Fault-tolerant Fulltext Information Retrieval in Digital Multilingual Encyclopedias with Weighted Pattern Morphing

Wolfram M. Esser

Chair of Computer Science II, University of Würzburg
Am Hubland, 97074 Würzburg, Germany
esser@informatik.uni-wuerzburg.de
http://www2.informatik.uni-wuerzburg.de

Abstract. This paper introduces a new approach to add fault-tolerance to a fulltext retrieval system. The *weighted pattern morphing* technique circumvents some of the disadvantages of the widely used edit distance measure and can serve as a front end to almost any fast non fault-tolerant search engine. The technique enables approximate searches by carefully generating a set of modified patterns (morphs) from the original user pattern and by searching for promising members of this set by a non fault-tolerant search backend. Morphing is done by recursively applying so called *submorphs*, driven by a penalty weight matrix. The algorithm can handle phonetic similarities that often occur in multilingual scientific encyclopedias as well as normal typing errors such as omission or swapping of letters. We demonstrate the process of filtering out less promising morphs. We also show how results from approximate search experiments carried out on a huge encyclopedic text corpus were used to determine reasonable parameter settings.
A commercial pharmaceutic CD-ROM encyclopedia, a dermatological online encyclopedia and an online e-Learning system use an implementation of the presented approach and thus prove its "road capability".

1 Introduction

One of the main advantages of digitally stored texts is the possibility to easily retrieve information from their content. In written texts there is always the possibility of errors and ambiguities concerning their content. Particularly large scientific encyclopedias, though they may have passed a thorough scrutiny, often use more than one spelling for the same scientific term.

Publishers of encyclopedias and dictionaries are often confronted with a problem when a large number of contributing authors produce the text content. These authors tend to use synonymous notations for the same specific term. This might seem of minor importance to the user of a printed edition. The user of an electronic version, however, might be misled by the fact that a

Sharon McDonald, and John Tait (eds.)
Advances in Information Retrieval
LNCS 2997, ISBN 3-540-21382-1
© Springer, 2004
Proceedings of the 26th European Conference on Information Retrieval Research, ECIR'04, 5th - 7th April 2004, University of Sunderland, U.K.

pp. 338–351

retrieval produced results. The user might have had more results when searching for a different spelling of the search term (e.g., different hyphenation, usage of abbreviations, multilingual terms).

Since 1999 our chair of computer-science has cooperated with Springer-Verlag, a well-known scientific publishing company, to compile the annual electronic version of *Hagers Handbuch der Pharmazeutischen Praxis* (Hager's Handbook of Pharmaceutic Practice) [1], the standard encyclopedia for German speaking pharmacists and pharmacologists. The printed version of "Hager's Handbook" consists of twelve volumes with about 12,300 pages. The first part (five volumes) describes active substances drawn from plants (herbal drugs), and the second (five volumes) is about synthetically produced agents (drugs). The two last volumes contain the manually generated index.

The first electronic version was released as *HagerROM 2001* at the end of 2000, and the current 3rd release was in June 2003 as *HagerROM 2003* [2]. To make the vast amount of Hager's information accessible to the end-user, a fast q-gram based fulltext retrieval system, which is briefly described in section 2, was built into *HagerROM*.

For a better understanding of the decisions we made during the development of the text retrieval system, some key data regarding *HagerROM* follows:

- The first edition was published by Mr. Hager in 1876
- 600 contributing authors wrote the current fifth edition (1995–2000) in
- 6,100 separate articles, which led to
- 121 MB of XML tagged text, which, in turn, lead to
- 157 MB of HTML tagged text in
- 16,584 HTML files, which resulted in
- 53 MB raw text T (after removing layout tags) with
- >160 symbol long alphabet Σ (after lowercasing T)

We soon were confronted with the need for making the text retrieval fault-tolerant, owing to the following characteristics of Hager: The two expert-generated index volumes of the print edition list about 111,000 different technical terms drawn manually from the Hager text corpus. These entries were used in the articles by the large number of contributing authors and consist of German, Latin and English terms – making text retrieval a multilingual problem.

For example "`Kalzium`" (Ger.) has 42 occurrences and "`Calcium`" (Lat.) has about 3750 occurrences in text T. So, whatever word variant a user feeds into a non fault-tolerant search algorithm, not all usages of the technical term will be found. Additionally spelling and typing errors are a problem in such a large text corpus. For example the word "`Kronblätter`" (crown leaves), with about 600 hits, was once typed wrong as "`Kronbläter`" and occurs once correctly as a slightly changed substring of "`kronblättrig`" (crown petaled). Empirical studies by Kukich [3] have shown that the percentage of mistakes in texts is not negligible. More precisely, she found that texts contain 1%–3.3% typing errors and 1.5%–2.5% spelling errors.

This paper is organized as follows: In section 2 we first give an overview of previous work, describe our non fault-tolerant search backend, and summarize other existing fault-tolerant approaches. In section 3 we introduce the algorithm of weighted pattern morphing and provide the rationale for the parameter choices used in the design of the algorithm. Section 4 then shows the results of some retrieval experiments we carried out on the text corpus of HagerROM, to give an idea of the average running time of the algorithm. Finally, section 5 presents conclusions and ideas of future work.

2 Previous Research

Fast search algorithms store their knowledge of a text in an appropriate index, commonly implemented using one of the following data structures (see [4]): *suffix tree*, *suffix array*, *q-grams* or *q-samples* (Sometimes authors refer to q-grams and q-samples as *n-grams* and *n-samples* respectively).

2.1 Our Non Fault-tolerant Search Backend

The non fault-tolerant variant of our search engine uses a compressed q-gram index. When this index is created, the position (offset from first symbol in T) of every substring Q with length q inside text T is stored in an appropriate index (see [5] for details).

As the position of every substring of length q is stored in the index, this leads to quite large index sizes, which is seen as the main disadvantage of this indexing technique (see [4], [6]). On the other hand storage space is nowadays often a negligible factor, and so one can benefit from the enormous retrieval speed q-gram indices provide.

In the field of fault-tolerant information retrieval, vectors of q-grams are sometimes used to calculate the spelling distance of two given strings (see Ukkonen's q-gram distance measure [7]). But as this technique is rather oriented towards spelling and not towards sound we use weighted pattern morphing (WPM) for approximate matching. For our approach the q-gram index serves as an (exchangeable) exact, non-approximate search backend, where other data structures like suffix tree or a word index would also be feasible. In our case, q-grams are a good choice, as they are very fast in detecting that a special pattern is not part of the text: e.g., if any of the q-grams contained in the search pattern is not part of the q-gram index the algorithm can terminate – there are no occurrences of the pattern inside the text. This is useful, as many patterns that WPM generates may not be part of the text.

It is obvious that the size of the above mentioned offset lists is independent of q, as the position of the Q window always increases by one. Further, with increased values of q, the average length of the offset lists drops, while the total number of these lists raises, and so does the required storage space for the managing meta structure for the index.

To get reasonable retrieval performance, values of $q \geq 3$ are mandatory to avoid processing long offset lists during a search. However, when only an index for $q \geq i$ is generated, search patterns P with $|P| < i$ cannot be found in acceptable time. Consequently, indices for more than one q are needed, which leads to even more storage space requirements for the total index structure. (Note: $|X|$ denotes the length of string X in characters).

Instead of saving storage space by using q-samples, which are non-overlapping q-grams (i.e., every h^{th} q-gram, $h \geq q$, is stored in the index), we use normal, overlapping q-grams with $q = \{1,2,3,4\}$. For an approximate search approach with q-samples see [6]. But to save space, we skip every 3- and 4-gram Q where at least one character of Q is not among the f most frequent unograms (i.e., 1-gram) of the text, so called favorites.

So while the unogram and duogram index is complete, we skip every occurrence of, for example, `17_°`, `7_°C` and `_°C_` (where '`_`' denotes 'space'), while we store every position of, for example, `_rat`, `rats` and `ats_`.

This technique turned out to be extremely flexible for the process of tuning our search engine to maximum speed by filling up the available storage space (e.g., of the distributed CD-ROM) with more and more 3- and 4-grams in our index structure.

Though the storage of unograms might seem obsolete, when duograms are present, unograms are necessary for two reasons: First, retrieval of seldom used symbols like a Greek 'δ' might be important to the end-user, as even this single symbol carries enough information content to be interesting. Second, our fault-tolerant add-on (see section 3) may modify user patterns using '?' wildcards, leaving unograms close to the borders of the new pattern.

2.2 Common Techniques for Fault-tolerant Fulltext Retrieval

In 1918 Robert C. Russell obtained a patent for his Soundex algorithm [8], where every word of a given list was transformed in a phonetic, sound-based code. Using this Soundex code, U.S. census could look up surnames of citizens rather by sound instead of spelling, e.g., `Shmied` and `Smith` both have the Soundex code `S530`. Unfortunately Soundex too often gives the same code for two completely different words: `catherine` and `cotroneo` result in `C365` and similar sounding words get different codes: `night=N230` and `knight=K523`.

Although there have been many improved successors to this technique (e.g., [9] and [10]), all of them are word based and thus lack the ability to find patterns at arbitrary positions inside a text (e.g., pattern is substring of a word inside the text). Further, with sound code based systems it is impossible to rank strings that have the same code: strings are either similar (same code) or not (different code). Last, phonetic codes are usually truncated at a special word length, which make them less usable in texts with long scientific terms.

In [4] a taxonomy for approximate text searching is specified. According to this taxonomy, three major classes of approaches are known: *neighborhood generation*, *partitioning into exact search* and *intermediate partitioning.*

Neighborhood generation generates all patterns $P' \in U_k(P)$ that exist in the text, where `editdistance`$(P, P') \leq k$ for a given k (for a description of edit distance see below). These neighbor patterns are then searched with a normal, exact search algorithm. This approach works best with suffix trees and suffix arrays but suffers from the fact that $U_k(P)$ can become quite large for long patterns P and greater values of k.

Partitioning into exact search carefully selects parts of the given pattern that have to appear unaltered in the text, then searches for these pattern parts with a normal, exact search algorithm and finally checks whether the surrounding parts of the text are close enough to the original pattern parts.

Intermediate partitioning, as the name implies, is located between the other two approach classes. First, parts of the pattern are extracted, and neighborhood generation is applied to these small pieces. Because these pieces are much smaller and may have fewer errors than the whole pattern, their neighborhood is also much smaller. Then exact searching is performed on the generated pattern pieces and checked to see whether the surrounding text forms a search hit.

Various approaches have been developed to combine the speed and flexibility of q-gram indices with fault-tolerance. Owing to the structure of q-gram indices, a direct neighborhood generation is not possible in reasonable time. Jokinen and Ukkonen present in [11], how an approximate search with a q-gram index structure can be realized with *partitioning into exact search.* Navarro and Baeza-Yates in [5] use the same basic approach, but assume the error to occur in the pattern, while Jokinen and Ukkonen presume the error to be in the text, which leads to different algorithms. Myers demonstrates in [12] an *intermediate partitioning* approach to the approximate search problem on a q-gram index.

All the above methods are based on the definition of one of the two string similarity metrics published by Levenshtein in [13] called *Levenshtein distance* and *edit distance.* Both metrics calculate the distance between two strings by summing up the minimal costs of transforming one string into the other by counting the atomic actions *insert*, *delete* and *substitute* of single symbols [14].

Though these metrics provide a mathematically well-defined measure for string similarities, they also suffer from the inability to model similarity of natural language fragments satisfactorily, from a human point of view.

With regard to the special characteristics of the Hager text corpus, the use of the edit distance measure did not seem appropriate. This is mainly due to the fact that the edit distance processes only single letters (regardless of any context information) and does not provide the means of preferring a string substitution $A \rightarrow B$ versus $A \rightarrow C$, where $|A| \geq 1$, $|B| \geq 1$ and $|C| \geq 1$ and $|A|$, $|B|$ and $|C|$ are arbitrarily different.

For example: `editdistance`(“`kalzium`”, “`calcium`”)=2 and `editdistance` ("kalzium", "tallium")=2, are the same – despite the fact that every human reader would rate the similarity of first two strings much higher than the similarity of the second pair of strings.

Because the edit distance is more suited to model random typing mistakes or transmission errors, we needed a way to approximate patterns where the differences between text and pattern are less "random" but more due to the fact that a great number of authors may use the same scientific term in different (but correct) spellings. We also wanted to cope with the problem of non-experts knowing how a scientific term sounds, without exactly knowing the correct spelling. Our technique of *weighted pattern morphing* is described in the next section.

3 Weighted Pattern Morphing

In this section we present the architecture and algorithms of our fault-tolerant frontend, which is based on the *weighted pattern morphing* approach. Afterwards we show the results of experiments that led to reasonable parameter settings for our fault-tolerant search engine.

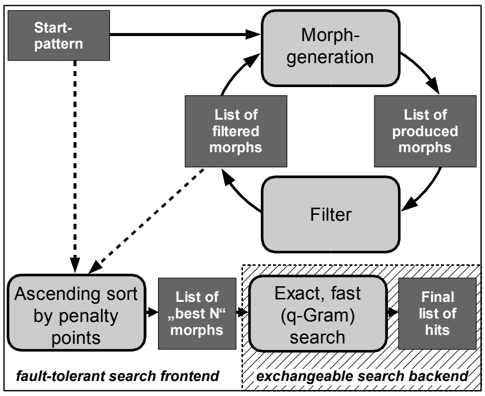


Fig. 1. Workflow of weighted pattern morphing frontend and search backend

3.1 The Fault-tolerant Search Frontend

As stated at the end of the previous section the edit distance metric, which is used by most available approximate text retrieval algorithms, is not appropriate, when one is trying to model a more human-oriented string similarity. Weighted pattern morphing (WPM) overcomes the mentioned disadvantages with a simple but powerful idea:

Browse searchpattern P for all substrings $p_{i,j}$ $(1 \leq i \leq j \leq |P|)$, which are part of a phoneme group G with $G=\{g_1,g_2,\ldots,g_z\}$ and where $p_{i,j}=g_k$ $(1 \leq k \leq z)$ and try to replace $p_{i,j}$ by all g_l $(l \neq k)$ which are members of the same phoneme group G.

More general as with the edit distance, here $|p_{i,j}| \geq 1$, $|g_l| \geq 1$ and even $|p_{i,j}| \neq |g_l|$ is possible. A pattern P', where at least one substitution took place is called a *morph* of P and a single substitution of $p_{i,j}$ to g_l is called *submorph* $p_{i,j} \longrightarrow g_l$.

As the interchangeability of members of the same phoneme group should be different, the concept of *penalty weights* was introduced. These penalty weights were stored in two-dimensional *submorph matrices* with source strings g_k in rows and destination strings g_l in columns (see examples in table 1).

Table 1. Two example penalty weight matrices (phoneme group "cgk..."; numbers)

	c	**g**	**k**	…
c	–	–	1	…
g	10	–	10	
k	1	15	–	
…	…			

	1	**one**	**2**	…
one	1	–	–	…
1	–	1	–	
two	–	–	1	
…	…			

As the table demonstrates, not every possible submorph is allowed, and the matrix may be asymmetric to the diagonal. There exist submorph tables for every common phoneme group like "a/ah/aa/ar", "i/ie/y/ih/ii", "g/j", "c/g/k/cc/ck/kk/ch", and so on. The possibilities of the edit distance can be approximated by submorphs like $\varepsilon \longrightarrow$ ”?” (insert any char), $c \in \Sigma \longrightarrow$ ”?” (substitute a char c), $c \in \Sigma \longrightarrow \varepsilon$ (delete), where ε is the empty word, Σ the alphabet and "?" is the one-letter wildcard for our search engine. But even more exotic submorphs like `solution` ⟶ `sol.`, `acid` ⟶ `ac.`, `5` ⟶ `five` are defined. These are often helpful in a biochemical and medical contexts, because abbreviations are used inconsistently by different authors (e.g, in *HagerROM* the terms "`5-petaled`" and "`five-petaled`" occur).

For the German language mixed with Greek and Latin terms we manually identified about 25 different phoneme groups that lead to about 350 submorphs. The penalty weights for these string pairs were adjusted manually from a native speaker's point of view. Automatically adjusting the weights is subject to ongoing research, and our early results seem quite promising. For generation of English morph matrices we relied on linguistic research publications like e.g., Mark Rosenfelder's "*Hou tu pranownse Inglish*" [15]. Though Rosenfelder presents rules to get from spelling to sound, we used his work to identify about 35 English sound groups and their possible spelling which lead to English morph matrices with about 900 submorphs. Additional submorphs for number names and numbers (`100`⟶`hundred`, `hundred`⟶`100`) and domain specific abbreviations were added afterwards.

Every morphed pattern P' is recursively fed into the same morph algorithm, to perform even more submorphs. To avoid recursion loops, the first index i_{min} where submorphs $p_{i,j} \longrightarrow g_l$ may start, is always increased for deeper recursion levels. Loops otherwise may appear through submorphs at different recursion levels like $u \longrightarrow v$, $v \longrightarrow w$, $w \longrightarrow u$. On every recursion level, P is also fed unaltered into the next recursion, with only i_{min} increased, to also allow submorphs only towards the end of the pattern.

Because the recursion tree can get large, the total penalty S, as sum of the penalty weights for all applied submorphs, and M, the total number of applied submorphs (=recursion depth), are updated for every recursion call. Recursion backtracking is performed when either S or M pass configurable limits S_{max}, M_{max} or when $i_{min} > |P|$. As S_{max}, M_{max} and i_{min} grow with every recursion level, the algorithm terminates in reasonable time (see section 4).

Obviously, the above algorithm generates many morphs that are not part of the text corpus. Though the q-gram algorithm is very fast in finding out that a pattern has no hits in the text (this is so, because the search always starts with the shortest q-gram offset list, see [16]), pre-filtering of "useless morphs" was achieved by the introduction of the *hexagram filter*.

This hexagram filter possesses a trie structure with a maximum depth of six, but does not store actual offsets of hexagrams. It simply indicates whether a specific q-gram class ($q \leq 6$) exists in the text at all.

So when a morph P' is generated, the hexagram trie is traversed for every (2^{nd} overlapping) hexagram that is part of P'. If any of the morph's hexagrams is not part of the trie, P' as a whole cannot be part of text T and is discarded. However, if all the hexagrams of P' are part of the trie, there is no guarantee that P' occurs in T, because all hexagrams are part of T, though not necessarily in the same order as in P'. In these cases we rely on the ability of the q-gram algorithm to terminate quickly for those patterns that are not part of the text.

When checking the q-grams of P′ against the trie structure, there are two parameters that influence the accuracy of the filter: *trie depth* TD (we used a depth of six) and *window delta* WD of the hexagrams drawn from P′. The window delta states whether every hexagram of P′ is taken (delta=1) or every second hexagram (delta=2) and so on. Smaller values of trie depth and larger values of window delta increase filter speed but reduce accuracy – and thus result in more promising morph candidates, which results in longer overall time for the algorithm.

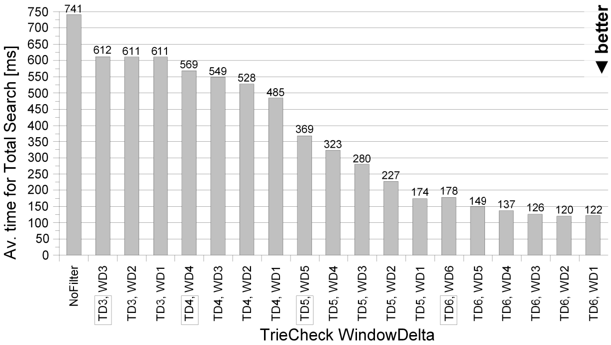


Fig. 2. *Operating time for different accuracy levels of the trie filter*

So, to obtain reasonable values for these two parameters, we executed fault-tolerant searches with about 14,000 patterns drawn from the text and recorded the average running times for different values of trie depth TD and window delta WD. These experiments were performed on an Intel® Pentium® IV 2.6 GHz processor with 512MB of RAM, and the results are shown in figure 2. We observed a minimum running time at TD = 6 and WD = 2, which is the reason why we chose these values for all subsequent experiments. Though these results seems portable to other Indo-European languages, it is a topic for future research whether the above values of TD and WD are appropriate for other text corpora, too.

Every time a submorph is applied, the resulting morph P^* (if it has passed the hexagram filter) is stored in a hashmap, together with S_{P^*}, its sum of penalty weights. When the WPM algorithm terminates, the list of generated morphs is sorted in ascending order by S_{P^*}. The best B morphs (those with least penalty weights) are then kept as the *final morph list* of the original pattern P. The limit B is configurable. Each triple of values S_{max}, M_{max} and B defines a *fault-tolerance level.*

3.2 Experiments to Determine Reasonable Parameter Settings

As stated in the previous subsection, the degree of fault-tolerance of the weighted pattern morphing algorithm can be controlled by 3 parameters:
1. S_{max} the maximum sum of penalty weights a morph may aggregate,
2. M_{max} the maximum number of submorphs a morph may accumulate, and
3. B the number of best rated morphs that is fed into the search backend.

The patterns an end-user presents to the search-engine remain an unknown factor, therefore we chose the following procedure to gain test patterns for our experiments: We first split up the whole text T into all of its words. As word delimiter d we chose (in perl notation):
$d \in$ `[\=\+\s\.\!\?\,\;\:\(\)\[\]\{\}\/\"\“\”\„\±\×\®\°\†\‡\…\~\'\*\•\xA0\%]`
Words with embedded hyphens were stored as a whole and additionally all of their fragments (separated by hyphens) were added. All words W with $|W|<9$ or $|W|>30$ were discarded. Applied to the texts of *HagerROM* this produced about 260,000 different words.

Every word W of the resulting word list WL_1 was then fed into our fault-tolerant search, while allowing very high values for S_{max}, M_{max} and B. All words of WL_1 where the algorithm generated morphs P' with $P' \in WL_1$ produced the condensed WL_2 with 14,000 different words. To minimize the runtime of the following experiments, every third word was chosen, resulting in WL_3 with about 4,600 words and an average word length of 14 chars.

So, every search pattern P' of WL_3 was part of the original text T and could be morphed (with high values for S_{max}, M_{max} and B), so that one or more of its morphs are again part of the total word list WL_1 – these morphs are called *valid target-morphs*. This was done to find out to what extent S_{max}, M_{max} and B can be decreased while keeping as many valid target-morphs as possible. The fact that only morphs P' with $P' \in WL_1$ were accepted in all the following experiments minimized the number of "useless" morphs. During the experiments we determined how many valid target morphs $P' \in WL_1$ the algorithm produced for a given parameter set of S_{max}, M_{max} and B.

The weight values for the submorph matrices were manually generated and carefully chosen from a linguistic point of view based on our experience with different multilingual text corpora (see section 5 for ideas on automatic weight generation and optimization).

Weight values were taken from integer values [*1, 2, 5, 10, 15, 20, 25, 30*] so that not every possible value was chosen, but rather "classes" of weights such as [*very-helpful, helpful, ..., maybe-dangerous*] were used. Other numerical weight ranges are possible, but probably won't lead to better results.

The following three figures present the results of experiments where two parameters were kept fixed and the third parameter varied on each test run.

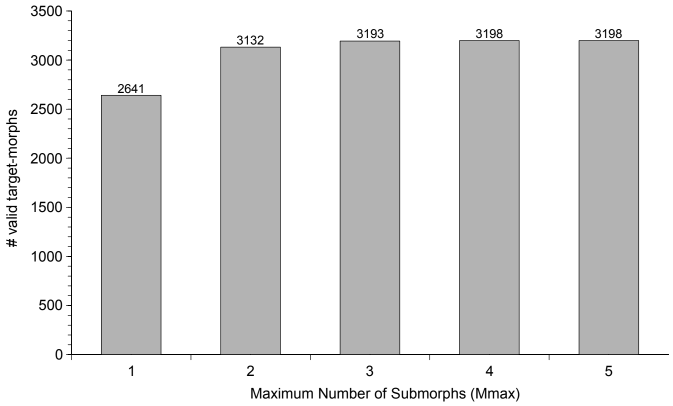


Fig. 3. *Experiment#1:* M_{max} variable [1, 2, ..., 5] (fixed: S_{max}=60, B=200)

Experiment#1 (see figure 3) led to the conclusion that M_{max} (the maximum number of applied submorphs on the original pattern) should not get greater than 4, because no increase in valid target-morphs was achieved by higher values – only more runtime was needed. The fast rise of valid target-morphs was based on the fact that S_{max} and B have quite high values in comparison to the maximum rule weight of 30.

The abrupt rise of the bar at "1 applied submorph" is due to the fact that for most word variants or words with errors only one small change (like insertion, deletion, transposition) has to be applied. Karen Kukich in [3] (see page 388) cites Mitton (1987) who examined a 170,016-word text corpus and revealed that about 70% of the misspelled words contained only a single error.

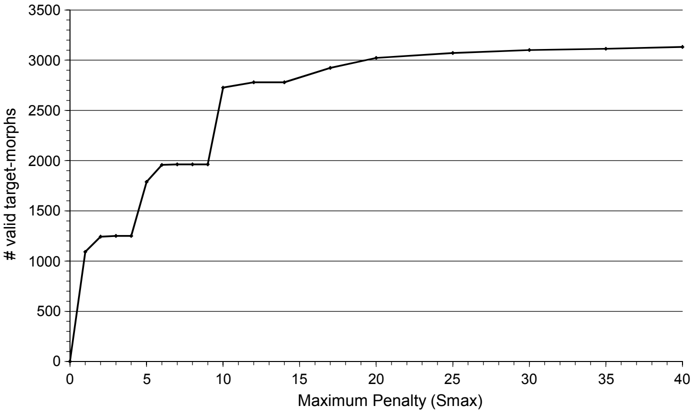


Fig. 4. *Experiment#2:* S_{max} variable [0, 1, ..., 40] (fixed: M_{max}=2, B=200)

Experiment#2 (see figure 4) showed that S_{max} (the maximum sum of penalty weights a morph is allowed to collect) should not be higher than 30, which is at the same time the maximum weight used in the weight matrices. The obvious step structure of the graph in figure 4 is due to the fact that not every arbitrary weight value from the interval [*1, 2, ..., 29, 30*] was used in the weight matrices (see above).

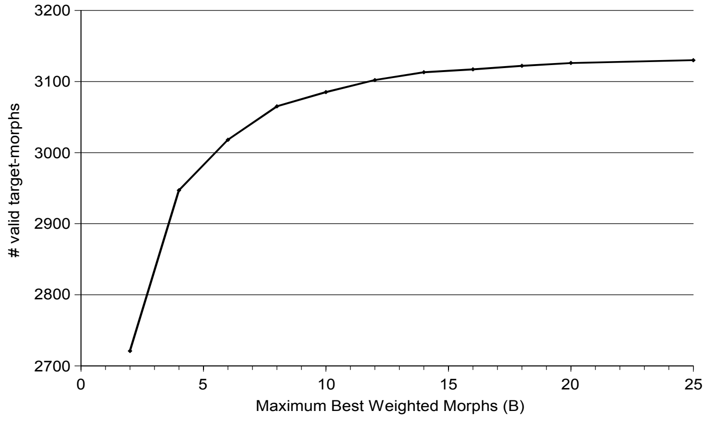


Fig. 5. *Experiment#3:* B variable [2, 3, ..., 25] (fixed: M_{max}=2, S_{max}=60)

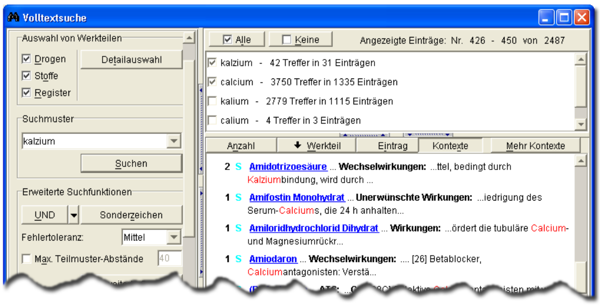
Finally, Experiment#3 (see figure 5) justifies our decision to always feed only a maximum of 20 best rated morphs to the non fault-tolerant search backend. Higher values for B may increase the overall runtime but won't improve search results any further. Note that the Y-axis of figure 5 was cut-off at a value of 2700 to allow better evaluation of the graph.

To simplify the use of the fault-tolerance feature by the end-user, macro levels labeled *low*, *medium* and *high* were established and grouped values for S_{max}, M_{max} and B, according to table 2.

Table 2. Reasonable parameter settings for different fault-tolerance levels

	low	***medium***	***high***
S_{max}	10	20	30
M_{max}	2	3	4
B	10	15	20

The graphical user interface provides the possibility to select and deselect from the list of occurring morphs, to post-filter variants of the original pattern which might be of less importance to the user. For example, a fault-tolerant search for `kalzium` produces also morphed hits for `kalium` and `calium` (Engl.: potassium), which is a different chemical element. The screenshot of figure 6 shows a section of the (German) user interface.



☑ Drogen
☑ Stoffe
☑ Register

Suchmuster: kalzium

UND

Fehlertoleranz: Mittel

☐ Max. Teilmuster-Abstände 40

Angezeigte Einträge: Nr. 426 - 450 von 2487

☑ kalzium - 42 Treffer in 31 Einträgen
☑ calcium - 3750 Treffer in 1335 Einträgen
☐ kalium - 2779 Treffer in 1115 Einträgen
☐ calium - 4 Treffer in 3 Einträgen

2 S Amidotrizoesäure ... **Wechselwirkungen:** ...ttel, bedingt durch Kalziumbindung, wird durch ...

1 S Amifostin Monohydrat ... **Unerwünschte Wirkungen:** ...iedrigung des Serum-Calciums, die 24 h anhalten...

1 S Amiloridhydrochlorid Dihydrat ... **Wirkungen:** ...ördert die tubuläre Calcium- und Magnesiumrückr...

1 S Amiodaron ... **Wechselwirkungen:** .... [26] Betablocker, Calciumantagonisten: Verstä...

Fig. 6. *HagerROM* – Results of a Fault-Tolerant Fulltext Retrieval with WPM

4 Experiments

In this section we discuss some experiments regarding the filter efficiency and the speed of the presented fault-tolerant approach. Based on the characteristics listed in the table below, we used the text corpus of *HagerROM* for our experiments, because the true power of WPM shows most notably on large texts which are a real challenge to a text retrieval system. This amount of text (13 times as large as "The Bible") and the vast number of about 600 contributing authors make the WPM based fulltext search an important part of the commercial CD-ROM product. Other examples for successful application of our WPM approach are the *DEJAVU* online e-Learning system and Prof. Altmeyer's "*Springer Enzyklopädie Dermatologie, Allergologie, Umweltmedizin*" (Springer's Encyclopedia on Dermatology, Allergology and Environmental Medicine). For details on DEJAVU (Dermatological Education as

Joint Accomplishment of Virtual Universities), see [17]. Springer′s encyclopedia provides free online-access for physicians on [18].

Table 3. Characteristics of three products using WPM search

Module	***DEJAVU***	***Altmeyer***	***HagerROM***
Text (with Layout)	1.0 MB	22.7 MB	121 MB
Raw text (w/o Layout)	0.4 MB	5.8 MB	53 MB
Hexagram trie filter	0.3 MB	1.2 MB	6 MB
q-gram index	4.3 MB	70.2 MB	450 MB

The following table shows the results of some experiments with fault-tolerant WPM searches. The number of actual hits of a search pattern is given within parentheses. We also tested patterns that were not part of the original text, but which were transformed into valid words after passing the WPM algorithm and so, finally, produced hits in the text corpus.

Table 4. Experiments with WPM on the *HagerROM* text corpus

Original pattern	***MT sec.***	***ST sec.***	***UT sec.***	***#M***	***#F***	***#N***	***#H***	***Morphs with hits***	***# w/o filter***
azethylsalizyl **(0)**	0.23	0.12	0.53	1669	1655	14	2	acetylsalizyl(4), acetylsalicyl(435)	15035
kalzium (42)	0.05	0.01	0.23	343	336	7	5	kalzium(42), calcium(3750), kalium(2779), calium(4), cal?cium(3)	639
pneumokocken-polysacharid **(0)**	0.27	1.19	1.63	2283	2192	91	1	pneumokokken-polysaccharid (4)	129040
schokolade (54)	0.47	2.05	2.75	1578	1551	27	4	schokolade(54), shokolade(1), chocolade(1), chocolate(4)	6498
sulfamethoxy-diazin (2)	0.33	1.03	1.58	2739	2656	83	3	sulfamethoxydiazin(2), sulfametoxydiazin(17), sulfametoxidiazin(1)	24739

Legend of table 4. MT=morph time: time consumed to calculate the best #N morphs; **ST**=search time: time consumed by the non fault-tolerant search back-end to search for these best #N morphs; **UT**=user time: the total time the user has to wait for all results (with program launch time). **#M**: number of actual generated morphs; **#F**: number of morphs that did not pass the hexagram filter; **#N**: number of morphs that passed the filter with an acceptable amount of penalty weights; **#H**: number of morphs from the #N that produced at least one hit in the text corpus; **#w/o filter**: without hexagram filtering this number of (mostly useless) different morphs would have been generated.

All experiments were performed on a standard PC with AMD Athlon® 1.33GHz CPU and 512 MB RAM on a local ATA-66 harddisk under Windows XP®. The compressed q-gram index q={$1,2,3,4$} needs about 450MB storagespace (this is 8 times $|T|$) and can be generated on an ordinary Linux computeserver in about one hour.

Table 4 demonstrates that on an average PC hardware, fault-tolerant text retrieval with practical search patterns can be accomplished using the approach of weighted pattern morphing in acceptable time. Within the presented examples the user has to wait an average of two seconds to obtain the wanted results. The hexagram trie filter prevents the algorithm from generating thousands of morphs that can't be part of the text and thus contributes to a faster response of the system.

From our discussion it is obvious that the filter becomes less accurate with longer search patterns. This is due to the fact that the filter can only determine that every six character substring of a morph is part of text T. The filter can't determine whether these existing six character substrings of the morphed pattern also occur in the same order and at the same distances inside text T.

5 Conclusion and Future Work

We demonstrated that nowadays average end-user PCs are capable of performing multiple, iterated, exact text retrievals over a set of morphed patterns and thus simulate a fault-tolerant search. Morph matrices with penalty weights seem much more suitable and flexible to model phonetic similarities and spelling variants in multilingual, multi-author texts than the edit distance metric or phonetic codes like Soundex and its successors. Weighted pattern morphing can generate edit distance like spelling variants (delete or swap letters, insert "?" one-letter wildcards) and the algorithm can also put emphasis on phonetic aspects like sound-code based algorithms. It thus combines the strength of these two approaches.

The presented algorithm can be added on top of any exact search engine to create a fault-tolerant behavior. A q-gram index fits extremely well as exact non-fuzzy search backend, because a "no-hit" result can be detected in short time and wildcards ("?", "*") are easy to implement without extra time costs.

It will be part of future research to automatically fine-tune the penalty weights in order to customize the system to a special text. We are planning to run large test series and keep track of how often a submorph produced a valid target-morph. The collected data will enable us to fine-tune submorph weights for even better performance.

References

1. Bruchhausen F.v. et al. (eds.): *Hagers Handbuch der Pharmazeutischen Praxis. 10 (+2) Bände. u. Folgebände.* Springer Verlag, Heidelberg (1992-2000)
2. Blaschek W., Ebel S., Hackenthal E., Holzgrabe U., Keller K.,Reichling J. (eds.): *HagerROM 2003 - Hagers Handbuch der Drogen und Arzneistoffe. CD-ROM.* Springer Verlag, Heidelberg (2003) http://www.hagerrom.de
3. Kukich K.: *Technique for automatically correcting words in text.* ACM Computing Surveys 24(4) (1992) 377-439
4. Navarro G., Baeza-Yates R., Sutinen E., Tarhio J.: *Indexing Methods for Approximate String Matching.* IEEE Bulletin of the Technical Committee on Data Engineering, Vol. 24, No. 4 (2001) 19-27
5. Navarro G., Baeza-Yates R.: *A Practical q-Gram Index for Text Retrieval Allowing Errors.* CLEI Electronic Journal, Vol. 1, No. 2 (1998) 1
6. Sutinen E.: *Filtration with q-Samples in Approximate String Matching.* LNCS 1075, Springer Verlag (1996) 50-63
7. Ukkonen, E.: *Approximate string-matching with q-grams and maximal matches.* Theoretical Computer Science 92 (1992) 191-211
8. Russell R.: *INDEX (Soundex Patent).* U.S. Patent No. 1,261,167 (1918) 1-4
9. Zobel J., Dart Ph.: *Phonetic String Matching: Lessons from Information Retrieval.* ACM Press: SIGIR96 (1996) 166-172
10. Hodge V., Austin J.: *An Evaluation of Phonetic Spell Checkers.* Dept. of CS, University of York, U.K. (2001)
11. Jokinen P., Ukkonen E.: *Two algorithms for approximate string matching in static texts.* LNCS 520, Springer Verlag (1991) 240-248
12. Myers E.: *A sublinear algorithm for approximate keyword searching.* Algorithmica, 12(4/5) (1994) 345–374
13. Levenshtein V.: *Binary codes capable of correcting deletions, insertions, and reversals.* Problems in Information Transmission 1 (1965) 8-17
14. Stephen G.: *String Searching Algorithms, Lecture Notes Series on Computing, Vol. 3.* World Scientific Publishing (1994)
15. Rosenfelder M.: *Hou tu pranownse Inglish* http://www.zompist.com/spell.html (2003)
16. Grimm M.: *Random Access und Caching für q-Gramm-Suchverfahren.* Lehrstuhl für Informatik II, Universität Würzburg (2001)
17. Projekt DEJAVU: *Homepage* http://www.projekt-dejavu.de (2003)
18. Altmeyer P., Bacharach-Buhles M.: *Springer Enzyklopädie Dermatologie, Allergologie, Umweltmedizin.* Springer-Verlag Berlin Heidelberg (2002) http://www.galderma.de/anmeldung_enz.html