that the classical task of exact key matching does not face—specifically, the problems related to data access locality.

In this paper, we will discuss these issues and propose a data structure optimized with respect to them, along with the basic operations and sample applications. The structure proves to be a variation of a 2-level B-tree with slight redundancy.

In Section 0, we explain the motivation for the task of prefix search in very large dictionaries, formulate the technical problem of access locality faced with by the prefix search task, and discuss the related work. In Section 0, we explain the proposed data structure, step-by-step improving the naïve approach to the solution. In Section 0, we give a sketch of the algorithms for the basic operations with this data structure. Finally, in Section 0 we illustrate its usefulness for morphological analysis of natural languages, and specifically, for spelling correction, i.e., approximate string matching combined with decomposition of the string by a set of dictionaries.

2 The Problem: Prefix Search in Very Large Dictionaries

Existing algorithms for prefix search, unlike those for exact search, require addressing different physical locations in the dictionary, which causes very inefficient behavior with block-oriented physical storage devices such as hard disks or virtual memory mechanism.

Below we give more details on the prefix search and the data access locality problem. The section 0 justifies the task of prefix search in very large dictionaries. The section 0 explains the technical problem arising in the search in large dictionaries. Finally, the section 0 describes the related work found in the literature and some other approaches to the mentioned problems.

2.1 Prefix Search in Very Large Dictionaries: Motivation for Morphological Analysis and DNA Computing

One of the main sources of prefix search tasks are the cases of analysis of the strings formed as a concatenation, without any delimiter, of the substrings belonging to the same or different dictionaries: $s = s_1 \dots s_n$, $s_i \in D$ or $s_i \in D_i$.

A typical example is the analysis of a DNA chain s as a concatenation $g_1 \dots g_n$ of individual genes $g_i \in D$, while the number of genes $|D|$ can be very large.

Our main motivation, however, was morphological analysis of words in highly inflective languages such as Spanish, French, or Russian. In natural language morphology, an example of the task is the following. Let us consider a dictionary (a database) of Spanish stems[1] like the one shown in Table 1.

Table 1. A fragment of Spanish dictionary.

Key	Value	
.....	.....	.....
clar-	adjective	*claro*
co-	prefix	*co-*
com-	verb	*comer*
con-	preposition	*con*
concentr-	verb	*concentrar*
const-	verb	*constar*
constancia-	noun	*constancia*
constante-	adjective	*constante*
constat-	verb	*constatar*
constelación-	noun	*constelación*
constipad-	noun	*constipado, -a*
constru-	verb	*construir*
construcción-	noun	*construcción*
constructiv-	adjective	*constructivo*
constructivismo-	noun	*constructivismo*
consult-	verb	*consultar*
.....	.....	.....

In fact, the dictionary is much denser (Diccionario, 1992); we have omitted some lines for brevity. Spanish words have highly inflected forms. For example, the stem[2] *constipad-* gives rise to the wordforms *constipado, constipada, constipados, constipadas*; the stem *clar-* to the wordforms *claro, clara, claros, claras, clarísimo, clarísima, clarísimos, clarísimas, claramente*; the stem *constru-* to about 700 wordforms such as *construir, construyo, construido, constrúyanmelo*[3], etc. As we see, finding the stem given a word form is the prefix search, Task 3 above.

This task is ambiguous, since we need to find all prefixes and not just the longest one. Indeed, here is an example where the longest prefix is not the right one. Suppose the Spanish dictionary contains such stems as *ajen-* for (*derecho*) *ajeno*, *moren-* for (*color*) *moreno*, *escalen-* for (*triángulo*) *escaleno*, etc. Then, for the (subjunctive mode) verb forms like *ajen*, *moren*, *escalen*, the stems mentioned above are the longest ones, while the shorter stems *aj-*(*ar*), *mor-*(*ar*), *escal-*(*ar*) are the correct ones. Since making any linguistically meaningful decisions is not the business of the string search mechanism, we conclude that the dictionary search procedure should enumerate all prefixes it found, starting from the longest one since usually they have higher probability to prove to be the right one.

Though the examples of stem ambiguity are rather rare in Spanish, they are much more frequent in languages with more developed morphological inflection, such as Russian, Finnish, Turkish, just to mention a few. What is more, in German compound words are very frequent, so that one word can consist of many stems: for example, the first stem of *kommunikationstechnik* is *kommunikation-*. The extreme situation is, say, in Japanese writing system which just does

[1] The hyphen at the end of the stem is shown just for readability and is actually not a part of the stem.

[2] The meaning of the Spanish wordforms is not important for the discussion, so we do not give any glosses.

[3] We will explain below how we deal with letter alternations such as accents.

not use spaces between words,[4] much like the situation with DNA.

In such cases, when it is impossible to separate a single word before addressing to the stem dictionary, the query string *s* used to look for its stem (prefix) *x* is actually the whole text, i.e., can be thought of as unlimited in length.

On the other hand, there is another good reason to consider the whole unlimited text as the query string to search the stems, even in the languages that do mark word boundaries with spaces or other punctuation signs. In all such languages, such word delimiters are sometimes overused. For example, a Spanish preposition *a través de* contains two spaces and thus looks like three words, though there are no technical reasons to treat it so. Thus, even in Spanish, like in Japanese or German, not always we can easily detect the word boundaries![5] Fortunately, in order to use the search mechanism discussed in this paper, we really do not need to: it is enough to include the word *a_través_de* in the dictionary, with its both spaces, as one record key.

Thus, morphological analysis of natural languages requires prefix search in very large (stem) dictionaries.

2.2 The Data Locality Problem and its Motivation for Morphological Analysis

The second consideration contributing to the problem is of purely technical nature—the way the block-oriented storage devices process data access.

Data locality means access to a small region of data storage per operation. Suppose you need to retrieve the first name, last name, and age of a person from a file.

- If all you need to do is to read a line 1234 where this data is stored, then this operation *is* local.
- On the other hand, if you need to read the name from the line 678, surname from 901, and age from 2345, then this way of data retrieval is *not* local since you have to address several different regions of data storage to fulfill a single operation.

With most of data storage devices currently in use, local access is on average much faster than non-local access is. The most obvious example is a tape where the access time is proportional to the distance between the addressed locations. Disk storage, be that a hard disk or a CD, is less sensible to data locality but still sensible: in this case, the access time is roughly proportional to the number of addressed locations, so that in the example above, the second variant is trice slower than the first one. This is because the data are exchanged with a disk storage device by blocks, or sectors, of fixed size. Reading, say, a kilobyte of sequential data takes nearly the same time as reading one byte, but repositioning the reading head to another location takes significantly more time.

It might be argued that large amount of available memory renders the data locality problem unimportant since the cost of addressing to the random access memory (RAM) is proportional just to the number of retrieved bytes regardless to their location. However, this is not completely true. Under the operating systems widely used nowadays, like Windows 98 or NT, the large (virtual) memory is much "less random access" than in good old times of DOS.

Indeed, most of the physical memory is occupied by the active concurrent programs and the operating system itself, while random parts of any data loaded in memory are swapped out to the hard disk. Thus, under such systems, loading a very large dictionary into memory makes little difference from storing it on the disk. Simple experiments show that data locality problem keeps its importance even for a data structure that does "fit in memory"—i.e., has such size that it could be stored in RAM if no operating system nor concurrent processes existed.

On Intel processors, the quantum—called page—of memory is four kilobytes. While sequential accesses to the same page can be considered local, access to a different page with a certain probability causes hard disk read and write operations because of virtual memory swapping.

This, for an algorithm to work fast, it should mostly address data within a four-kilobyte address window and avoid random accessing to many addresses differing more than four kilobytes.

The data locality problem is especially important for natural language analysis applications that use huge amount of information in parallel for the analysis of each phrase. While each individual dictionary—a morphological dictionary, syntactic dictionaries such as subcategorization and lexical attraction (collocation) (Yuret, 1998) dictionaries, a semantic network dictionary (Cassidy, 2000; Fellbaum, 1998), world knowledge dictionaries (Lenat & Guha, 1990), etc.—might fit in the physical memory, all of them together will not. Meanwhile, the processing of each phrase requires using of all of these resources in parallel, before the processing of the next phrase begins, and with tens of thousands of different words occurring in essentially random order. Thus, natural language processing tasks are unique in their highly intensive memory using that causes intensive virtual memory disk swapping. With this, no single large dictionary can be considered "completely stored in RAM."

However, only morphological analysis faces the problem of data locality. All other dictionaries—syntactic, semantic, and world knowledge ones—can be stored in a classical database with exact search—Task 1 in the terminology of Section 0, for which data locality problem can be easily avoided. With this, not more than one data storage access per word is necessary for each of the dictionaries but the morphological one. Thus, the data locality problems with

[4] Though switching from Kana letters to Kanji hieroglyphs in many cases indicates the word boundary, still the problem of the text flow segmentation into words is more difficult in Japanese than in, say, English.

[5] English examples: *in order to*, *each other*, *New York*; Russian example: *vo chto by to ni stalo* 'by all means'.

which the morphological analysis faces is the bottleneck of the data access during natural language analysis.[6]

Returning to the search algorithms, let us consider an example of a data locality-friendly one and an example of one that is not.

- *The classical hash table* is data-local. To find the record with the key *construcción*, its hash value is calculated, which defines the address starting from which (at this address or at some near one in case of collision) the record is located. However, hash table is good for exact search—Task 1, but it does not work for prefix search—Task 3, for which a potentially infinite number of hypotheses are to be tried.
- *Binary search* in an alphabetically ordered list works fine for prefix search task. The main search operation in this case is that of finding the alphabetic place of a given string in the list.[7] However, this structure is *not* local: for a binary search in, say, a 1000 items, one needs to address 10 different locations in the list, some of which are quite far from each other, see Fig. 1.

With binary search, the problem is not only in a wrong algorithm. Prefix search task for an alphabetically ordered list is inherently non-local, at least if all possible prefixes are to be found. Indeed, let us consider the search procedure for the Spanish verb form *consto*. Its alphabetic place in the dictionary in Table 1 is after *constelación-*, while the true answer is *const-*, and what is more, since the prefix search procedure should find all possible prefixes, the records *con-*, *co-*, and *c-* are also to be retrieved. In (Diccionario, 1992), the distances between these locations are as follows: *c-* ... (9327 words) ... *co-* ... (2101 words) ... *con-* ... (1445 words) ... *const-* ... (95 words) ... *constituyentes-* (*consto*), see Fig. 2. Since these are the results of the query and thus any algorithm is to address these locations, no algorithm based on this data structure can be local.

In this paper, we will discuss a data locality-friendly modification of alphabet search. Namely, at the cost of slight redundancy, our data structure guarantees that after retrieval of exactly one block (page) of data from the main storage, all the records with the keys being initial substrings of the given string are found.

2.3 Related Work

There is a vast literature on the prefix-matching data structures, such as tries, B-Trees, Prefix B-Trees, B+-Trees, extendible hashing, etc. (Aho, 1990; Bayer & Unterauer, 1977; Comer, 1979; Gusfield, 1997). The theory of such structures is developed in the context of the database index management rather than on dictionary application; thus, these structures and the corresponding algorithms are optimized with respect to efficient updating (insertions and deletions) (Johnson & Shasha, 1993), which is rather irrelevant for natural language processing. On the other hand, these structures are not optimized for the access locality (see the next section) that is the main topic of our paper. Actually, what we propose is similar to a 2-level B-tree (that we later extend to an *n*-level B-Tree) with the additional feature of repetition of some keys, which optimizes the structure with respect to data access locality.

Data locality is a well-known issue in programming and computer science (Knuth, 1998). Recently there is no much discussion of this topic since as the amount of the available RAM increases, the problem looses its importance for many applications. As we have shown, however, this is not the case of morphological analysis module of a natural language analysis system.

In computational linguistics, there are two major approaches to morphological analysis, one based on morphonological transformation and another on simple dictionary lookup. The first approach is exemplified by the two-level morphology (Koskenniemi, 1983). With such methods, the program first guesses what the stem of the wordform could be, and then checks each hypothesis against the dictionary. E.g., for the wordforms like *ladies* and *caries*, the hypotheses *lady-* and **cary-* would be tried, correspondingly. Thus, no prefix search is required. However, the method is not data-local due to multiple hypotheses: for the example above, the hypothetical stems **ladies-* and *caries-*, as well as many others, are to be looked up in the dictionary, too.

The other, currently more widely used approach to morphological analysis was called by Hausser (1999) *allomorph method*. With this method, the wordform is considered as a concatenation of substrings (allomorphs) listed in separate dictionaries, as we discussed in the Section 0. There are two variations of this method that can be conventionally called right-to-left and left-to-right analysis (Gelbukh, 1995). With the first method, a word is separated from the letter string, then all possible affixes are determined by the small lists of affixes, and then each hypothetical stem is looked for in the dictionary (Sidorov, 1996). This leads to the same problems with multiple hypotheses as with two-level morphology discussed above.

[6] Here we do not discuss the CPU time of analysis since it depends solely on the speed of the CPU and not of the data storage device.

[7] By alphabetic place of a given string *s* in the alphabetically ordered list *D*, we mean the position to which *s* could be inserted into *D*, see Definition 2 below.

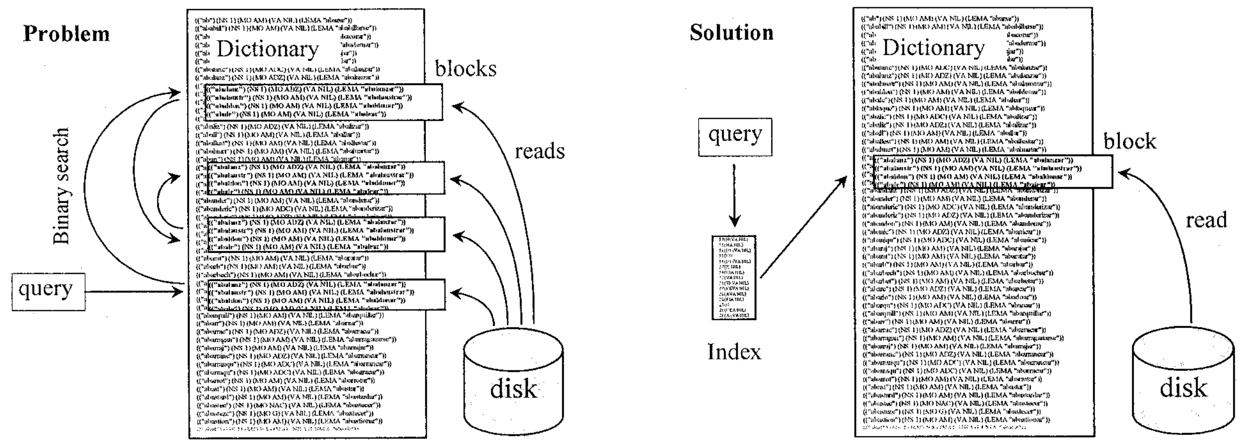


Figure 1: Algorithm locality problem and the suggested solution.

Finally, the left-to-right analysis uses the whole word or even the whole text from the current analysis point on, as the query string *s* for the dictionary lookup procedure that uses prefix search to find all hypothetical stems in one search operation. We have discussed the advantages of this method in the Section 0. In this work, we show that such search can be made in a data locality-friendly way.

Thus, the main antecedents of our work are left-to-right allomorph method of morphological analysis (Gelbukh, 1995; Hausser 1999) and the techniques of prefix search in B-trees. What is new in our task is considering data locality problem with respect to the prefix search, and what is new in our solution is the repetition of some records in the data structure, which guarantees one storage block access per word, as explained in Section 0.

2.3.1 Motivation for Spelling Correction Task

In the Section 5.2 we will illustrate the usefulness of the suggested structure on the spelling correction application, which is known to be especially computationally expensive and thus especially sensitive to performance of dictionary lookup procedure.

There is extensive literature on spelling correction and approximate string matching (Aho, 1990; Frakes & Baeza-Yates, 1992; Gusfield, 1997). The spelling-related problems can be classified into the following increasingly complex tasks (Kukich, 1992):

1. Non-word error detection (**graffe* for *giraffe*),
2. Isolated-word (or out-of-context) error correction (find *giraffe* given **graffe*), and
3. Context-dependent error detection and correction (**all there apples* for *all three apples*).

The solution to the former task can be obtained as a by-product of morphological analysis: a word that cannot be assigned any plausible morphological structure is a possible error. In this paper, we argue for that the dictionary lookup algorithms used nowadays for morphological analysis are not optimized with respect to the data locality problem; we show how the proposed data structure can improve its performance.

As to both the second and the third problem, the corresponding algorithms typically consist of two sequential steps: (1) generating all possible spelling correction candidates and (2) ranking these candidates according to some criterion (Kernighan *et al.*, 1990; Jurafsky & Martin, 2000). For example, to correct the string **acress*, first the set of correction candidates is generated: *actress*, *cress*, *caress*, *access*, *across*, *acres*; then for each candidate, some estimation of its quality is computed, and the best-scored variant is chosen. Thus, the task is broken down into two nearly independent subtasks: generation and scoring of the candidates.

Scoring is the more "linguistically-rich" task. Many strategies have been suggested for scoring. For context-dependent error correction, mostly probabilistic approaches are used, such as Bayesian inference: of all candidates the one is chosen that maximizes the probability of the surrounding N-gram (Kernighan *et al.*, 1990). Other, more linguistic coherence measures are possible, such as semantic coherence (Hirst & Budanitsky, 2003) or collocation detection (Bolshakov & Gelbukh, 2003a). In many cases, however, a shallow parser is enough to judge on which candidate results in a grammatical sentence.

For out-of-context error correction, the most popular measure is the edit distance: the minimum number of operations (out of some inventory of operations) needed to step by step transform one of the strings to another one. Sometimes the following three operations are considered: substitution, deletion, or insertion of one letter (Levenshtein, 1966); in this case, the edit distance between *table* and *cables* is 2: one substitution (*c* for *t*) and one insertion (*s*). Other possible operations can include transposition of two adjacent letters (Damerau, 1964) or more complex transpositions, see Section 5.2. To measure the edit distance between two given strings, dynamic programming approaches are used (Wagner and Fisher, 1974).

In contrast to scoring, generation of possible variants is usually less "intelligent" task. Some formal measure of string similarity is assumed, and the strings are generated that are close enough to the original string with respect to this measure. Obviously, the string similarity measure and the threshold for the closeness of the generated variants to the original string are to be coherent with the scoring policy. In this paper, we consider only the edit distance measure; thus, our approach would not be appropriate for, say, substitution of near-synonyms (**tall school* → *high school*) or paronyms (**highly sensible detector* → *highly sensitive detector*) (Bolshakov & Gelbukh, 2003b). As the inventory of edit operations we consider substitution, omission, deletion, and two kinds of transposition (see Section 5.2), though our discussion does not really depend on the specific set of edit operations.

We consider the task of generation of correction candidates, as opposed to their scoring or selection of the best variant. We restrict the set of the generated variants to those at the distance 1 from the original string. This is sufficient for correction of single-letter errors (**shool*, **schol*, **schooll*), which cover at least 80% of the errors people make (Damerau 1964). Our algorithm can be, though, easily modified to generate the variants of the distance 2, 3, etc., as well to other edit operations (e.g., **scholl*).

Our discussion of spelling correction is not aimed at proposing a new approximate string-matching algorithm *per se*. Instead, we will show how the data structure used for exact string matching within an integrated morphological analysis system can be used for fast spelling correction without any modifications that would slow down the normal (exact) operation or require additional storage space.

3 Data Structure

Hereafter we will speak of the keys or the records that have these keys interchangeably, when this does not create any confusion. For example, we will say "list of keys" implicitly supposing that attached to each key in the list is some information (value) structure of which is not important for our discussion.

We will start from a naïve approach and then describe two improvements to it. Namely, let us start from an alphabetically ordered list of keys D that appears to be quite natural data structure for the prefix search task. As we have seen in Section 0, there are two locality problems with this list.

First, even if we are interested in finding only the alphabetic place of a string s in D, binary search is not a local algorithm.

Second, if we are interested in finding the initial substrings of s, especially all of them rather than only one, then non-locality is an inherent property of such a data structure, i.e., it does not depend on the algorithm used.

In this section, we will consider these two problems and suggest the corresponding solutions. In Section 0, we will describe a simple solution to the first problem, which appears to be a kind of 2-level B-tree. In Section 0, we will describe a solution to the second problem, which is our novel contribution.

3.1 Algorithm Locality: Index of Blocks

By $a \le b$ we will denote the alphabetic order on the letter strings.

***Definition* 2**. *Alphabetic place* $D[s]$ of a string s in D is the number of the records $r \in D$ such that $r \le s$, i.e., the position of the key immediately before the place that s had if it were inserted in D.[8]

In an appropriate context, we will understand by $D[s]$ the corresponding record itself rather than its number. This should not cause confusion since it is always clear whether we mean a number or a record. Note that the record $D[s] = s$ if and only if $s \in D$.

Let us split D into segments, or blocks, B_i corresponding to the physical storage units such as disk sectors, memory pages, network packages, etc.: $D = \bigcup B_i$. Such splitting can be done sequentially, from the first to the last record. We add each record to the current block; if adding a record exceeds the block size, we leave some space in the current block unused, form the next block and make the current record first in it. The system $\{B_i\}$ can be formally defined as the only system such that:

1. Each B_i is a continuous segment of D which we will denote by analogy with geometry as $B_i = [a, b] \equiv \{r \in D \mid a \le r \le b\}$,
2. $D = \bigcup B_i$,
3. $B_i \cap B_j = \varnothing$ for $i \ne j$,
4. The total size of all records in each B_i does not exceed the required size of the block, and
5. No B_i can be extended to the right without violating this restriction on the size.

The uniqueness of such a system is obvious by induction by the block number: the first block is defined uniquely, then the second, etc.

Let us consider the task of finding the alphabetic place $D[s]$ of a given string s. Binary search is not a local algorithm. However, it can be easily improved using an index of the blocks.

Let us denote the first key of the block B_i as b_i, and the last key as e_i, so that the block is a segment $B_i = [b_i, e_i] \subseteq D$.

Let us consider a set of keys $I = \{b_i\}$, where b_i is the first key in B_i. We call this set (first-level) *index* of the blocks.

***Theorem* 1**. The alphabetic place of a string s in D is located in the block with the number $I[s]$: $D[s] \in B_{I[s]}$.

Proof: $b_{I[s]} \le s < b_{I[s]+1}$. ◊

[8] This notation is motivated by C++ programming language where an array D of strings can be indexed with a string s, which is denoted as $D[s]$.

Thus, the necessary block can be found by means of search in the index I. We consider I small enough so that the data locality problem is not applicable to it. What is more, it can be made even smaller by removing redundant letters of some keys that do not change the alphabetic place of any string. Namely, let us replace each element $b_i \in I$ by its minimal initial substring b_i' that is not an initial substring of the last key of the previous block e_{i-1}: $b_i' \prec b_i$, $b_i' \not\prec e_{i-1}$, and b_i' is the minimal with this property (see Definition 1 for the symbol $\prec$). Let $I' = \{b_i'\}$.

***Lemma* 1**. There do not exist x, y, z such that $z \prec x$, $z \not\prec y$, $z \le y$, and $y \le x$.

Proof: Suppose the first three conditions hold. Since $z \not\prec y$ then $z \ne y$ and from $z \le y$ it follows $z < y$. Then from $z \prec x$, $z \not\prec y$ by definition of the lexicographic order it follows $x < y$. ◊

***Theorem* 2**. The alphabetic place of a string s in I is the same as in I': $I[s] = I'[s]$.

Proof: Let $i = I[s]$, $j = I'[s]$. First, since $b_i \le s$ and $b_i' \prec b_i$, then $b_i' \le s$. Second, since $s \le e_i < b_{i+1}$, $b_{i+1}' \prec b_{i+1}$, and $b_{i+1}' \not\prec e_i$, then, by Lemma 1, $s < b_{i+1}'$. Thus, $b_i' \le s < b_{i+1}'$, that means that $i = I'[s] = j$. ◊

Thus, we can use an even smaller index I' to find the necessary block. First, the place of s is found in I', then the block B_i with the corresponding number is retrieved, and then s is found in B_i by any suitable method. Ignoring the problem of locality of search in a small list I', we can say that this algorithm of finding the alphabetic place of s in D is local.

In practical implementation, for speed and simplicity we implemented the search for s in $I' = I_1$ by using a second level index: we split I_1 into blocks and compiled an index I_2 of their first keys. We applied this procedure recursively, splitting that second level index into blocks and compiling a third level index I_3, which in our case consisted in only one block and thus did not need in any its own index.

3.2 Data Locality: Block Prefix

Though, as we have shown in Section 0, the problem of finding $D[s]$ is local, the problem of finding an $x \in D$, $x \prec s$ is not. As it was discussed in Section 0, first, x can be located far from $D[s]$; second, if we are interested in finding all such keys, they can be located quite far from each other, as we have seen with the keys *co-*, *con-*, *const-*, etc. Such keys are probably located in different blocks B_i.

What we propose is to add to each block all keys that can be necessary for any query addressing this block. As we will show, the changes will result only in appending some small amount of information at the beginning of the block. We call this additional piece of information a *block prefix*.

Namely, for any s such that $D[s] \in B$, for any $x \in D$ such that $x \prec s$, we propose to duplicate the key x with its corresponding record to the block B. Probably some records of D will be moved to other blocks since the size of the block is fixed; thus this operation can cause changing the way D is split into blocks and increase the number of blocks. However, with this operation we reach our goal: by retrieval of only one block B_i, namely, such that $D[s] \in B_i$, we find in it, locally, all such keys $x \in D$ that $x \prec s$.

Does this cause a significant increase of the dictionary size? Does this result in complicated algorithms of search and of dictionary building? Let us examine closer the set of keys x_j that are to be added to B and the way D is now split into $\{B_i\}$.

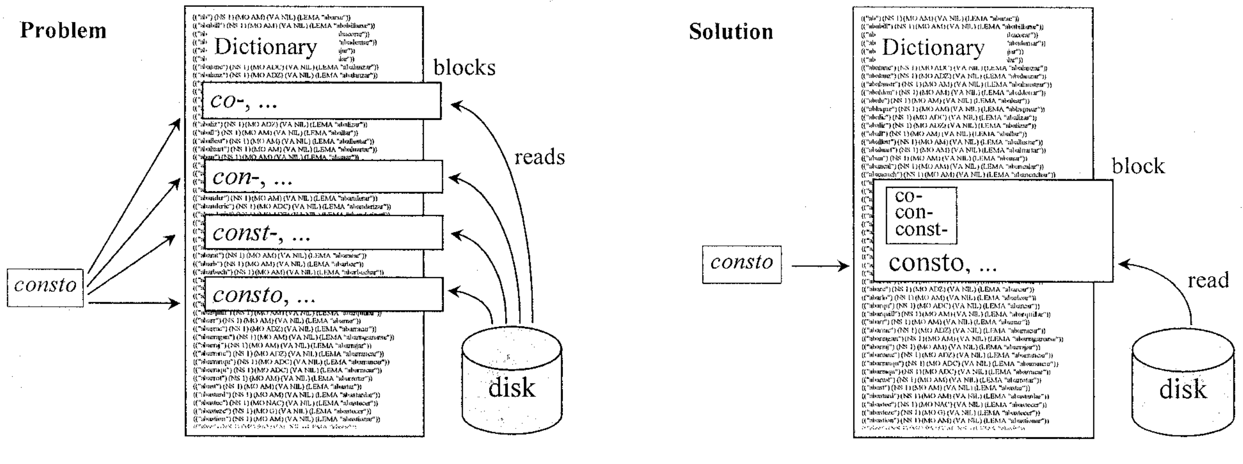


Figure 2: Data locality problem and the suggested solution.

Definition 3. A set of strings $S \subseteq D$ is *closed* if $\forall x \in S$ and $\forall y \in D$, if $y \prec x$ then $y \in S$.

Definition 4. The closure $\bar{S}$ of a set $S \subseteq D$ is the minimal closed subset of D that contains S.

Lemma 2. Definition 4 is correct, i.e., there exists exactly one minimal closed subset of D that contains S, namely, $\bar{S} = \{y \in D \mid \exists x \in S \text{ such that } y \prec x\}$.

Proof: Obviously, this set is closed and contains S, since for any x, $x \prec x$. On the other hand, it is contained in any other set with these two properties.

Now we can formalize our solution. Let us split D into a set of segments B_i (which are not necessary the same B_i that we discussed before), such that each closure $\bar{B}_i$, rather than the block B_i itself, has the size corresponding the physical storage unit. All the statements of the previous sections hold for this segmentation of D.

Let this segmentation of D be minimal, i.e., splits D into a minimal possible number of blocks. We propose to use $\{\bar{B}_i\}$ as our main data structure (instead of the original set of blocks $\{B_i\}$). Now we will show that this structure is only insignificantly larger than the original set D: $\Sigma \mid \bar{B}_i \mid \approx |D|$.

Lemma 3. For any s and any $x \in D$, if $x \prec s$, then $x \prec D[s]$.

Proof: By definition of $D[s]$, if $x \in D$ and $x \le s$, then $x \le D[s]$. If $x = D[s]$, then $x \prec D[s]$, else, by Lemma 1, $x \prec D[s]$. ◊

Lemma 4. For any block $B \subseteq D$ and any $x \in D$, the following conditions are equivalent:

- $\exists s$ such that $D[s] \in B$ and $x \prec s$,
- $\exists a \in B$ such that $x \prec a$.

Proof: By Lemma 3. ◊

Thus, the set of keys x_j to be added to a block B is exactly the set of all initial substrings $x \in D$, $x \prec a$ of all keys $a \in B$. What is more, the keys to be actually added are just initial substrings of the very first key $b \in B$ since the other ones are already in B. By $b < a$ we denote the alphabetical order.

Lemma 5. Let b be the first key of a block $B \subseteq D$, $x \in D$, $x \notin B$. Then if $\exists a \in B$ such that $x \prec a$, then $x \prec b$.

Proof: Since $x \in D$, $x \le a \in B$, and $x \notin B$, then $x < b$. Since $b \le a$ and $x \prec a$, then, by Lemma 1, $x \prec b$. ◊

Thus, finally, the only keys to be duplicated into the block in order to guarantee access locality are the initial substrings of the very first record of the block:

Theorem 3. For any segment $S = [b, e] \subseteq D$, $\bar{S} = S \cup \{y \in D \mid y < b,\ y \prec b\}$.

Proof: By Lemma 5. ◊

Thus, each closed block $\bar{B}_i$ is obtained from the correspondent block B_i by adding before its first record all records from D whose keys are initial substrings of only one, namely the first one, key of B_i. Clearly, there are not many such records in D.

For example, for a block of Spanish dictionary starting with the stem *constat-* for *constatar*, only the records with the keys *const-*, *con-* and *co-* are duplicated into the block, no matter how large it is.

We carried out the experiments with the AMORE morphological analysis and synthesis system we have developed (Gelbukh 1995). In the dictionary, together with each stem, the morphological information of the size about 20 bytes on average was stored. The blocks were of 1 kilobyte and contained about 30 records each on average.

In our experiments, there were about 2 to 3 records in average added to each block, so the redundancy was about 10%, and would be proportionally less if we used larger blocks, say, of 4 kilobytes.

With such a small redundancy, we guarantee the complete locality of data: any initial substring query is processed with retrieving of exactly one block of the main data storage.

4 Operations: Searching, Building, and Exporting

To prove the practical usefulness of the proposed data structure, we provide the algorithms of its building, exporting, and searching. We do not provide, however, any fast algorithm of insertion and deletion, since these operations are rather irrelevant for linguistic dictionaries, which are not updated very frequently, though they are relevant for databases (Johnson & Shasha, 1993).

4.1 Search

We have already discussed the idea of the search algorithm in the previous sections; here we formalize it. Given a string s and a dictionary (database) D, our task is to find all $x \in D$ such that $x \prec s$ (Task 3 in Section 0).

Suppose we have an algorithm A to find the alphabetic place $\bar{B}_i$ [s] in a given block, see Section 0 below. Then, we start from the highest level of index I_n. With A, we find the number $I_n[s]$, and retrieve the block with this number from I_{n-1}. Then we repeat the search in this block, extract the corresponding block from I_{n-2}, etc., until I_1, and finally retrieve from the main storage the block $\bar{B}_i$ that contains D [s] and thus all such $x \in D$ that $x \prec s$.

The description of the algorithm A is not essential for the present paper. It does not affect the issue of locality since all its memory accesses are within one block (physical memory page). However, we will discuss it briefly because of its importance for the practical implementation of a system based on the proposed data structure. The algorithm depends on the internal representation and possible compression of data in the block.

The following Sections 0 and 0 are not essential for understanding the main topic of the paper.

4.1.1 Search in a Compressed Block

Data compression is important for our algorithms because of the fixed size of the block. Really, as we have seen in the Section 0, additional overload to each block—the block prefix size—does not depend on the block size. Thus, the percentage of redundancy depends on the size of block. If the size of block prefix is comparable with the total block size, our algorithms are rendered unaffordable.

In our blocks, data is ordered alphabetically, so that the keys with common initial substring are located together. To compress such data, Cooper compression works well (Cooper, 1958): instead of an initial substring equal to that of the immediately previous key, only the number of common letters is indicated. For example, the keys in Table 1 are compressed in the following way: 0/*clar*-, 1/*o*-, 2/*m*-, 2/*n*-, 3/*centr*-, 3/*st*-, 5/*ancia*-, 7/*te*-, 6/*t*-, 5/*elación*-, 5/*ipad*-, 5/*ru*-, 7/*cción*-, 8/*tiv*-, 11/*ismo*-, 4/*ult*-. The first key in the block always has the Cooper coefficient 0.

Such compression is especially useful for morphological dictionaries since in many languages there are variants of stems for the same word differing only in the last few letters. English examples are *lady – ladi-es, stop – stopp-ed,* Spanish *traduc-ir – traduj-o – traduzc-a; indic-ar – indiqu-en*; *alcanz-ar – alcanc-en*, Russian *prevozmo-ch' – prevozmog-u – prevozmozh-esh'*[9] (Zaliznyak, 1987).

Here we will not give any detailed comments on the search algorithm but only a short description; we again ignore the complications caused by the records with equal keys. The main personage of the algorithm is the common initial substring length *e* with the meaning of the length of the maximal common initial substring of the query string *s* and the current key $a_i \in \overline{B}$ being processed.

The keys a_i are examined one by one. Initially, $i = 0$ and $c = 0$. Let the Cooper coefficient of the current key a_i be *c*. If $c > e$, the algorithm just proceeds to the next key a_{i+1}. Otherwise, starting from the position *c*, the string *s* is compared with a_i, and the new value for *e* is determined as the first position differing the two strings. During this comparison, two special situations can be found. If $s < a_i$ or the block is exhausted, the search process is over. If $a_i \prec s$, one of the keys we are looking for is found. It is added to the set of the search results. In our implementation, the search results are stored in a LIFO stack, and each found key is pushed onto this stack. After the search is over, the results can be one by one popped out of the stack and examined; with this, they are examined in the order natural for most applications, i.e., from the longest to the shortest ones.

Since the block $\overline{B}$ contains the prefix with duplicated records, all the necessary records are found in this only one block.

[9] 'To overcome,' 'I will overcome,' 'you will overcome.'

4.1.2 Substring Chains

The search method described above looks through all the records in the block until the alphabet place of the query string *s* is found. However, there are algorithms jumping directly to this position $\overline{B}\,[s]$, for example, binary search. By storing some additional information in the dictionary, we can find all the initial substrings of *s* starting directly from $\overline{B}\,[s]$. For this, let us suppose that with each record in $a \in \overline{B}$, a pointer to, or just the length of, the maximal $a' \in \overline{B}$ such that $a' \prec a$ is kept in the dictionary. Let $C\,(a) = \{\ldots \prec a'' \prec a' \prec a\} \subseteq \overline{B}$ be a chain of such maximal substrings.

Lemma 6. $\forall s$ and $\forall x \in \overline{B}$, if $x \prec s$, then $x \in C\,(\overline{B}\,[s])$.

Proof: By Lemma 3. ◊

Thus, to find all the initial substrings of *s*, the chain $C\,(\overline{B}\,[s])$ is to be looked through starting from $\overline{B}\,[s]$ until an element $x \in C\,(\overline{B}\,[s])$ found such that $x \prec s$. Then the chain $C\,(x)$ is the result of the query. For most applications, this chain is to be examined starting from *x*.

4.2 Building

In spite of simplicity of the formula for the duplicated records, its application sounds like vice circle. Really, the duplicated records depend on the first records of the blocks, while the block layout itself depends on such duplication, since now it is the size of $\overline{B}_i$ that is fixed, not that of B_i. However, we propose a simple algorithm for building such a data structure given an alphabetically ordered list of records *D*. Note that the algorithm builds the complete structure rather than updates individual elements in it.

Our task is to build the structure in the block format discussed above, i.e., to split *D* into a minimal set of blocks B_i so that $D = \bigcup B_i$ and for each block, the size of its closure $|\overline{B}_i| \le C$, where *C* is a constant, say, 4 kilobytes.

Let us first consider an auxiliary data structure that we call a stack of initial substrings. This is not exactly a LIFO stack, but is similar to it. The stack of initial substrings *S* has an underlying LIFO stack S' of records of *D*, and implements one operation, a kind of push, with the following behavior; see Fig. 3. When a record with the key *s* is pushed onto *S*, then the records are popped out of S' until the record with a key $s' \prec s$ is found on the top of S' or S' is empty. Then, the new record is pushed onto S'.

Let the records of the ordered list *D* are read one by one from *D* and pushed onto *S*. At the beginning of the process, *S* is empty.

```
Until S is empty or S.top ≺ s
    Pop it from S.
Push s onto S.
```

Figure 3: Updating stack *S* with record ***s***

```
Create an empty block B.
For each input record s do:
    If there is not enough space in B to add s then
        Output B.
        Make B empty.
        Add stack contents to B.
    Add s to B.
    Update the stack S with s.
Output B if contains any new records.
```

Figure 4: Dictionary formation algorithm

Let the records of the ordered list *D* are read one by one from *D* and pushed onto *S*. At the beginning of the process, *S* is empty.

Theorem **4.** At each step of this process, after $s \in D$ is processed, *S* is ordered alphabetically from bottom to top and contains exactly all records whose keys are initial substrings of s: $S = \{ x \in D \mid x \prec s \}$.

Proof: First, since the new records are pushed onto S' (after some pop operations), then for each record, only the records that came before it are deeper than it in S'. Since the records come in alphabetical order, S' is ordered alphabetically. Second, when a record is pushed onto S', the key immediately below it is its initial substring; then, S' contains a chain of initial substrings; in particular, $\forall x \in S' \Rightarrow x \prec s$. On the other hand, if $x \in D$ and $x \prec s$, then $x \leq s$, so that *x* has come at some step and was pushed onto S'. However, at no step was it popped out of S'. Really, if it were popped out by some record *y* that came after *x*, then $x \leq y$, $x \not\prec y$, $y \leq s$, and $x \prec s$, which is impossible by Lemma 1. ◊

The algorithm of formation of the dictionary works as follows, see Figure 4.

- At the beginning of the process, an empty block $\overline{B}_1$ is created.
- During the work cycle, the records $s \in D$ are read one by one and pushed onto *S*. After that, an attempt is made to append *s* to the block $\overline{B}_i$ being currently formed. If there is any block compression as discussed below, it is performed. The new potential size of the block is estimated. If still $|\overline{B}_i| \leq C$, then the process repeats with the next record of *D*.
- However, if with the new record the size $|\overline{B}_i|$ would exceed *C*, then the block $\overline{B}_i$, without the new record *s*, is ready. It is appended to the dictionary structure $\{\overline{B}_k\}$ being formed, and a new empty block $\overline{B}_{i+1}$ is created. The contents of the stack *S* is copied to $\overline{B}_{i+1}$ from bottom to top, thus making the block closed. With this, *s*, which is currently on the top of *S*, becomes the first record of the new "logical" (not closed) block B_{i+1}, and $\overline{B}_{i+1}$ contains all the records whose keys are initial substrings of *s* and is ordered alphabetically. After that, the process repeats with the next record of *D*.

In our implementation, at the moment of formation of the new block B_i, we dump the key b_i' of its first record to compile later the index I'. The key is truncated according to the definition of I' (see Theorem 2) since the last record of the previous block e_{i-1} is known at the moment of formation of b_i.

Also, with each block we kept a header containing only two numbers: (1) the total number of the records in the block and (2) the size of the block prefix, i.e., the number of the redundant records duplicated into the block, which is $|S|-1$ at the moment of creation of the new block B_{i+1}.

We did not mention here a special treatment for the first and last blocks. Another complication is the special case of records with equal keys. This case can be ignored if we prohibit equal keys in *D* merging the values of such records when necessary. Alternatively, special precautions are to be taken for a group of records with equal keys not to be split across block boundary.

The described algorithm is a single-pass one.

4.3 Exporting and Updating

Now let us consider the inverse task: given the block structure $\{\overline{B}_k\}$, restore the ordered list of records *D*. This task is rather trivial. In our implementation, each block $\overline{B}_i$ contains a header with the number r_i of the redundant records in it, so that it is enough to dump out the contents of each block, from first to last record, ignoring the first r_i records of each block. Even if the number r_i were not kept with the block, it would be easily restored by looking for the first record $b \in \overline{B}_i$ such that $a > e$ where *e* is the last record of $\overline{B}_{i-1}$. This procedure is also a one-pass algorithm.

As we have pointed out, in this paper we do not discuss here any fast algorithm of updating the block dictionary structure by adding, deleting, or changing a few records. In the context of database indices management, there is much literature on updating similar structures such as B-Trees and their variants (Gusfield, 1997). However, unlike the indices of the databases, the morphological dictionaries do not change frequently, so that we do not need to develop efficient algorithms for updating the block structure.

The simplest way of updating the structure is to export it into a sorted list *D*, then merge in, remove, or change some records, and then create a new block structure out of the updated list. Since both the exporting and building the structure. Then adding new records will result in reconstructing only a limited number of blocks. The possible need in duplicating the new records into some blocks complicates things, though in the case of minor changes, the number of changing blocks still remains much less than the total size of the dictionary. Here we do not discuss this algorithm; see (Gelbukh, 1995).

5 Applications: Morphological Decomposition and Spelling Correction. Approximate Prefix Search

Decomposition and error correction—or approximate matching—tasks arise in nearly any search application of the type we consider. However, our main motivation here is again natural language analysis. The close relatedness of spelling correction in inflexion languages with morphological analysis and thus in the suffix and prefix search was realized long ago (Bolshakov, 1991).

We will describe our algorithm of approximate prefix search in tight integration with the decomposition algorithm. This gives the solution to the task approximate decomposition, i.e., approximate matching with a string that can be decomposed by the given set of dictionaries.

The role of this section is mostly illustrative since it shows how our structure can be used for typical tasks of string processing. Because of the marginal role of this section and relative complexity of the algorithms, the description of the latter will be quite sketchy.

5.1 Decomposition of a String

Let us consider several string lists D_i. By decomposition of a given string s, we mean finding such a set of substrings $x_i \in D_i$ that their concatenation is s: $s = x_1 \dots x_n$. If this can be done in several different ways, all such sets are to be found. Such decomposition is the main operation in morphological analysis of natural languages: Spanish *habl-ába-mos*, German *kommunikation-s-technik*. The specifics of this task for natural languages is that at least one (and usually exactly one) list of substrings D_i is very large—namely, the stem dictionary.

We view the decomposition task as sequential application of the substring search algorithm discussed above. First, initial substrings of s are found in D_1. For each of them, initial substrings of the rest of s are looked up in D_2, etc. After the last dictionary is analyzed, only the variants of analysis are accepted covering the entire string s. Actually, in natural language analysis, the latter condition is slightly more complicated: the variants of analysis are accepted for which after the last piece, a symbol goes in the text that can be a word delimiter, such as a comma or period.

This method—analyzing the rest of the string s for each initial substring found in $D_1, \dots, D_i$—in effect implements a kind of backtracking.

5.2 Approximate Prefix Search. Spelling Correction

By error correction, we mean solving the decomposition task not for the given string s but for some another string s' that differ from s in a specific way. To put it in other words, the task is to find such strings s' that differ from s in a specific way and that can be decomposed by the given set of dictionaries D_i.

Our algorithm does not heavily depend on the string similarity criterion being used. In our implementation of a morphological analyzer, we used so-called single letter error (called also single-error misspelling by Damerau (1964)), which is defined as either substitution of one letter, omission of one letter, insertion of one letter, or swapping two neighboring letters. In addition, we took into account some other types of errors such as transposition of two vowels around one consonant or two consonants around one vowel, as in **com<u>t</u>u<u>p</u>ational*.

In the rest of the paper, we will rely on a possibility for any given string s, position p, and letter l to generate all strings similar to s according to the criterion being used, such that all of them have the letter l on the position p and coincide with s by the initial $p-1$ letters. For example, for a string **comtupational*, the letter "*p*," position 4, and the criterion mentioned above, the following strings are generated: **compupational*, **comptupational*, and *computational* (the latter one by swapping two consonants around one vowel).

5.2.1 Minimization of the Number of the Hypotheses Tried

Since we consider the similarity criterion as a generating procedure, our algorithm will form and try some hypotheses for the variants of the string s. There are several possible ways to measure the performance of an error correction algorithm. Let us consider the following task: to find all possible variants of the given string s that can be decomposed by the given set of dictionaries D_i trying as few hypotheses as possible. We will divide the sketch of our algorithm here in three parts, starting from a simple case and then adding more complications.

5.2.1.1 One Dictionary: Approximate Search

First, let us consider a simpler task: to find exactly the variants of s in only one dictionary D. Our algorithm takes advantage of the possibility for alphabetic place $D[s']$ of a variant string s' to predict the following position and letter to be tried.

We try the positions p of the string s one by one, starting from $p = 1$, see Fig. 5. In the exhaustive search, at each position p the letters from a to z would be tried to form the hypotheses s' by our generative similarity criterion. However, in our algorithm not all letters have to be really tried. For a hypothesis s', let us consider $a = D[s'] + 1$: the key following the alphabetic place of the current

```
**Let** *limit* = 1 + length of maximal common initial substring of $s$ and $D[s] + 1$.
**For each** position $p$ starting from 1 up to *limit* do:
    **Set** *letter* to "$a$".
    **Repeat** while *letter* is less or equal to "$z$":
        Form a set of hypotheses $s'$ that have the *letter* at the position $p$.
        Order this set alphabetically.
        **For each** hypothesis do
            **If** $D[s'] = s'$ **then** report a possible variant.
            **If** this is the last hypothesis **then**
                **Let** $a' = D[s'] + 1$.
                **If** $a'(1, p-1) \neq s'(1, p-1)$ **then** break the loop and go to next $p$.
                **If** $a'(p) \neq s'(p)$ **then set** *letter* to $a'(p)$, **else** increase *letter* by 1.
```

Figure 5: Simplified algorithm of error correction.

position p the letters from a to z would be tried to form the hypotheses s' by our generative similarity criterion. However, in our algorithm not all letters have to be really tried. For a hypothesis s', let us consider $a = D[s'] + 1$: the key following the alphabetic place of the current hypothesis.[10] Let us denote by $w(u, v)$ the substring of a string w starting at the position u and ending at v, and $w(u)$ the letter in w at the position u. We will count the positions from 1, so that $w(1, v)$ is the initial substring of w with the length v.

***Lemma* 7a.** If $a(1, p-1) = s'(1, p-1)$ and $a(p) \neq s'(p)$, then there is no such $x \in D$ that $x(1, p-1) = s'(1, p-1)$ and $s'(p) < x(p) < a(p)$.

Proof: By definition of alphabetic order, for the hypothetical $x \in D$, $s' < x < a = D[s'] + 1$, i.e., $s' < x \leq D[s']$, that implies $s' < D[s']$, which is impossible by the definition of $D[s']$. ◊

Thus, in the case of $a(1, p-1) = s'(1, p-1)$, there is no reason to try all the letters until $a(p)$, so the next set of hypotheses s'' is formed with $s''(p) = a(p)$. This significantly decreases the number of hypotheses to be tried.

***Lemma* 7b.** If $a(1, p-1) \neq s'(1, p-1)$, then there is no such $x \in D$ that $x(1, p-1) = s'(1, p-1)$ and $s'(p) < x(p)$.

Proof: The same as for *Lemma* 7a, since $s' < a$. ◊

In this case, the entire rest of the alphabet can be skipped. Finally, if $a(1, p) = s'(1, p)$, there is no information to skip any letters. For best results, the hypotheses generated by the similarity mechanism for a particular position and letter are to be tried in alphabetical order. In this case, the hypothesis that satisfies the conditions of the two lemmas is with a greater probability the last one; this simplifies the algorithm.

When the process stops? So far, we considered s unlimited, so there is no last position in it. This problem is easy to solve. The last position p to try is one plus the length of the maximal initial part common for s and $D[s] + 1$. By *Lemma* 7a, there are no strings in D having a larger initial part in common with s.

5.2.1.2 Many Dictionaries: Approximate Decomposition

The described procedure works with only one dictionary. Now we can return to the case of decomposition of a string by several dictionaries $D_1 \ldots D_n$. We will not discuss the algorithm in detail, but the idea is as follows. If in a dictionary D_i, the next key a is an initial substring of s' and therefore the conditions of no lemma above are satisfied, then the decomposition algorithm normally proceeds to the next dictionary D_{i+1}, etc., until some difference between $a_k = D_k[s'] + 1$ and the correspondent rest of s' is found. The position p is considered relative to s' rather than to a_k; the rest of our algorithm remains the same. Effectively, a Cartesian product $\prod D_i$ is considered as a single dictionary D, and the algorithm described above is applied to it.

5.2.1.3 Block Boundaries

Finally, let us discuss one more complication. As we have proposed in a previous section, for locality of access, each of the dictionaries D_i is stored in the form of blocks $\vec{B}_m$, such that only one such block is fetched from the main storage device when the dictionary is searched. What to do then if the next key $a_k = D_k[s'] + 1$ is not located in this block, but in the next block $\vec{B}_{m+1}$? Fetching that next block is not desirable. Fortunately, it is not necessary.

Namely, the necessary information can be found in the index I', or, more precisely, in the index I'_t of some level t. Really, if $D[s']$ is located at the boundary of the blocks,

[10] To avoid too complicated notation, we use interchangeably the index of a string in D and the string itself when this does not cause any confusion.

```
Set string $a = D[s] + 1$.
Set number start = 1 + length of the maximal common initial substring of $s$ and $a$.
For each position $p$ starting from start down to 1 do:
    Set letter to $a(p)$.
    Repeat while letter $\neq s(p)$:
        Form a set of hypotheses $s'$ that have the letter at the position $p$.
        Order this set alphabetically.
        For each hypothesis do:
            If $D[s'] = s'$ then report a possible variant.
            If this is the last hypothesis then
                Set $a' = D[s'] + 1$.
                If $a'(1, p-1) \neq s'(1, p-1)$ then reset letter to "a".
                If $a'(p) \neq s'(p)$ then set letter to $a'(p)$, else increase letter by 1.
```

Figure 6: Improved algorithm of error correction.

then the initial part of $I_1'[s'] + 1$ of just the necessary length[11] is equal to that of $D[s'] + 1$. If, in its turn, $I_1'[s']$ is located at the boundary of the blocks in I_1', then $I_2'[s'] + 1$ can be used, etc. until the last level index that has no block boundaries. Note that neither **Figure 5** nor **Figure 6** below reflect the complications just discussed.

Thus, we have discussed an algorithm of decomposition of a string by a set of dictionaries with error correction. The number of the hypotheses tried is substantially less than exhaustive search. For each hypothesis, only one block is fetched from the very first dictionary, which in natural language analysis is the very large stem dictionary). For other dictionaries $D_2 \dots D_n$, only one block is accessed at each corresponding step of backtracking in the decomposition process.

5.2.2 Minimization of the Median Number of Storage Accesses

Now let us discuss a different parameter being minimized. Suppose we have a procedure (or just a human user) that, for each hypothesis, can decide whether a corrected string s' for the string s is to be accepted and the search process is to be stopped. We will minimize the time of the work of the algorithm until it finds the correct variant.

We do not make any assumptions about the nature of the mechanism that causes the errors in the strings s and thus about the nature of the procedure that tests the variants. Instead, we will just organize the search in such a way that most of the variants are found at the very early stage of its work, and the most of the work that is scarcely result in new variants is done at the later stage.

For a mathematical measure of grouping the variants, we chose the median: the time when half of the total number of variants is generated. For example, let first 10 variants of error correction in s are generated in the first second and then other 10 variants in 5 minutes. Then the total time of work of the algorithm is 301 seconds, average is 15.05 seconds, but the median time is 1 second. This is the behavior that we want, in comparison with an algorithm that produces a variant each 15.05 seconds; the median time of such an algorithm being 150.5 seconds. In fact, the algorithm described in the previous section has the latter type of median time.

In accordance with the data locality principle, we suppose that the most time-consuming operation is fetching a block from the dictionary. Thus, what we want to minimize is the number of blocks fetched until the correct variant is found. For this, we should try to first examine the block in which the most variants are located. Fortunately, with our dictionary structure, most of them are located in the block $\bar{B}$ containing $D[s]$. What is more, this block probably is already fetched into memory at the moment of error correction, because to find that error correction is necessary, first the string s was searched for (and not found) in D.

Only a minor correction to the algorithm described in the previous section is necessary, see **Figure 6**. To look in $\bar{B}$ first, we start trying the hypotheses not from the first position $p = 1$, but instead from the last position in s, and advance in decreasing order. More precisely, since we consider s unlimited, really the first position p to try is the first position that is different in s and the string at the position $D[s] + 1$ in D, as it was discussed in the previous section (there it was the last position to try).

For even better locality, the letters in each position p are tried not in the order from a to z, but from $s(p) + 1$ to z and then from a to $s(p) - 1$. This increases a little bit the chances to find the correct variant in the same block where $D[s]$ is located.

With this modification of the algorithm described in the previous section, practically without any increase of the number of the hypotheses tried, the median number of the blocks accessed during the search decreases dramatically.

[11] This can be easily proved by analysis of the definition of the indices I_i'.

such as English and Spanish, e.g., in MediaLingva's Multilex system (Multilex, 1996).

Russian is a highly inflective language with multiple-morpheme word structure and many fusion effects (internal sandhi) discussed above, such as *prevozmo-ch'* – *prevozmog-u* – *prevozmozh-esh'*.[12] Our dictionary consisted of 100 thousand lexemes, which resulted in about 200 thousand dictionary entries (records). Each record consisted of a compressed stem and the corresponding morphological information. In decomposition, we used one general stem list and up to four (for verbs) short lists of endings. The main stem list was indexed in three levels of I_k, as described in Section 0.

A typical result of error correction is as follows: for the mutilated form **prevosmozhesh'*, the complete search process took 5 seconds, while the only variant *prevozmozhesh'* was found in 0.05 second.

Conclusions

A data structure that guarantees one memory block access per prefix search query with a very large dictionary was introduced.

The following tasks have been discussed:

1. Prefix search in a very large dictionary: finding all records in the database whose keys are initial substrings of a given unlimited string *s*,
2. Given the unlimited text string *s*, separating its leftmost "word" *w* that is defined as an initial substring of *s* that can be decomposed into concatenation of substrings $w = x_1 \dots x_n$, $x_i \in D_i$ for a set of dictionaries $D_1 \dots D_n$.
3. Actually decomposing *w* into concatenation of elements (allomorphs) belonging to $D_1 \dots D_n$
4. Approximate search and decomposition: solving the above tasks approximately, for a given criterion of similarity between strings.

These tasks constitute the main part of the morphological analysis and spelling correction algorithms. Since with our algorithm, the leftmost word (stem) is found automatically, a stream of letters without boundaries between words, like Japanese text or DNA structure, can be thus decomposed into words.

Our variants of alphabetic search and spelling correction algorithms are optimized for data locality requirement that arises when data are accessed in chunks (blocks: disk sectors, memory pages, or network packages) and the cost (time) of data access is proportional to the number of chunks accessed rather than number of bytes used. In particular, this is the situation with virtual memory under the widely used operating systems such as Windows and UNIX.

The structure of dictionary was discussed, which is similar to a 2-level B-tree. At the cost of slight redundancy of storage, it was guaranteed that only one block of dictionary is fetched per query, in spite of the inherent unlocality of the task.

As an illustration of practical usefulness of the proposed structure, algorithms of its building, exporting, searching, and searching with spelling correction were sketched.

The mentioned duplication of a very small percentage of records for sake of guaranteed one memory block access per prefix search query is a novel technique not previously discussed in the theory of B-trees.

[12] We do not provide any glosses since the meaning of the words is irrelevant for our discussion.

References

Aho, Alfred V. "Algorithms for finding patterns in strings", J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 5, pp. 254-300. Elsevier Science Publishers B. V., 1990.

Bayer, R., and **K. Unterauer**. "Prefix B-Trees", *ACM Trans. Database Systems* 2., p. 11-26, 1977.

Bolshakov, I. A. "Automatic Error Correction in Inflected Languages." *Journal of Soviet Mathematics* 56 (1): 2263–2287, 1991.

Bolshakov, I. A., and **A. Gelbukh.** "On Detection of Malapropisms by Multistage Collocation Testing", *NLDB-2003, 8th International Workshop on Applications of Natural Language to Information Systems, Lecture Notes in Computer Science*, 2003, to appear.

Bolshakov, I. A., and **A. Gelbukh.** "Paronyms for Accelerated Correction of Semantic Errors", *KDS-2003, Knowledge-Dialogue-Solution*, Varna Bulgaria, 2003, to appear.

Cassidy, P. "An Investigation of the Semantic Relations in the Roget's Thesaurus: Preliminary Results", A. Gelbukh (ed.), *Computational Linguistics and Intelligent Text Processing*, Fondo de Cultura Económica, Mexico, to appear in 2001. See also *Proc. of CICLing-2000*, February 13 to 19, 2000, CIC-IPN, Mexico City, ISBN 970-18-4206-5.

Comer, Douglas. "The Ubiquitous B-Tree", *Computing Surveys* 11 (2), 1979, pp. 121–137.

Cooper, W. S. "The storage problem", *Mech. Translat.*, 1958, pp. 74–83.

Damerau, F. J. "A technique for computer detection and correction of spelling errors", *Communications of the ACM*, 7(3), 1964, pp. 171–176.

Diccionario. *Diccionario de la lengua española*. Real academia española, vigésima primera edición, 1992.

Damerau, F. J. "A technique for computer detection and correction of spelling errors", *Communications of the ACM*, 7(3), 1964, pp. 171–176.

Diccionario. *Diccionario de la lengua española.* Real academia española, vigésima primera edición, 1992.

Fellbaum, Ch. (ed.) *WordNet as Electronic Lexical Database*. MIT Press, 1998.

Frakes, W., and **R. Baeza-Yates**, editors. *Information Retrieval: Data Structures and Algorithms.* Prentice-Hall, 1992.

Gelbukh, A. F. *An efficiently implementable model of morphology of an inflective language.* Ph.D. thesis, VINITI, Moscow, Russia, 1995; see http://www.Gelbukh.com.

Gel'bukh, A. F. "Effective implementation of morphology model for an inflectional natural language", *Automatic Documentation and Mathematical Linguistics*, Allerton Press, vol. 26, N 1, 1992, pp. 22-31; see http://www.Gelbukh.com

Gel'bukh, A. F. "Minimizing the number of memory accesses in dictionary morphologic analysis", *Automatic Documentation and Mathematical Linguistics*, Allerton Press, vol. 25, N 3, 1991, pp. 40-45. see http://www.Gelbukh.com

Gusfield, Dan. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology.* Cambridge University Press, 1997.

Hausser, Ronald. *Foundations of Computational Linguistics. Man-Machine Communication in Natural Language*. Springer-Verlag, 1999.

Hirst, G., A. Budanitsky. "Correcting Real-Word Spelling Errors by Restoring Lexical Cohesion". *Computational Linguistics* (to appear), 2003.

Johnson, Theodore, and **Dennis Shasha**. "B-Trees with Inserts and Deletes: Why Free-at-Empty Is Better Than Merge-at-Half", *JCSS* 47 (1): 45-76 (1993). See other publications at http://www.informatik.uni-trier.de/~ley/db/access/btree.html.

Jurafsky, Daniel, and **James H. Martin**. *Speech and Language Processing*, Prentice-Hall, 2000, 934 pp.

Kernighan, M. D., K. W. Church, W. A. Gale. "A spelling correction program based on a noisy channel model" *COLING-90*, Helsinki, Vol. II, 1990, pp. 205–211.

Knuth, Donald. *The Art of Computer Programming: Sorting and Searching*, Vol 3, 2nd Ed, Addison-Wesley, 1998.

Koskenniemi, Kimmo. *Two-level Morphology: A General Computational Model for Word-Form Recognition and Production*, University of Helsinki, Department of General Linguistics, Publications, N 11, 1983, 160 pp.

Kukich, K. "Techniques for automatically correcting words in texts", ACM Computing Surveys, 24(4), 1992, pp. 377–439.

Lenat, D. B. and **R. V. Guha**. *Building Large Knowledge Based Systems*. Reading, Massachusetts: Addison Wesley, 1990. See also more recent publications on CYC project, http://www.cyc.com.

Levenshtein, V. I.. "Binary codes capable of correcting deletions, insertions, and reversals" *Cybernetics and Control Theory*, 10(8), 1966, pp. 707–710. (Originally published in Doklady Academii Nauk SSSR 163(4), 1965, pp. 845–848.)

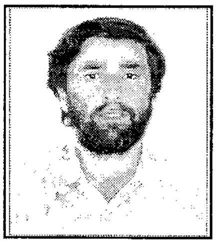
Multilex. *Electronic dictionary family Multilex*, ver. 1, 1996. See http://www.multilex.ru.

Sidorov, G. O. "Lemmatization in automatized system for compilation of personal style dictionaries of literature writers." In *Word of Dostoyevsky*, Russian Academy of Sciences, Moscow, 1996. pp. 266–300.

Wagner, R. A., and **M. J. Fisher.** "The string-to-string correction problem". *Journal of the Association for Computing Machinery*, 21, 1974. pp. 168–173.

Yuret, Deniz. *Discovery of linguistic relations using lexical attraction.* Ph.D. thesis, MIT, 1998. See http://xxx.lanl.gov/abs/cmp-lg/9805009.

Zaliznyak, A. A. *Grammatical dictionary of Russian. Word formation* (in Russian). Russkij Jazyk, Moscow, Russia, 1987, 878 pp.



Alexander Gelbukh, *was born in Moscow, Russia, in 1962. He obtained his M.Sc. degree in Mathematics in 1990 from the department of Mechanics and Mathematics of the Moscow State "Lomonossov" University, Russia, and his Ph.D. degree in Computer Science in 1995 from the All-Russian Institute of the Scientific and Technical Information (VINITI), Russia. Since 1997 he is the head of the Natural Language and Text Processing Laboratory of the Computing Research Center, National Polytechnic Institute, Mexico City. He is member of Mexican Academy of Sciences since 2000, of National System of Researchers of Mexico since 1998, lecturer of ACM Distinguished Lecturship Program for Mexico and Central America since 2000; author, co-author, or editor of about 200 publications on computational linguistics; the founder and organizer of CICLing Computational Linguistics conference series, see www.CICLing.org. For more information, see www.Gelbukh.com or www.cic.ipn.mx/~gelbukh.*