Fast Error-Tolerant Search on Very Large Texts

Marjan Celikik
Max Planck Institute for Computer Science
Saarbrücken, Germany
mcelikik@mpi-inf.mpg.de

Holger Bast
Max Planck Institute for Computer Science
Saarbrücken, Germany
bast@mpi-inf.mpg.de

ABSTRACT

We consider the following spelling variants clustering problem: Given a list of distinct words, called lexicon, compute (possibly overlapping) clusters of words which are spelling variants of each other. This problem naturally arises in the context of error-tolerant full-text search of the following kind: For a given query, return not only documents matching the query words exactly but also those matching their spelling variants. This is the inverse of the well-known "Did you mean: ... ?" web search engine feature, where the error tolerance is on the side of the query, and not on the side of the documents.

We combine various ideas from the large body of literature on approximate string searching and spelling correction techniques to a new algorithm for the spelling variants clustering problem that is both accurate and very efficient in time and space. Our largest lexicon, containing roughly 10 million words, can be processed in about 16 minutes on a standard PC using 10 MB of additional space. This beats the previously best scheme by a factor of two in running time and by a factor of more than ten in space usage. We have integrated our algorithms into the CompleteSearch engine in a way that achieves error-tolerant search without significant blowup in neither index size nor query processing time.

Categories and Subject Descriptors

H.3.3 [**Information Search and Retrieval**]: Search process; H.3.4 [**Systems and Software**]: Performance evaluation (efficiency and effectiveness); H.5.2 [**User Interfaces**]: Theory and Methods

General Terms

Algorithms, Experimentation, Measurement, Performance

Keywords

Spelling Variants, Approximate String Matching, Error-Tolerant Search

1. INTRODUCTION

Text search is still one of the primary means of accessing information. However, large collections of natural language texts abound in misspellings, either produced by the authors of the documents, or incurred in the process of converting the document into electronic form, e.g., via optical character recognition (OCR). Such errors can be a problem when searching for information in a scenario where high recall matters, i.e., we do not want to miss a single relevant document.

For example, in the current version of the DBLP dataset, a very carefully maintained collection of meta data on over a million computer science articles (`http://dblp.uni-trier.de`), one of the paper titles is misspelled as "An accurate probalistic model for error detection." A query *probabilistic error detection* will fail to retrieve this document, possibly leading to the erroneous conclusion that it does not exist in the database. In a study of [8], 10% of one million queries returned irrelevant hits or no hits at all due to misspellings and OCR errors in the searched text collection. Other studies [19, 20] show that the accuracy on short documents under probabilistic IR and the vector space model decreases rapidly as the number of OCR errors increases.

Our goal in this paper is error-tolerant text search: given a query, retrieve not only those documents which contain the query words, but also those documents which contain misspellings of the query words. We will show that we can realize this feature with high accuracy (most of the misspellings will be caught), yet with only small extra cost in both query processing time and index space, as compared to ordinary text search. In contrast, all search engines we know of that implement the kind of error-tolerant search we consider here, have a slow down in query processing time by an order of magnitude if that feature is activated; see Section 2.

Note that our problem here is the exact opposite of the well-known "Did you mean . . . ?" feature of many of today's web search engines. This feature deals with the fact that users tend to mistype their query, and in that case offers the most likely correct version of the query. For example, the analogon to our example above would be that a user typed *probalistic error detection*, and the system would ask "Did you mean: probabilistic error detection". However, the correctly typed query would not find documents where one of the query words occurs only in misspelled form. The two features are exactly complementary, and ideally one would like to have both. From the perspective of a web search engine operator, the "Did you mean . . . ?" feature is more important, since most queries are for popular topics with an abundance of relevant documents (containing the query words in correct spelling), but users tend to misspell their queries. For more specialized queries or more homogeneous collection, however, recall becomes a critical issue, and the second feature becomes essential. Literature search is a good example for that: we do not want to miss even a single relevant paper.

The rest of the paper is organized as follows. In Section 2 we sift the large body of literature on approximate string matching and spelling correction for work relevant to our problem. In Section 3 and Section 4 we give a formal definition of our problem and how a solution can be used for the kind of error-tolerant search we consider here. In Section 5 we present our new algorithm. Section 6 provides the results of our experiments, where we compare our algorithm against the six most relevant methods from the literature on six different datasets ranging from small to very large and with various kinds of spelling errors. We also compare the efficiency of our error-tolerant text search with the state of the art Lucene search engine.

2. RELATED WORK

The body of literature on approximate string matching, spelling correction, and the like is enormous, and we have done our best to get an as complete as possible overview of that part of this work relevant to our problem.

2.1 Techniques for Similarity String Search

In this subsection, we give an account of all the techniques we know of that can be meaningfully applied to solve the problem of *similarity string search* efficiently: for a large set of strings (words) and a given query word find all similar words from the set. Section 5 describes a generic way of how to use an algorithm for the similarity string search problem to solve our spelling variants clustering problem. We will make use of this generic way when comparing our algorithm against most of the techniques discussed in the following.

We focus on algorithms based on the *Levenshtein distance*, which we found most appropriate for our problem of error tolerant search. We give some evidence for this decision in Section 6. Essentially the same conclusion was reached in [21].

Searching in metric spaces This is a general approach in similarity searching which exploits the triangle inequality to preprocess a given data set in order to minimize the number of distance computations at query time [13]. As an extreme representative of this method we consider the AESA algorithm, which has the fastest query time of any algorithm in this class at the price of very expensive $\Theta(n^2)$ time and space preprocessing. Very recently there has been a proposal (iAESA) which improves the efficiency of up to 75% [9]. In our experiments we will ignore the preprocessing costs.

Signature-based algorithms and LSH These approaches work by mapping strings to smaller *signatures* or *fingerprints* such that similar objects hash to the same signature with probability equal to their similarity. Hence, the expensive string distance computations are reduced to cheaper signature comparisons. The most powerful representative of this class is locally sensitive hashing (LSH). All LSH schemes we know of are based on simple distance like Jaccard and Cosine distance [6] which are clearly inferior to the Levenshtein distance as far as accuracy of our error-tolerant search is concerned.

Searching with a trie A trie is an ordered tree data structure used to store strings where all the descendants of a node have a common prefix of the string associated with that node. Using a trie is a classical approach to compute the Levenshtein distance of a string against a dictionary of strings (for an example, see [3]). The savings come from the fact that the distance, up to their common prefixes, is calculated for all strings in a single step. Pruning takes place whenever the minimum value of the current column is larger than the threshold since the minimum in all descendant nodes will be also larger. The algorithm requires $O(n)$ space with a large constant. It's inefficiency comes from traversing a large fraction of the trie if the threshold is not sufficiently low (e.g. 1 or 2).

Q-gram indexing and prefix filtering These methods convert the constraint given by one distance function into a weaker constraint, usually the Q-gram overlap constraint and then integrate various filtering criteria (e.g. the prefix filter) with a Q-gram index. Approaches based on inverted indices usually address the similarity join problem (which is similar to ours) [4, 2, 5]. A very recent algorithm given in [5] outperforms all other in the same class. We conducted a small experiment which measures the running time of an ideal filtering algorithm based on the weaker constraint and we assumed that the candidate generation takes no time. Our observation was that verifying the candidate pairs alone takes longer than the total time of the algorithms we consider.

FastSS and neighborhood generation The FastSS algorithm presented in [17] uses the deletion neighborhood of a string to model the Levenshtein distance. It exploits the fact that after performing a certain sequence of deletions two strings will eventually end up being equal. The Levenshtein distance can then be efficiently computed. However, generating the deletion neighborhoods is efficient only if the strings are sufficiently short.

The algorithm has a good worst-case search complexity of $O(km^k)$, where k is the Levenshtein distance threshold and m is the average string length. The space complexity of the algorithm is $O(m^{k+1}n)$ and it can be problematic if the indexed lexicon is large.

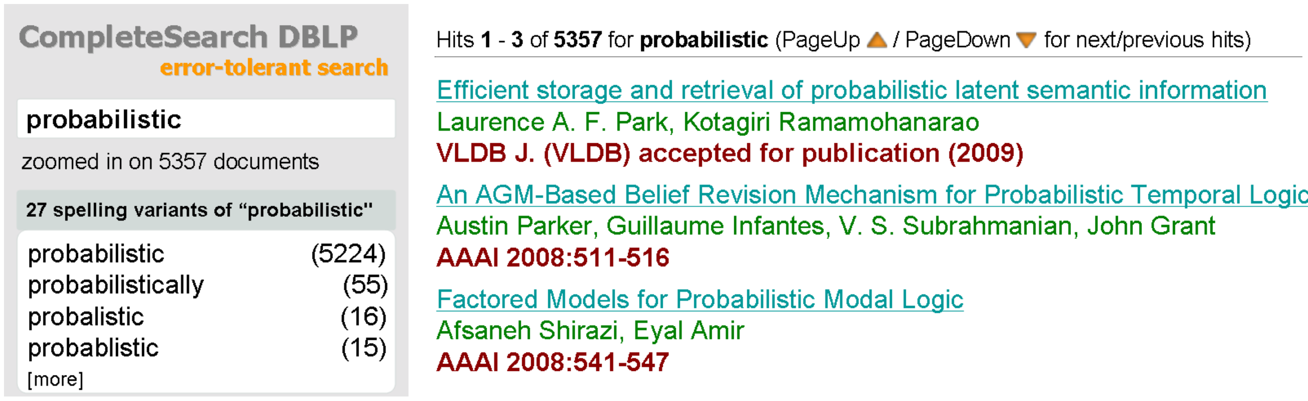
Universal Levenshtein automata (ULA) is a finite state automaton which given special precomputed information, for any pattern p can recognize all strings s with Levenshtein distance less than a threshold [11]. Once a FSA representation of the dictionary (trie) is computed its traversal is navigated by backtracking the ULA, much in spirit like the trie-based algorithm described above (thus suffering from the same problem). The pattern is then partitioned and multiple subsearches with lower thresholds are launched. The inefficiency comes from the potentially large overlap of the multiple traversals. It should be noted that our algorithm performed roughly seven times faster than (their own implementation of) their algorithm.

2.2 Existing IR Systems

Most of the more widely used open source search engines come with a fuzzy search feature similar to our error-tolerant search feature. In all of those we have looked at, this feature is realized by a simple *disjunctive query expansion*: just replace each query word q by a disjunction of all its spelling variants $v_1, \ldots, v_l$ that occur in the collection. This badly hurts efficiency in two ways: (i) a much larger number of index items needs to be read from disk and processed, and (ii) top-k techniques which avoid scanning the complete index lists for each query word do not pay off for large disjunctions. Indeed, our experiments in Section 6 will show that for Lucene, the most prominent open source engine, the query processing time for error-tolerant search is one to two orders of magnitude higher than for ordinary search or our implementation of error-tolerant search.

There is also a large number of research prototypes developed by the IR community. We have looked at Zettair, Indri, Wumpus, MG, Terrier, and Galago, and none of them seems to offer any kind of error-tolerant search. This is understandable, since these systems were built primarily as research tools and are not meant to cover the whole spectrum of standard search engine functionality as required in applications.

As far as the big commercial web search engines are con-



CompleteSearch DBLP
error-tolerant search

probabilistic

zoomed in on 5357 documents

27 spelling variants of "probabilistic"

probabilistic	(5224)
probabilistically	(55)
probalistic	(16)
probablistic	(15)

[more]

Hits **1** - **3** of **5357** for **probabilistic** (PageUp / PageDown for next/previous hits)

Efficient storage and retrieval of probabilistic latent semantic information
Laurence A. F. Park, Kotagiri Ramamohanarao
VLDB J. (VLDB) accepted for publication (2009)

An AGM-Based Belief Revision Mechanism for Probabilistic Temporal Logics
Austin Parker, Guillaume Infantes, V. S. Subrahmanian, John Grant
AAAI 2008:511-516

Factored Models for Probabilistic Modal Logic
Afsaneh Shirazi, Eyal Amir
AAAI 2008:541-547

Figure 1: A screen shot of our error-tolerant search for the query `probabilistic` **on DBLP. The box on the left shows spelling variants and the number of hits they lead to. The box on the right shows selected hits for all spelling variants. A click on a particular variant would show only the respective hits.**

cerned (Google, Yahoo!, MSN, Amazon etc.), most of them provide the "Did you mean ...?" feature suggesting the (presumable) correction of a mistyped query, but none of them addresses the problem of error-tolerant search. As we discussed earlier already, this makes sense in a web setting, where relevant documents (with keywords correctly spelled) abound.

3. PROBLEM DEFINITION

We define a *lexicon* to be the set of all terms that appear in a collection of documents. A term can be a *valid word* or a *non-word*. Non-word can be a *garbage string* or a *misspelling* and a misspelling on the other hand can be a *spelling-variant* of a valid word (e.g. `informatin` for `information`); a *run-on*, i.e., concatenation of two or more valid words (e.g. `informationprocessing`) and a *split word* (e.g. `infor` and `mation`). The latter two types of errors are called word-boundary errors [10].

Definition: (Spelling variants clustering problem) *Given a lexicon of distinct words, compute (possibly overlapping) clusters of words which are spelling variants of each other.*

For example, consider the (very small) lexicon consisting of the words: `algorithm`, `alogritm`, `algorithm`, `logarithm`, `logaythm`, `maschine`, `mahcine`, `machine` and `logarithmmachine`. Then a reasonable clustering would be as follows:

```
algorithm   logarithm          machine
alogritm    logaythm           mahcine
alogrithm   alogrithm          maschine
alogrithm   logarithmmachine   logarithmmachine
```

The non-word `alogrithm` belongs to two clusters since it can reasonably assumed to be a misspelling for both `algorithm` and `logarithm`.

4. ERROR-TOLERANT SEARCH

Given a clustering according to the definition above, we realize our error-tolerant search via the prefix search and completion mechanism introduced in [1].

Definition: (Prefix search and completion) *Given a query, compute a ranked list of (ids of) documents, containing the query words as prefixes, as well as for each query word a ranked list of (ids of) words starting with that word.*

For example, on DBLP the query `err* tol*` (to emphasize the prefix search, we will here and in the following append a star to every query word) will produce completions `err`, `error`, `errol` (an author's name), etc. for `err`, and `tolerant`, `tolerating`, `tolhuizen` (another author's name), etc. for `tol`, and a list of hits containing any combination of these completions.

In the following description, we view the search index as consisting of *postings*, where each posting is a tuple (word, document id, position, score). That is, each posting corresponds to a particular occurrence of a particular word in a particular document (if a word occurs three times in a document, there will be three different postings, one for each occurrence). We assume that scores are used in the standard way, that is, the score of a matching document is computed as the sum of the scores of the matching postings, and documents are then ranked by these score sums. Positions are used for phrase and proximity queries. Scores and/or positions may be omitted in which case the index is not capable of ranking and/or phrase and proximity queries.

Given a solution of the spelling variants clustering problem, we *add* the following postings to the index. At index time, for every posting corresponding to an occurrence of a misspelled word `m`, we create a copy of that posting, changing the word to `w:m`, where `w` is the correct word corresponding to `m`. (If there is more than one correct word, we create one copy for each.) For example, for the posting `alogritm, docid=3, pos=7, score=0.7` we add the posting `algorithm:alogritm, docid=3, pos=7, score=0.7`. With the index enhanced in this way, we get the desired error-tolerant search via the prefix search and completion defined above. For example, the one-word query `algorithm*` now matches `algorithm` as well as `algorithm:alogritm`, and in this way matches all documents containing algorithm or any of its spelling variants.

It is important to note that we are getting another valuable feature besides the error-tolerant search here. Namely, for each query word we get a list of the spelling variants that actually occur in the hit set; see Figure 1. This is important, because it makes the error-tolerant search both *transparent* and *interactive*. For example, a user would find that the most frequent spelling variants in Figure 1 all look very reasonable, and she could click on any particular variant and check the corresponding hits.

Note that it is an option to lower the score of the occurrences of misspellings appropriately. Then documents with correct words would tend to be ranked before documents with misspellings, e.g. to alleviate the risk of false-positives. In an extreme case, these scores could be lowered to zero, which could be equivalent to removing all postings corresponding to misspellings from the collection. This has the advantage of not increasing the total number of postings (however, the increase is small, see Section 6.6), but the disadvantage of throwing away potentially meaningful words. Designing a good strategy of adjusting the scores of the additional postings is beyond the scope of this work.

5. SPELLING VARIANTS CLUSTERING

In this section we present our spelling variants clustering algorithm which, on a high level, does the following:

Step 1: Preprocessing

Filter out all garbage strings;

Compute a set V of words which are likely to be valid (e.g. using a trusted dictionary) and a set M of words which are not valid (non-words), not necessarily disjoint;

Step 2: Building clusters

Build an index for fast similarity searching on V and for each $w \in M$ find all $v \in V$ with $d(w, v) < \delta$;

For each $v \in V$ initialize an empty cluster C_v. Then for each v and all w with $d(w, v) < \delta$ set $C_v = C_v \cup \{w\}$

Step 3: Postprocess the clusters (see Section 5.3)

5.1 Preprocessing

Text collections typically contain a large number of garbage strings usually produced when an OCR device tries to scan certain non-text document sections as images, graphs, tables, etc. Our evidence shows that in some cases they can take as much as 50% of the lexicon. Since garbage strings can worsen the efficiency of our algorithm we clean them out by using a discriminator based on qualities of English words in contrast of meaningless strings generated by an OCR device. These rules were designed based on [18] and make use of word length, layout of vowels and consonants, consecutive repetitions of characters, number of digits, etc.

In this step we also determine the valid words in the lexicon, i.e., the words on which the error tolerant text search should be enabled. We use a simple method which consists of a lookup in a hash table of a large english dictionary of approximately 400,000 words based on word lists from the SCOWL project[1]. We note that one can be more "loose" and include a set of words which are not in the trusted dictionary. These words should be included in both sets, V and M. As a consequence the error-tolerant text search will be enabled for these words as well.

5.2 Building Clusters

The core of our cluster building process is a fast similarity searching algorithm. Given a set L of strings coming from an alphabet Σ, a query string q, and a distance function $d : \Sigma^* \times \Sigma^* \rightarrow \mathbb{R}$, a similarity searching algorithm finds all strings $s \in L$ such that $d(s, q) \leq \delta$, where δ is a distance threshold.

In our setting the set L overlaps with the set of valid words V, and the query strings q come from the set of non-words M. The algorithm we propose is based on normalized Levenshtein distance which is based on Levenshtein distance defined in Section 6.2. In Section 6 we give a short survey over existing distance functions which encourages our choice of a distance function.

Definition: (Normalized Levenshtein distance) *Normalized Levenshtein distance between strings s_1 and s_2 is defined as $\frac{ED(s_1,s_2)}{\max(|s_1|,|s_2|)}$, where* $\mathrm{ED}(s_1, s_2)$ *is the Levenshtein distance*

[1]http://wordlist.sourceforge.net

A normalized version of the Levenshtein distance (also proposed in [4]) puts more penalty on short strings and less penalty on long strings. Our motivation is to avoid assigning a large fixed threshold (e.g. 3 symbols) on short strings. According to [10] short words have more dense neighborhoods of similar words and are more difficult to correct. An older study given in [14] suggest that 43% of all miscorrections are generated from short words. In our setting this translates to very large clusters of short words which incurs additional overhead in the index size but also a lot of false-positive misspellings and potentially false search results.

5.2.1 Basic Algorithm: Permuted Lexicon

A permuted lexicon is a simple data structure which allows substring search queries to be answered by prefix queries as follows. For each string s we make $|s|$ copies by rotating the string $|s|$ times. For example the rotations of `variant` are `variant, ariantv, riantva, iantvar, antvari, ntvaria, tvarian`. Each rotations is represented by a integer pointer to the original string and another pointer for the rotation within that string (thus 5 bytes are more than sufficient). After sorting the permuted lexicon the original words will be ordered in buckets with respect to their common substrings (as opposed to common prefixes). We can efficiently find all strings that share a non-empty substring with a query string s by performing a binary search for each of its rotations. The approach originally proposed by Zobel et al. in [21] scans fixed size neighborhoods in the permuted lexicon that correspond to the rotations of a query string. The assumption is that most of the similar strings lie in these neighborhoods.

We propose a filtering algorithm based on permuted lexicons. A filtering algorithm consists of three phases: *indexing* or *preprocessing phase*, *candidate generation phase* and *verification phase* where candidate pair strings that have the potential of meeting the similarity threshold are verified against the distance function. Our algorithm is based on two filters together with probabilistic pruning to drastically reduce the number of distance computations done. In addition, to improve the accuracy, short words are processed in another phase of our algorithm.

5.2.2 Filter 1: Longest Common Substring

Let's consider two string s_1 and s_2 where s_2 is produced by introducing a single error in s_1. This means that the sequence of symbols in s_1 is at some point interrupted and as a result s_1 and s_2 will share two substrings. In the best case the longer substring has length $|s_1|$ and the in the worst case $\lceil \frac{|s_1|-1}{2} \rceil$ (when a deletion or a substitution error takes place in the middle of s_1). The longest substring however can not be shorter than this. Similarly, two errors will break s_1 into three parts and the longest substring will have at least $\lceil \frac{|s_1|-2}{3} \rceil$ symbols. Note that δ errors can break a string into at most $\delta + 1$ substrings.

Now let's consider strings as conceptually circular, i.e., an uninterrupted sequence of letters, where a substring from the end of the string continues at the beginning. δ errors break a circular string into at most δ parts instead of $\delta + 1$ (which means that the common substrings has to be longer).

Lemma 1: *Let the Levenshtein distance between two strings s_1 and s_2 be δ. Then the longest common substring cannot have less than $\lceil \frac{\max\{|s_1|,|s_2|\}-\delta}{\delta+1} \rceil$ symbols, accordingly $\lceil \frac{\max\{|s_1|,|s_2|\}-\delta}{\delta} \rceil$ when s_1 and s_2 are considered circular strings.*

Transposition errors affect two consecutive symbols instead of one. However, under the assumption that only one transposition error is allowed, transpositions make difference to

Lemma 1 only in the case when a single isolated transposition error takes place in a string with even number of symbols. This case is problematic only when the (absolute) threshold is 1. Strings with threshold of 1 (short strings) are processed in a different phase of our algorithm which does not rely on the lemma (see Section 5.2.5).

The aim is to filter out pairs of strings which do not meet the longest common substring constraint. Our implementation is based on circular strings since they make the constraint tighter. We combine this filter with the obvious length constraint, i.e., we apply it only if $||s_1| - |s_2|| \leq \delta$.

Note that we can make use of the normalized Levenshtein distance to get tighter constraint on shorter strings since the absolute threshold δ in Lemma 1 depends on the length of the longer string. By plugging the normalized Levenshtein distance one gets the following lower bound on the maximum common substring length:

$$\lceil \frac{\max\{|s_1|, |s_2|\} - \lfloor \delta_n \cdot \max\{|s_1|, |s_2|\} \rfloor}{\lfloor \delta_n \cdot \max\{|s_1|, |s_2|\} \rfloor} \rceil \qquad (1)$$

where $0 \leq \delta_n \leq 1$ is the normalized Levenshtein distance threshold. Note that we assume that a misspelling can not have more than 3 errors. This limits $\lfloor \delta_n \cdot \max\{|s_1|, |s_2|\} \rfloor$ in Equation 1 by 3.

Example: *Let* $|s_1| = 6$*,* $|s_2| = 7$ *and* $\delta_n = 0.28$*. Then the longest common (circular) substring between* s_1 *and* s_2 *must be at least* $\lceil \frac{\max\{7,6\}-1}{1} \rceil = 6$ *symbols long.*

Finding the longest common substrings of two strings s_1 and s_2 with a dynamic programming algorithm requires $\Theta(|s_1||s_2|)$ time, i.e., it costs equally as computing their Levenshtein distance. We make use of the permuted lexicon and apply Lemma 1 in this context.

We find all strings that share a non-empty substring (we call this a a bucket) with the query string by performing a binary search for each of its rotations. The strings with longer common substrings will lie closer to these locations. The algorithm then proceeds by scanning the strings below and above the current location in the bucket as follows. We compute the length of the common substring of the next string in the bucket by computing the common prefix of the corresponding rotations which can be done in constant time by precomputing the lengths of the common prefixes of each consecutive pair of rotations at the beginning of the algorithm. A string is considered a candidate only if it meets the constraint given by Equation 1 and pruned otherwise. Of course the current rotation does not need to share the longest common prefix (substring) with the current rotation of the query string but it can still be pruned as the rotation with the longest common substring is then in one of the next buckets. Once it is assured that the distance is below or above the threshold the string is marked as seen to avoid redundant computations. Note that from Equation 1 we can compute minimum acceptable longest common substring length of a string s of length $|s|$ to any other string and early terminate the scanning of the current bucket:

$$\min_i \lceil \frac{\max\{|s|, i\} - \min\{3, \lfloor \delta_n \cdot \max\{|s|, i\} \rfloor\}}{\min\{3, \lfloor \delta_n \cdot \max\{|s|, i\} \rfloor\}} \rceil \qquad (2)$$

where

$$\max\{|s| - 3, |s| - \lfloor \delta_n \cdot |s| \rfloor\} \leq i \leq \min\{|s| + 3, |s| + \lfloor \frac{\delta_n \cdot |s|}{1 - \delta_n} \rfloor\}$$

5.2.3 Filter 2: Intersection Size

While scanning the strings in a bucket we already compute the common prefix with the next permuted word. We can exploit this to further apply another filter as follows.

Table 1: Number of candidates which can potentially meet the similarity threshold after applying each filter. The experiment is to find all similar non-words for each valid in the DBLP dataset of approximately 1.5M words.

Filter	/	LCS	Prune	Intersection size
Candidate set size	212M	61.9M	29.7M	6.97M

Lemma 2: *Let the Levenshtein distance between two strings* s_1 *and* s_2 *(including transpositions) be at most* δ *and let* $M(s_1)$ *and* $M(s_2)$ *be the multisets of the symbols in* s_1 *and* s_2*. Then* $|M(s_1) \cap M(s_2)| \geq \max\{|s_1|, |s_2|\} - \delta$*.*

This lemma is special case of a well known property which has originally appeared in the literature in [16]. This filter is applied only if the current permuted query word and the next permuted word in the bucket share long enough prefix. Then we only need the size of the multiset intersection of the suffixes which can be computed in time proportional to the suffix length of the candidate string by accumulating the symbol counts of the query string into an array at the beginning of the query execution. Since the suffixes are short this is not very expensive. Note that transposition errors do not affect Lemma 2.

Table 1 shows the filtration effect, i.e., the number of candidate pairs that have the potential of meeting the similarity threshold after applying each filter.

5.2.4 Probabilistic Pruning

For a small sacrifice (e.g. 5%) of the accuracy and additional improvement of the efficiency we can prematurely stop the scanning of a certain bucket and decrease the number of distance and filter computations done. A simple threshold on the number of scanned strings in each bucket as in [21] is inappropriate as some buckets require covering much more strings than other. We employ a heuristic which allows the number of similar strings found to remain close to 100%. The stopping criterion is based on upper bound on the number of consecutive strings with normalized Levenshtein distance above the threshold which are encountered while scanning a bucket. For a given accuracy (e.g. 95%) we estimate this upper bound by probing and sampling from the lexicon. Due to space limitation we omit further technical details.

5.2.5 Short Words

Since similar short strings can have small common substrings there is the potential danger of being missed by the premature stopping of the algorithm. Therefore we processed them in another phase as follows. As short we define all strings s such that there is no string s_1 longer than 1 symbol with $\mathrm{ED}_n(s, s_1) \leq \delta_n$, where $\mathrm{ED}_n(.,.)$ is the normalized Levenshtein distance. This is equivalent to the constraint $2/(|s| + 2) > \delta_n$ or $|s| \leq \lfloor 2/\delta_n - 2 \rfloor$. For example if $\delta_n = 0.28$, then as short are defined all strings that have at most 5 symbols. Since short strings satisfying the above constraint permit only one error we can use a specialized fast algorithm adapted for this setting (e.g. [12], [17]). Currently we are using a simple neighborhood[2] generation paired with dictionary look ups to find all similar string in the dictionary. This phase usually takes a small fraction of the total time of our main algorithm.

[2]All words that can be obtained by a single substitution, insertion, deletion and transposition of symbols

5.2.6 Complexity

Our algorithm requires $O(mN)$ space where N is the number of valid words in the dataset and $(m \approx 10)$ is the average string length. Trivially, the average query complexity of our algorithm is $O(m^3 \log N)$ since we need m binary searches and a number of distance computation per each rotation. As for the constant factors, the average number of distance computations per rotation on English words is below 1, meaning that in average we do less than m distance computations per query. On long (e.g. 20 symbols) and random strings the number of distance computations per rotation is below 0.1 which means that our algorithm becomes more efficient. This is expected as the number of correction candidates decreases and the filtering effectiveness increases.

5.3 Post-Processing

At the end of the similarity searching step, each cluster is preprocessed as follows. First, false-positive misspellings are post-filtered using a spelling-correction technique and second, non-words which are not similar to any valid word are checked for word-boundary errors and assigned to clusters.

5.3.1 Post-Filtering

To eliminate false-positive misspellings we combine insights from the noisy channel model for spelling-correction proposed in [3] and insights from the EM fitting algorithm given in [15]. Both of them can be applied to our problem and both have shown good results in the literature.

Consider an intended word - misspelling pair w, m. The model works by learning generalized edits of the form $\alpha \rightarrow \beta$ together with their probabilities, where α is a substring of w and β is a substring of m. The probability $P(w|m)$, that m is a misspelling of w is modelled as the maximum probability over all finite sequences S of generalized edits (including null-edits) that transform w to m:

$$\max_{S:S(w)=m} \prod_{\alpha \rightarrow \beta \in S} p(\alpha \rightarrow \beta) \tag{3}$$

The main problem is to correctly estimate the probabilities $p(\alpha \rightarrow \beta)$. The approach proposed in [3] is intricate to implement since it either has to rely on heuristics or it requires a large corpus of text together with all spelling variants identified. A more elegant EM algorithm for single edit operations that only requires a collection of spelling-variant pairs is proposed in [15]. We extended this algorithm to tackle generalized edits.

The clusters are post-filtered by re-ranking the valid words which they correspond to using Equation 3 and then choosing the best k clusters for each misspelling, where k is small (usually 2).

Note that if the collection comes from OCRed text we leave out the post-filtering step even though we imply that a similar technique can be used to fit a symbol confusion matrix (instead of sequences of edits).

5.3.2 Word-Boundary Errors

Solving the word-boundary problem requires computing all partitions of a string such that its components are valid words and then choosing the most likely partition [10]. This results in a combinatorial explosion in the number of possible partitions that must be checked. Nevertheless, we have found that in some OCRed texts these errors can take up to 25% of all lexicon terms, whereas in human-typed text up to 5%.

We approach both problems greedily and compute reasonable approximations. For briefness technical details are skipped. We note that this step takes negligible time in our algorithm.

Table 2: Characteristics of our six datasets. The last three columns give the size of the lexicon, the number of valid words, and the average word length, respectively (garbage strings has been cleared out).

Dataset	Size	# terms	# valid	avg len
20 Newsgroups	35 MB	140,559	40,920	13.1
MPII	235 MB	507,518	43,832	10.9
DBLP	1 GB	913,869	68,702	15.1
Wikipedia	4.9 GB	4,395,331	208,139	9.4
TREC Terabyte	164 GB	10,120,946	230,233	10.1
Artificial	-	1,000,000	50,000	22.2

6. EXPERIMENTS

We compared our algorithms on six datasets ranging from small to large and with different kinds of spelling errors. We note that for the first three dataset we carry out quality experiments and for the last three efficiency experiments.

6.1 Datasets

20 Newsgroups is a small collection of approximately 20,000 newsgroup documents, partitioned across 20 different newsgroups. It contains human-typed text only.

MPII are the about 50,000 web pages of the Max Planck Institute for computer science and it reflects the usual web mess. It consists of both, human-typed text (including foreign languages) and OCRed text.

DBLP contains about 50,000 computer science articles, many of them OCRed.

Artificial We generated an artificial lexicon containing random strings with lengths coming from uniform $U(3, 40)$ distribution. The purpose was to evaluate the algorithms on longer strings where a distance computations is more expensive and filtering effectiveness more important.

Wikipedia is the (November 2007 dump of the) English Wikipedia with about 3 million documents.

Terabyte is our largest collection, the standard TREC .GOV collection with about 25 millions documents.

Details for each dataset are shown in Table 2.

6.2 Distance Functions

We considered the following distance function in our study:

Levenshtein distance (modified) *between two strings s_1 and s_2 is the minimum number of edit operations required to transform s_1 to s_2. The edit operations include insertions, deletions, substitutions and transposition of characters.*

Jaccard similarity *is defined as $J(s_1, s_2) = |\frac{s_1 \cap s_2}{s_1 \cup s_2}|$ where s_1 and s_2 are the sets of Q-grams of the corresponding strings.*

Ukkonen's Q-gram distance *is defined as $Ukk(s_1, s_2) = \sum_{q \in s_1 \cup s_2} |w(q) - m(q)|$ where $w(q)$ and $m(q)$ indicate how many times the Q-gram q appears in s_1 (s_2)*

To experimentally assess which distance functions is the most appropriate for our problem we compiled random subsets from three of our datasets (see Section 6) and produced a spelling variants clustering for each of them (excluding any post-processing). For each distance function we chose a threshold such that the average precision was more or less fixed, i.e., close to 0.8 and then measured the recall of each cluster. The average recall is shown in Table 3.

The main observation is that the Levenshtein distance gives the best recall for reasonably high precision. From

Table 3: Average recall for the computed spelling variants clustering for each distance function. The thresholds are chosen such that the precision is approximately 0.8

Distance function	20 Newsgroups	MPII	DBLP
Levenshtein	**0.97**	**0.94**	**0.92**
Jaccard	0.90	0.78	0.61
Ukkonen's	0.87	0.80	0.59

Table 3 it is clear that this effect is more prominent for OCRed text as most of the errors in the DBLP dataset are OCR-induced.

6.3 Algorithms

All algorithms in this section were written in C++ and compiled with GCC 4.1.2 with -O3 flag. All experiments were performed on a machine with 16 GB of main memory, 4 dual-core AMD Opteron 2.8 GHz processors (but we used only one core at a time), operating in 32 bit mode and running Debian 4.1.1-19. For each of the approaches discussed in Section 2, we picked the most competitive (off the shelf) representative for each of the techniques discussed there.

- **AESA** We were interested in how well a metric space based searching algorithm performs in our problem setting.
- **LSH** is probabilistic signature-based algorithm for fast similarity searching. We include it in our experiments for efficiency comparison only.
- **Trie** We optimize the trie-based algorithm by the fact that the queries are known in advance so that they can be initially sorted. Then savings come based on potentially large common prefixes of consecutive queries.
- **Q-gram index** is a Q-gram indexing based algorithm exploiting the Q-gram overlap constraint.
- **FastSS** is an efficient algorithm based on deletion neighborhood to model the Levenshtein distance. We use an implementation from the *Metric Spaces library*[3] written in C.
- **Permuted, FastPermuted** Respectively the basic[4] permuted lexicon approach from [21] (i.e. no filters) and our own permuted lexicon approach. For both we set the accuracy to 95%.

6.4 Clustering Efficiency

Table 4 shows the running times of ours and all compared methods on our three largest datasets for the task of the spelling variants clustering problem.

Our algorithm outperforms all of the compared algorithm in both running time and number of distance computations performed. Clearly, the reason for this is the small computational cost of FastPermuted. This is achieved via its careful tradeoff between the effectiveness of its filters and their cost, as appropriate for our setting with a relatively large number of relatively short strings but also via exploiting the normalized Levenshtein distance. The closest competitor is FastSS with more than twice the running time of our algorithm. However, FastSS pays for its high speed with a space usage more than ten times higher than for our method. For our artificial lexicon with 50,000 strings set as valid, which even the $\Theta(n^2)$ AESA algorithm could handle, FastSS required more than 4 GB of memory which was infeasible on our 32-bit machine.

On random (and long) strings, our algorithm could easily compute the exact answer with very small computational effort. This is not surprising as first, filtering effectiveness becomes much better, and second, the length of common substrings among strings is then minimized and no probabilistic pruning is required.

[3]http://sisap.org/?f=library

[4]To achieve the desired accuracy with the basic Permuted algorithm we had to process the short words in the same way as for FastPermuted

Table 4: Clustering efficiency results. The first entry for each dataset corresponds to the running time (in seconds) and the second corresponds to the number of (millions of) distance computations (if applicable). The LSH method works with Jaccard distance. Some of the methods were space infeasible.

Dataset	Wikipedia		Terabyte		Artificial	
FastPermuted	**399**	**47.8**	**963**	**117**	**21**	0.21
Permuted	1303	1159	3498	2820	161	48.6
Q-gram index	4876	257	13K	807	67	**0.18**
Trie based	12K	-	37K	-	4547	-
FastSS	904	-	2241	-	-	-
AESA	-	-	-	-	24K	10
LSH*	866	218	2209	443	81	0.24

Table 5: Accuracy of the clustering produced by our method for three of our datasets. A distance threshold of 0.28 which gave the best tradeoff between quality, efficiency and index space overhead.

Dataset	Precision	Recall	F-measure
20 Newsgroups	0.950	0.953	0.951
MPII	0.954	0.918	0.938
DBLP	0.901	0.890	0.894

6.5 Clustering Quality

Table 5 summarizes the quality results of our spelling-variants clustering. Manually computing spelling-variant clusters on large datasets like Wikipedia or Terabyte required prohibitive effort and time. The results shown are the average precision and recall for each dataset measured against 100 manually computed clusters as explained in Section 6.2.

The best quality results were achieved on the 20 Newsgroups dataset as it contains human-made errors only. The MPII and the DBLP datasets contain both human-made and OCR errors which are harder to detect and correct and in turn the clustering quality was slightly worse. To our knowledge most of the spelling correction approaches are aimed towards human-made errors and quality results on OCRed texts are usually not shown.

We note that one can sacrifice some index space overhead (and potentially some precision) by choosing a higher distance threshold and achieve a better recall on OCRed texts. Finally we note that we did not lose any quality as a consequence of our approximate (and not exact) similarity searching method. In fact, an exact similarity searching algorithm produced the same clustering in most of the cases.

6.6 Search Efficiency

Table 6 compares the performance of our error-tolerant search, as described in Section 4, with the same feature realized by the state of the art Lucene search engine. The

Table 6: Average query processing time. The three entries of CompleteSearch correspond to average query processing time without tolerance, with tolerance and error-tolerant text search based on vanilla implementation of disjunctive search

Dataset	CompleteSearch			Lucene
DBLP	32.7 ms	35.5 ms	1225 ms	776 ms
Wikipedia	205 ms	230 ms	5093 ms	6148 ms
TREC Terabyte	3414 ms	3630 ms	≈1 min	≈20 min

Table 7: Index and compressed vocabulary size with and without error-tolerant text search.

Dataset	Index size		Lexicon size	
DBLP	450 MB	532 MB	4.0 MB	5.2 MB
Wikipedia	2.24 GB	2.36 GB	26 MB	30 MB
TREC Terabyte	41.1 GB	41.9 GB	71 MB	80 MB

Table 8: Average number of hits with and without the error-tolerant text search feature. The left number of each entry corresponds to low-recall ($\leq$ 50 hits) queries and the right number corresponds to all queries

Dataset	DBLP		Wikipedia		Terabyte	
Ordinary	20.8	1246	16.4	17.2K	22.5	106.0K
Error-tolerant	23.3	1256	22.1	17.3K	25.2	106.1K

entries in the table correspond to average execution time over 20,000 queries ranging from 1 to 4 terms. The results show that the disjunctive approach of Lucene incurs an enormous blowup in query processing time. In contrast, our approach of adding the valid words to the index and accessing them via CompleteSearch's efficient prefix search achieves the same feature at essentially no cost in query processing time. We note that Lucene's ordinary query processing times are comparable to those of CompleteSearch (in fact somewhat faster because Lucene employs top-k techniques, which CompleteSearch, due to its extended search facilities cannot easily do), that is, Lucene's slow error-tolerant search is indeed due to the inherent complexity of the disjunctive approach.

The price paid for our fast error-tolerant text search is the small overhead in index size due to the additional artificial postings in the index. Table 7 summarizes the overhead of CompleteSearch on three of our biggest datasets.

6.7 Search Quality

As shown in Table 8, error-tolerant search increases recall by 10-30 % for queries with 50 or less hits. Such low-recall queries are the most interesting ones for error-tolerant search; recall the literature search example from the introduction. For the remaining higher-recall queries, the increase in recall is less than 1%. Given the high accuracy of our spelling variants clustering (Section 6.5), the precision of the additional results will be about the same as the precision of the results that would have been returned without error-tolerant search. That is, the error-tolerant search increases recall without sacrificing much precision. A manual investigation has indeed confirmed this. A similar conclusion has been drawn in [19].

7. CONCLUSIONS AND FUTURE WORK

We have defined the spelling variants clustering problem, and presented a new and very fast algorithm for its solution. We have shown how to use this solution for error-tolerant full-text search with little extra cost, in neither query processing time nor index size.

A weakness of our approach is that it relies on a dictionary of valid words. An alternative would be to use supervised learning to tell valid words from misspellings, as proposed in [7]. The net result would be a widened coverage of valid words at the price of a somewhat decreased accuracy.

We did not consider the case of so-called "real-word errors", that is, one valid word accidentally used in place of another, for example, piece instead of peace. Other works consider this by looking at the context of the surrounding words. It is an open problem, however, how to achieve this at the high efficiency we were aiming at in this paper.

8. REFERENCES

[1] H. Bast and I. Weber. Type less, find more: fast autocompletion search with a succinct index. In *SIGIR '06*, 2006.

[2] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *WWW '07*, pages 131–140, 2007.

[3] E. Brill and R. C. Moore. An improved error model for noisy channel spelling correction. In *ACL'00*, 2000.

[4] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *ICDE '06*, page 5, 2006.

[5] X. L. Chuan Xiao, Wei Wang and J. X. Yu. Efficient similarity joins for near duplicate detection. In *WWW 2008*, 2008.

[6] E. Cohen, M. Datar, S. Fujiwara, A. Gionis, P. Indyk, R. Motwani, J. D. Ullman, and C. Yang. Finding interesting associations without support pruning. *ICDE'00*, page 489, 2000.

[7] D. C. Comeau and W. J. Wilbur. Non-word identification or spell checking without a dictionary. *JASIST*, 55:169–177, 2004.

[8] H. Dalianis. Evaluating a spelling support in a search engine. In *NLDB '02*, pages 183–190, 2002.

[9] K. Figueroa, E. Chávez, G. Navarro, and R. Paredes. On the least cost for proximity searching in metric spaces. In *WEA*, pages 279–290, 2006.

[10] K. Kukich. Technique for automatically correcting words in text. *ACM Comput. Surv.*, 24:377–439, 1992.

[11] S. Mihov and K. U. Schulz. Fast approximate search in large dictionaries. *Comput. Linguist.*, pages 451–477, 2004.

[12] R. Muth and U. Manber. Approximate multiple string search. In *CPM'96*, pages 75–86, 1996.

[13] G. Navarro and R. Baeza-yates. Searching in metric spaces. *ACM Comput. Surv.*, pages 273–321, 2001.

[14] J. J. Pollock and A. Zamora. Automatic spelling correction in scientific and scholarly text. In *Commun. ACM 27, 4 (Apr.)*, pages 358–368, 1984.

[15] E. S. Ristad and P. N. Yianilos. Learning string-edit distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20:522–532, 1998.

[16] E. Sutinen and J. Tarhio. Filtration with q-samples in approximate matching. In *CPM'96*, pages 50–63, 1996.

[17] B. S. T. Bocek, E. Hunt. Fast Similarity Search in Large Dictionaries. Technical report, Department of Informatics, University of Zurich, 2007.

[18] K. Taghva, J. Borsack, and A. Condit. An expert system for automatically correcting ocr output. In *SPIE*, pages 270–278, 1994.

[19] K. Taghva, J. Borsack, and A. Condit. Results of applying probabilistic IR to OCR text. In *Research and Development in Information Retrieval*, pages 202–211, 1994.

[20] K. Taghva, J. Borsack, and A. Condit. Effects of ocr errors on ranking and feedback using the vector space model. *Inf. Process. Manage.*, 32:317–327, 1996.

[21] J. Zobel and P. W. Dart. Finding approximate matches in large lexicons. *Software - Practice and Experience*, 25:331–345, 1995.