University of Zurich
Department of Informatics

Thomas Bocek
Ela Hunt
Burkhard Stiller

Fast Similarity Search in Large Dictionaries

TECHNICAL REPORT – No. ifi-2007.02

April 2007

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

T. Bocek, E. Hunt, B. Stiller:
Technical Report No. ifi-2007.02, April 2007
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zurich, Switzerland
URL: http://www.csg.uzh.ch

Fast Similarity Search in Large Dictionaries

Thomas Bocek
Communication Systems Group
University of Zurich
Switzerland
bocek@ifi.uzh.ch

Ela Hunt
GlobIS
ETH Zurich
Switzerland
hunt@inf.ethz.ch

Burkhard Stiller
Communication Systems Group
University of Zurich and
TIK
ETH Zurich
Switzerland
stiller@ifi.uzh.ch

April 2007

Abstract

Fast similarity search is important for time-sensitive applications. Those include both enterprise and web scenarios, where typos, misspellings, and noise need to be removed in an efficient way, in order to improve data quality, or to find all information of interest to the user. This paper presents a new algorithm called Fast Similarity Search (FastSS) that performs an exhaustive similarity search in a dictionary, based on the edit distance model of string similarity. The algorithm uses deletions to model the edit distance. For a dictionary containing n words of average length m, and given a maximum number of spelling errors k, FastSS uses a deletion dictionary of size $O(nm^k)$. At search time each query is mutated to generate a deletion neighborhood of size $O(m^k)$, which is compared to the indexed deletion dictionary. As a deletion neighborhood is smaller than a neighborhood using deletions, insertions and replacements, this contributes to a faster search. FastSS looks up misspellings in a time which is independent of n for a hash-based index, or logarithmic in the size of the dictionary, for a tree-based one.

FastSS has been evaluated and compared with NR-grep, a keyword tree, dynamic programming, n-grams, and neighborhood generation

algorithms, using an English and a random dictionary. All results show that FastSS outperforms other algorithms with respect to the search time and is a perfect candidate for time-critical applications requiring approximate keyword searching. A web application compares n-grams and FastSS in searching English Wikipedia articles and meta pages, and times FastSS in *Moby Dick* [20].

Keywords: Edit distance, approximate string matching, indexing for text, IR, deletion neighbourhood, n-grams, fast similarity search, neighborhood generation.

1 Introduction

The increasing amounts of electronic information available on the web and within enterprises call for fast similarity searches. A similarity search algorithm should provide a list of data similar to an input query. Similarity searches in time that is linear in the dictionary size may be too slow for many applications, and therefore, it is important to focus on sub linear similarity search algorithms. Such algorithms rely on indexing, as this is the only way to ensure acceptable performance. Search engines like Google or Yahoo offer fast similarity searches for large amounts of data found in the World Wide Web. In the context of search engines, the terms document similarity and query string similarity have to be distinguished. Document similarity refers to the overall similarity of an entire document to the query, while string similarity concerns just any two strings. The new algorithm, Fast Similarity Search (FastSS), belongs to the second category.

A query string similarity search example is Google's or Yahoo's "Did you mean" link, that suggests a more popular query string. This approach is based on machine learning on the query string [13]. An evaluation of spelling support tools [16] shows that approximately 10% of all queries are not found, because of typos or misspellings. In Intranets and content management system solutions, such as Joomla! [5], Zope [10] or TYPO3 [9], search engines are deployed without similarity search algorithms. This is because algorithms based on statistical analysis may not be applied, as the actual query strings are not produced in sufficient quantities. However, fast similarity searches are a desirable application feature, and spelling correction tools would help users improve the way they express their information need.

The edit or Levenshtein distance [19] can be used to measure the distance between two sequences of symbols without using statistical analysis.

However, the time complexity of the dynamic programming approach for searching similar words in a dictionary is $O(nm^2)$, where m is the average word length, and n the dictionary size. The goal of this work was to develop a fast sub linear similarity search algorithm using the edit distance metric.

The main idea of FastSS consists in using an efficient variant of the neighborhood generation algorithm [21, 12, 18, 29], adapted to use deletions only. This produces a smaller neighborhood which is then looked up in the index. The key findings are that neighborhood generation, using deletions only, performs a lookup in $O(km^k \log(nm^k))$, with a space usage of $O(nm^k)$, where k is the number of edit operations. A variant of FastSS, Fast Block Similarity Search (FastBlockSS), reduces the space usage even further by splitting sequences into b-grams and by removing redundant b-grams.

Experiments presented in this paper were based on an English dictionary and a randomly generated dictionary, and compared search performance for NR-grep [24], dynamic programming, a keyword tree, neighborhood generation, and n-grams with index lookup. For both dictionaries, all three FastSS variants outperform any other algorithm with respect to the search time. A demonstration server [27] for FastSS finds in an indexed version of *Moby Dick* [20] and in an indexed version of the English Wikipedia and meta pages words similar to those submitted as a query. The test application uses FastSS to search in an index stored in a database and displays the results in the order of retrieval. The experimental evaluation shows the superiority of FastSS over all other methods in both in-memory and database settings.

Contributions. The first contribution of this paper is applying deletions to solve the general edit distance problem for dictionaries. The second contribution is a practical demonstration of the feasibility of this approach.

The rest of this paper is structured as follows. Section 2 discusses related work, while Section 3 introduces the details of the new algorithm. Section 4 provides an experimental evaluation. Section 5 shows demonstrator applications using the new algorithm. Driven by the discussion in Section 6, conclusions are drawn in Section 7.

2 Related Work

The review that follows focuses on exhaustive approaches, with the exception of the discussion of indexes to misspellings. The following exhaustive algorithms are discussed: edit distance, NR-grep, n-grams and cosine similarity, keyword trees and suffix indexes, and neighborhood generation.

2.1 Edit Distance

Edit distance, ED [19], is the minimum number of operations required to transform one string into another, with operations being a deletion, an insertion or a replacement. For example, ED(*test*,*fest*) = 1, requiring one replacement of the first *t* in *test* with an *f*. ED calculation uses dynamic programming, DP, and a matrix d of size $(|s1|+1)(|s2|+1)$, where $|s1|$ and $|s2|$ are the lengths of strings *s1* and *s2* , and $i = 0..|s1|$, $j = 0..|s2|$. d is defined as follows.

$$\begin{aligned} d[i,0] = &\ i, \\ d[0,j] = &\ j, \\ d[i,j] = &\ min(d[i-1,j]+1, \\ &\ d[i,j-1]+1, \\ &\ d[i-1,j-1]+(if\ s1[i]=s2[j]\ then\ 0\ else\ 1)) \end{aligned}$$

Table 1 shows a matrix for *test* and *fest*. The values in bold show a minimum edit cost path. The time complexity of DP is $O(|s_1||s_2|)$. A simple algorithm searching a dictionary for similar words using ED will iterate over all dictionary words and use DP to align every word to the query. This approach scales linearly with dictionary size, and, for large dictionaries, is slow. Faster algorithms are described in the following.

Table 1: Example DP calculation matrix

		f	e	s	t
	0	1	2	3	4
t	1	**1**	2	3	3
e	2	2	**1**	2	3
s	3	3	2	**1**	2
t	4	4	3	2	**1**

2.2 NR-grep

Navarro's NR-grep [24] is an exhaustive online similarity search algorithm. NR stands for non-deterministic reverse pattern matching, where non-determinism refers to the pattern automaton used. NR-grep uses bit-parallelism and forward and backward searching. It splits the query of length m into words, to match register word size w. The number of such words is $\lceil m/w \rceil$. This allows for efficient approximate matching by using a strategy for query splitting and result assembly, combined with automata

accepting words with up to a certain number of errors. NR-grep uses ED enhanced with a swap operation, as misspellings often include transpositions. Algorithm complexity is linear in the size of the target data, but, overall, significantly better than other linear approaches.

2.3 N-grams and Cosine Similarity

An n-gram index of a target dataset is created by sliding a window of length n over the data and noting the content and position of all such windows. N-grams can be overlapping or non-overlapping and represent an offline approach, with the index being either memory- or disk-resident. As a similarity metric one can use the number of shared n-grams in the query and the index. An example target *abracadabra*, split into 3-grams, produces *(abr,1), (aca,4), (dab,7)*, and *(bra,9)*. With a query *bbracadabra*, having a 3-gram list *(bbr,1), (aca,4), (dab,7), (bra,9)*, ED is 1, as only the first two n-grams, *abr* and *bbr* differ in one letter. Strings *s1* and *s2*, with *ED=k* share at least $|s1| - n + 1 - kn$ n-grams, which provides a filter, and requires a verification phase [25].

An extension of this approach for large text collections uses cosine similarity [23], which is a global measure. All n-grams present in the target data are represented as a vector of their frequencies. A query is decomposed into a n-gram vector of the same dimension, and a lookup in the index follows. Similarity is measured by the scalar product (cosine) of these two vectors i and j:

$$\cos(i, j) = \frac{\vec{i} \cdot \vec{j}}{|i||j|}$$

A similar approach is used in DB2 Net Search Extender [3] and Apache Lucene [1].

2.4 Keyword Trees and Suffix Indexes

Tree-shaped indexes in combination with DP can be used to find similar words under ED model [28, 15, 17, 12, 18]. The complexity of search for a word of length m over an alphabet of size s with k mismatches in suffix trees without suffix links is $O(s^{(m+k)})$, as the tree is traversed to the depth $m + k$ and each node can have up to s children. This figure is smaller if suffix links are used and search is terminated as soon as one can tell that a match is not possible. Since a tree encodes only the strings present in the data, the actual search space is often much smaller than the theoretical bound.

2.5 Search with Neighborhood Generation

Search with sub linear complexity was proposed by Myers [21], with focus on DNA sequences consisting of letters *A,C,G,* and *T*. The target data are indexed using n-grams arranged into bins, and hashing maps each word to an integer. A query is split into words of length q. For each such word a neighborhood of all words with a given ED is generated and the candidate words are fetched from the index, and assembled into longer matches. Neighborhood generation works well with low k and small alphabets [26].

2.6 Indexing Misspellings

Although seemingly inefficient, this is what actually happens in search engines which show to the user an alternative spelling of the query term, but are based on data mining and not on exhaustive search. In FLASH [14] patterns with misspellings are indexed, and matching is based on statistics. FLASH indexes proteins and uses heuristics to deliver fast matching of high sensitivity. It could be said that FastSS follows a similar approach, by indexing words with deletions, but with guarantees of exhaustivity. To our knowledge, such an approach has not been reported on before, and next section explains FastSS in detail.

2.7 Related Work Overview

Algorithms for similarity searching can be categorized as online or offline, exhaustive or heuristic, and using a global or a local metric of similarity. Online algorithms search without pre-processing the target data, and need to traverse all data during the search. Offline algorithms pre-process the target data and may store it in memory or on disk to speed up query processing. In contrast to exhaustive algorithms which guarantee to find all occurrences of the query in the target, heuristic algorithms may not find all similar data. In heuristics, a reduction of the search time is achieved by evaluating only the statistically interesting patterns. However, heuristics are not suitable where all similar data needs to be found. Global metrics measure the similarity of all target data to the query, while local metrics measure the similarity between some part of the target and the query.

In this section all presented algorithms are exhaustive and based on the local similarity model. Edit distance and NR-grep are online algorithms, n-grams and cosine similarity, keyword trees and suffix indexes, and neighborhood generation are offline algorithms. FastSS is an exhaustive, offline search, based on the local similarity model.

3 Fast Similarity Search

This subsection motivates FastSS with selected examples and then presents the newly developed algorithm in detail. Then, three variants of FastSS are presented.

3.1 Examples

The following five examples demonstrate the basic concept of FastSS.

3.1.1 Example: Deletion

Expression *d(w,p)* stands for the transformation of word w by deletion of the letter at position p, *d(test,1)=est* and *d(test,2)=tst*.

3.1.2 Example: Indexing

For a given maximal number of deletions k, the results of all possible deletions in a given word are indexed. Deletions are applied recursively, and indices of deleted letters are ordered. For a word *super* with $k = 2$ the following relationships will be indexed. Zero deletions produces *super →(super)*; one deletion produces *uper → (super,1)*, *sper→(super,2)*, *suer → (super,3)*, *supr → (super,4)*, and *supe → (super,5)*; and two deletions produce for each of the words with one deletion four variants, for instance for *uper → (super,1)*, adding a deletion produces *per → (super,1,1)*, *uer → (super,1,2)*, *upr → (super,1,3)* and *upe → (super,1,4)*.

3.1.3 Example: Single Deletion

Figure 1 shows that one deletion produces strings with edit distance=1, since *east* can be transformed to *est* by deleting *a*.

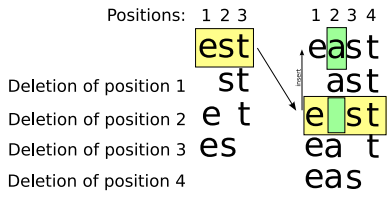

Figure 1: Deletion of *a* at position 2 in *east* produces *est*.

3.1.4 Example: Two Deletions, Same Position

Figure 2 shows that *d(test,1)* = *d(fest,1)* = *est*. A replacement *t*→*f* at position 1 corresponds to two simultaneous deletions, one in the query *test* and the other in the target *fest*, and models one replacement, with ED=1.

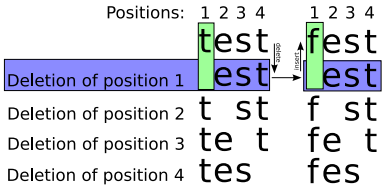

Figure 2: Replacement *t*→*f* at position one of *test*, to generate *fest*, modelled via deletions.

3.1.5 Example: Two Deletions, Different Positions

Figure 3 shows that two deletions at two different positions, *d(test,1)* and *d(east,2)* can produce the same effect, *est*, but in this case ED(*test,east*)=2, modeling a composition of two edit operations.

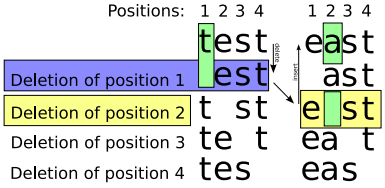

Figure 3: Deletion of *t* at position one of *test*, and deletion of *a* at position 2 in *east* produce the same result *est*.

3.2 Formalization

A general form of applying the recursion for a k-deletion neighborhood U_d of word v is:

$$U_d(v,k) = \bigcup_{g=1}^{|v|} U_d(d(v,g), k-1).$$

Theorem 1. *For strings s_1, s_2, character position x and recursion depth r, if $d_r(s_1, x_r) = s_2$ then $ed(s_1, s_2) = r$. (see example in section 3.1.3)*

Proof. A deletion is an edit distance operation and performing r deletions produces edit distance r. □

Theorem 2. *For strings s_1, s_2, character position x and recursion depth r, if $d_r(s_1, x_r) = d_r(s_2, x_r)$ then the edit distance is $ed(s_1, s_2) = r$. (see example in section 3.1.4)*

Proof. Performing r deletions at the same position in two strings corresponds to r character substitutions between the strings, resulting in edit distance of r. □

Theorem 3. *For the strings s_1, s_2, character positions x, y, where $x \neq y$, and recursion depth r, if $d_r(s_1, x_r) = d_r(s_2, y_r)$ then $ed(s_1, s_2) = 2r$. (see example in section 3.1.5)*

Proof. First, r deletions are performed to transform s_1 into $d_r(s_1, x_r)$ and then r inserts to transform $d_r(s_1, x_r)$ into s_2, using $2r$ edit operations. □

Theorem 4. *Given two sorted lists of delete positions x, y, two equal strings resulting from the application of those deletions, $f(x, y)$ as defined below, $x = [x_0, .., x_n]$, $hd(x) = x_0$ and $tl(x) = [x_1, .., x_n]$, $f(x, y)$ is equivalent to the edit distance.*

$$f(x,y) = \begin{cases} 0 & : \quad x = [], y = [] \quad (1) \\ |x| & : \quad x \neq [], y = [] \quad (2) \\ |y| & : \quad x = [], y \neq [] \quad (3) \\ 1 + f(tl(x), y) & : \quad hd(x) < hd(y) \quad (4) \\ 1 + f(x, tl(y)) & : \quad hd(x) > hd(y) \quad (5) \\ 1 + f(tl(x), tl(y)) & : \quad hd(x) = hd(y) \quad (6) \end{cases}$$

Proof. (1)-(3) are stop conditions. (2) and (3) are explained by Theorem 1. In (1) ED=0, as there are no delete positions to consider. (4) and (5) are explained by Theorem 3, define the induction steps, and apply if a delete position in x is not present in y and vice versa. (6) is explained by Theorem 2 and applies if a delete position is present in both x and y, which denotes a replacement. □

3.2.1 Indexing

For all words in a dictionary, and a given number of edit operations k, FastSS generates all variant spellings recursively and save them as tuples of type $v\prime \in U_d(v, k) \rightarrow (v, x)$ where v is a dictionary word and x a list of deletion positions.

Theorem 5. *Index uses $O(nm^{k+1})$ space, as it stores all the variants for n dictionary words of length m with k mismatches.*

Proof. There are $C_m^k = \binom{m}{k}$ ways of deleting k letters out of m. This is equivalent to $O(m^k)$. Since there are n words to index with word length m, index size is $O(nm^k)$. □

3.2.2 Retrieval

For a query p and edit distance k, first generate the neighborhood $U_d(p, k)$. Then compare the words in the neighborhood with the index, and find matching candidates. Compare deletion positions for each candidate with the deletion positions in $U_d(p, k)$, using Theorem 4.

Theorem 6. *If the deletion dictionary is accessed using hashing, search complexity is $O(km^k)$.*

Proof. Neighborhood generation requires $O(m^k)$ time. A hashed data structure finds all $O(m^k)$ candidates in $O(m^k)$ time. For each candidate two sorted deletion lists of size k are compared. Thus the search complexity is $O(km^k)$. □

Theorem 7. *Search complexity is $O(km^k \log(nm^k))$ if the deletion dictionary is stored in an index with logarithmic access time.*

Proof. Neighborhood generation requires $O(m^k)$ time. A tree index needs $O(\log(nm^k))$ time to find all candidates. A deletion list comparison needs $O(k)$. Thus the search complexity is $O(km^k \log(nm^k))$. □

3.2.3 Generalization

In a natural language, such as English, m can be seen as constant, as the longest word in the Oxford English Dictionary contains 30 letters. With m and k constant, lookup time with hashed index access is $O(1)$ and with logarithmic access is $O(\log(n))$. The following sections present this algorithm and two variants based on the above concepts.

3.3 FastSS

Figure 4 shows the pseudo code for FastSS. The index *Index* stores all word variants with deletions and positions of deletions for each variant. For a query p, its deletion neighborhood is generated and stored in $pVariants$. Then each variant $pMatch$ is looked up on disk via the *Index.get(pMatch)*

```
FSS(String p, k) {
   ResultList results = new List<String>
   StringList pVariants = precalculate(p, k)
   for pMatch in pVariants {
      for candidate in Index.get(pMatch) {
          if (FastED (candidate, pmatch) <=k) {
              results.add(candidate)
          }
      }
   }
   return results
}
//compare deletion lists, return ED
int FastED(p1[], p2[]) {
   int updates = 0;
   for (int i=0,j=0;i<p1.length && j<p2.length;) {
       if (p1[i] == p2[j]) {     //a substitution
           updates++;
           j++;
           i++;
       }
       else if (p1[i] < p2[j]) i++; //ins or del
       else if (p1[i] > p2[j]) j++; //ins or del
   }
   return p1.length + p2.length - updates;
}
```

Figure 4: Search using FastSS.

method, deletion positions are compared for each *candidate* by the method *FastED*, to find matches satisfying the edit distance k, and the *results* are output. FastED implements Theorem 4, using deletion lists $p1$ and $p2$. It returns the sum of both array lengths minus the number of substitutions. If *p1=p2*, the edit distance is $|p1| = |p2| = updates$.

3.4 FastSS with Candidates

This variant of FastSS, FastSS with candidates (FastSSwC), reduces space requirements of the index, but increases the search time complexity by not

storing a list of deletions for each word in the neighborhood. Instead, dynamic programming (DP) is used to verify candidate matches.

3.4.1 Indexing

For each indexed word and a given k only the words in the deletion neighborhood are stored, pointing to the original word, and no additional information. The space bound is the same as for FastSS, $O(nm^k)$. $U_d(fest, 2)$ contains *fest, est, fst, fet, fes, fe, fs, ft, es, et, st* pointing to *fest*. As the dictionary contains words like *jest, best*, also producing the misspelling *est*, *est* will point to a list containing *fest, best, jest* and possibly other words.

3.4.2 Retrieval

Figure 5 shows the pseudo code for FastSS with candidates. A query neighborhood $pVariants$ is first generated. This is looked up in the index, to return candidates that are within a given edit distance, and those with a higher ED. Results are verified using the edit distance formula implemented in procedure DP, see example in Table 1. The time complexity of the search is composed of the lookup cost of a term in a deletion dictionary with nm^k terms, and the cost of dynamic programming for each candidate, where $O(m^k)$ words are compared to c candidates in the deletion dictionary. With hashed index access this leads to $O(cm^k)$ time, and with log time index access to $O(cm^k log(nm^k))$, as DP needs $O(m^2)$ time for each candidate.

```
FSSwC(String p, int k) {
    StringList results = new List<String>
    StringList pVariants = precalculate(p, k)
    for pMatch in (pVariants) {
        for candidate in Index.get(pMatch) {
            if ( DP (candidate, pMatch) <= k) {
                results.add(candidate)
            }
        }
    }
    return results
}
```

Figure 5: FastSS with candidates and DP.

3.5 Fast Block Similarity Search

Theoretical time and space complexities are similar to FastSS with candidates. However, due to the fact that the index stores only non-redundant keys, the actual space usage is smaller that that of FastSSwC.

3.5.1 Indexing

The term b-gram or block is used here to represent distinct non-overlapping substrings of the mutated dictionary words, of maximum length b. The index in Table 2, right column, consists of non overlapping distinct b-grams (blocks) of length 1 to 3 and shows six such b-grams. The example word in the deletion neighborhood, *est1*, with $b = 3$ is split into *[est]* and *[1]*. Only unique b-grams are stored for each neighborhood, to remove redundancy. For example, the 2nd mutated sequence for *test1*, *tst1*, is split into *[tst]* and *[1]*. As *[1]* is already a key, derived from *est1*, it is not repeated in the index.

Table 2: $U_d(test1, 1)$ index keys for FastSSwC, and for FastBlocksSS with b=3.

FastSSwC	FastBlockSS
est1	[est][1]
tst1	[tst]
tet1	[tet]
tes1	[tes]
test	[t]
20 letters	14 letters, 6 b-grams

3.5.2 Retrieval

Figure 6 shows the pseudo code for FastBlockSS. First, generate a neighborhood *preList* of the query p, for up to k deletions. Second, for each word *bl* in the neighborhood generate a list of b-grams of size b or smaller. Third, process all the b-grams. For the first block, all matching blocks in the index, and all the words those blocks map to, are placed in a *tmpList*. Then, for each subsequent b-gram, the intersection of the *tmpList* and the word list pointed at from the next block are found. This finds all the dictionary words with an identical list of blocks. Those *candidates* are then verified using DP.

```
FastBlockSS(String p, int k, int b) {
    StringList results = new List<String>
    StringList candidates = new List<String>
    StringList preList = precalculate(p,k)
    for bl in preList {
        StringList icand = new List<String>
        StringList blocks = bl.split(b)
        for block in blocks {
            tmpList=Index.get(block)
            if(first block)
                icand.add(tmpList)
            else
                icand=intersect(icand,tmpList)
        }
        candidates.add(icand)
    }
    for cand in candidates {
        if DP(cand,p)<=k results.add(cand)
    }
    return results
}
```

Figure 6: Search using FastBlockSS with DP.

4 Evaluation

This section describes the testing data, the evaluation environment, and the experimental results.

4.1 Implementation

The benchmarks are presented in the following order: linear search based on DP, NR-grep, keyword tree, n-grams, and FastSS. All the software with the exception of NR-grep was implemented in Java and used `String` for string manipulation, and `HashMap`, `TreeMap`, and `HashSet` classes as data structures. To measure the index size, Java serialization is used.

4.1.1 Dynamic Programming

The DP algorithm was adapted from the Java implementation shown at `http://en.wikipedia.org/wiki/Levenshtein_distance#Java`. The benchmark iterated over all dictionary words, and calculated a full DP matrix for each query.

4.1.2 NR-grep

NR-grep was obtained from G. Navarro's web pages [24]. NR-grep is a fast and flexible text searching tool, which is written in C. It was run with the parameter $k = 2$, and set to use inserts, deletes and replacements, using option -dis.

4.1.3 Keyword Tree

Keyword tree implements the algorithm described by Gusfield [17], page 266 and following, called hybrid dynamic programming. The index structure represents each letter occurrence by a tree node, with hashed access to children via a Java `HashMap` class. DP similarity search algorithm has been implemented on top of this structure. To match a query of length p with k mismatches, one uses a matrix of size $(p + k + 1) \cdot (p + 1)$ to calculate the minimum cost path. This algorithm starts by filling the matrix with the pattern. The next step is to fill the matrix recursively with characters from the keyword tree. The words which share a prefix are not fully re-evaluated and their shared prefixes are effectively skipped while evaluating the minimum cost path at every character in the tree. If the minimum cost path has exceeded the specified number of edit operations, then this path will not be evaluated further.

4.1.4 Neighborhood Generation

Neighborhood generation was implemented for an alphabet of 26 letters, following [21]. It produces a large number of words not present in the dictionary. Neighbors were generated for a given k and looked up in an index. The index used Java class `HashSet` to store the dictionary.

4.1.5 N-grams

N-grams were implemented with $n = 2, 3, 4$ (2-gram, 3-gram, and 4-gram) and stored in three separate HashMaps. Depending on the minimum number

of common subsequences (mcs), as defined by the factors n (n-gram length), word length p, and number of mismatches k, one of the three HashMaps was chosen to perform the lookup. A desired property is to search first in 4-grams, then 3-grams and then 2-grams, as the larger the n, the fewer the n-grams. However, for short queries, no common subsequences may exist, and the complete dictionary has to be scanned, in $O(n)$ time. N-grams are indexed in a HashMap. For a faster lookup, the position of the n-gram in each word, together with the n-gram itself, have been indexed, resulting in a larger index and, possibly, a shorter lookup time, as fewer candidates will be found. Before a lookup is performed, the required number of minimum common subsequences mcs is calculated, $mcs = |p| - n + 1 - kn$. The lookup is then performed to search mcs n-grams that match the n-grams from p. $2k + 1$ n-grams are fetched, as the position of an n-gram may vary $2k + 1$ from the n-gram position in p. The results found are candidates and are verified with DP.

4.1.6 FastSS

FastSS was implemented with default hash and string functions. For FastSS, each tupel created via a deletion (mutated string, deletion position) points to the original dictionary word in a `HashMap`. For FastSSwC, each variant string points to the original dictionary word in the same way, and in FastBlockSS each b-gram points to the original dictionary word. An implementation of the same data structure using a `TreeMap` in Java was also undertaken, but proved not to be competitive. This is briefly discussed in the following section, with reference to Figure 7, which shows times for the `HashMap`.

4.2 In-Memory Measurements

This section provides details of in memory tests carried out. The evaluation was performed on a Pentium 4, 3.6 GHz, with 1 GB RAM, 16 KB level 1 cache, 2 MB level 2 cache, Java HotSpot(TM) Client VM 1.5.0_06, and Debian Linux with the 2.4.29 kernel. The running time of all evaluations was approximately one week. All tests were performed for 1000 queries, 50 times for each query, and from all test runs and queries the mean was calculated. The test involved a lookup of a randomly chosen word from the dictionary. The size of a b-gram for FastBlockSS was set to 4 for the English dictionary and 8 for the randomly generated dictionary, the edit distance was set to 2.

Figure 7 shows the average time for a lookup in an English dictionary

containing between 1,000 and 42,869 words (dictionary size on the x-axis). The dictionary is based on [4], but contains only characters [A-Za-z]. This graph shows the three FastSS variants. FastSS which uses the knowledge of delete positions is the fastest, and performs 1000 queries in less than 70 ms. FastSS with candidates requires the execution of DP and is up to three times slower. FastBlockSS is the slowest of the three and returns results in less than 900 ms, one order of magnitude slower than FastSS with delete positions. This is due to the large amount of intermediate candidates and candidate checking, involving DP calculation. The fluctuation in the FastBlockSS graph is due to the variation in the number of blocks that are shared between the query and the indexed dictionary, and the subsequent variation in the amount of DP that is executed.

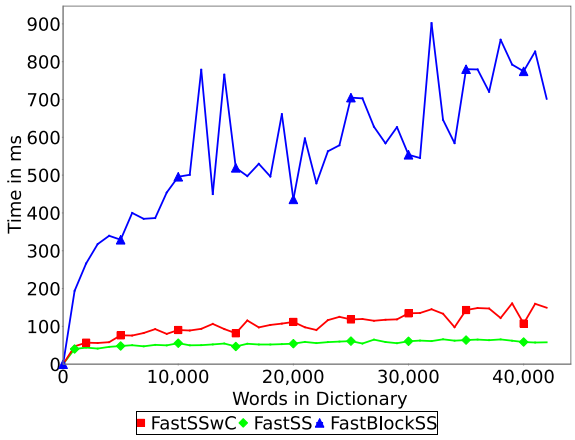

Figure 7: Lookup time for the three FastSS variants in an English dictionary

At this point a comparision of `TreeMap` and `HashMap` for FastSS and FastSSwC was undertaken. When TreeMap was used, lookup time doubled for FastSS to 150 ms, and FastSSwC time was 1.5 times longer, averaging 300 ms.

An overview of all algorithms looking up words in an English dictionary is presented in Figure 8 which uses a logarithmic scale for the y-axis. The family of FastSS algorithms perform about one to two orders of magnitude faster than the keyword map (keyword tree), n-grams, and NR-grep, which show similar performance, although the keyword map and n-grams use indexing and NR-grep does not. This is undoubtedly due to the fact that NR-grep is highly optimized and exploits the computer architecture better than our Java implementation of the keyword map. Both linear search

and neighborhood generation are not competitive, and perform lookups in 40,000 words in 50 to 100 seconds on average. The n-grams graph is shaky, as n-grams will not work for short words and the complete dictionary have to be scanned.

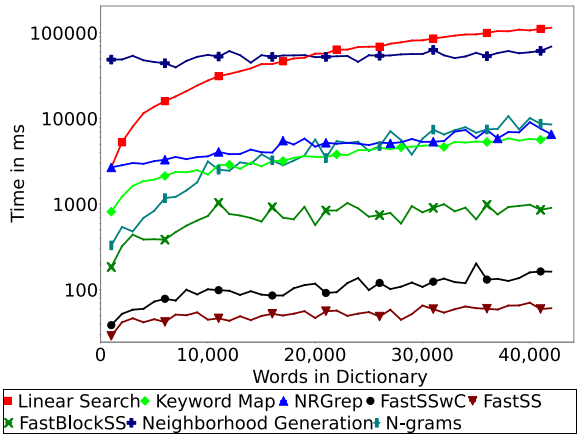

Figure 8: Lookup time in an English dictionary (log scale)

Figures 9 and 10 show average lookup time in ms in a randomly generated dictionary. The dictionary contains up to 10,000 words and the length of a word is chosen randomly to be between 3 and 30 characters. Figures 8 and 10 show that FastSS and its variants are considerably faster then all other algorithms. For the random dictionary, FastSS and FastSS with candidates perform the queries in under 250 ms, FastBlockSS in less than 750 ms. In the order of performance n-grams come next, followed by NR-grep, followed by the keyword map (keyword tree), and the linear search. Neighborhood generation is the slowest, as for long words the neighborhood size grows exponentially with word length.

It is interesting to observe that the keyword tree is slower with the randomly generated dictionary than with the English dictionary. This is due to the increased word length and artificial letter frequencies. Similar frequencies for all letters will lead to a more bushy tree, and with uncommon letter combinations, each node will have more children than is observed in English words.

Figures 11 and 12 show index size for the English and for the randomly generated dictionary. This is measured as the size of the serialized Java index object on disk, and does not apply to NR-grep and linear search, which are online approaches. Index sizes of the FastSS algorithms appear to

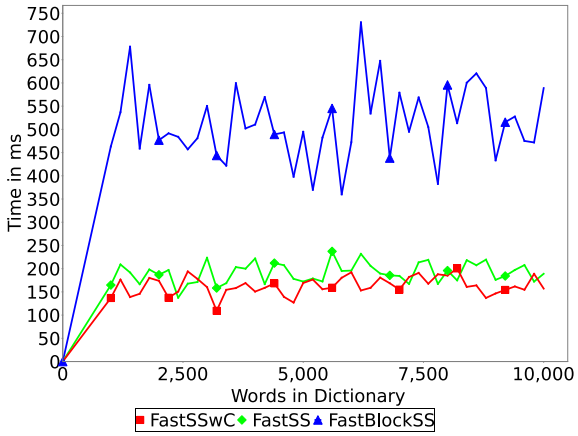

Figure 9: Lookup time for the 3 FastSS variants in a randomly generated dictionary

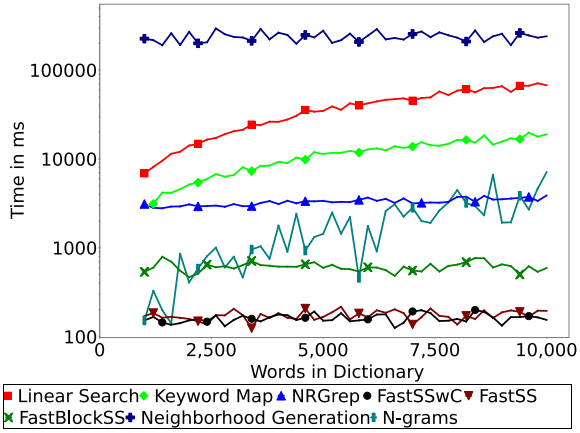

Figure 10: Lookup time in a randomly generated dictionary (log scale)

be linear in dictionary size, due to the fact that dictionaries have a fixed word length, and k is fixed at 2. The smallest here is the keyword map, followed by n-grams, FastBlockSS, FastSS with candidates, and finally FastSS. The largest of those indexes, FastSS index has the fastest performance, while the smallest (FastBlockSS) is the slowest of the three. This allows an application programmer to select the index best suited to the memory and performance requirements of the application using the index.

Figure 12 shows increased index sizes for the FastSS family of algorithms for the random dictionary, in comparison to Figure 11 showing an English

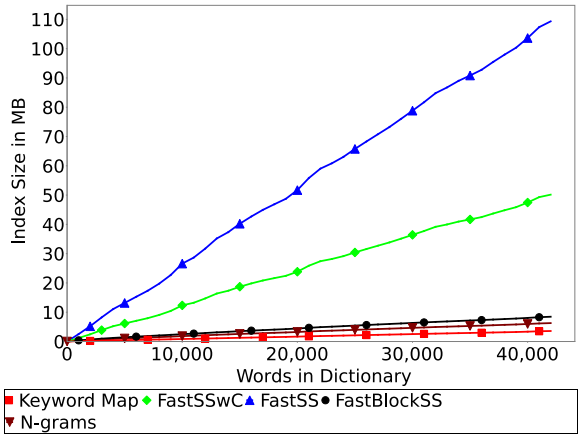

Figure 11: Index size for an English dictionary

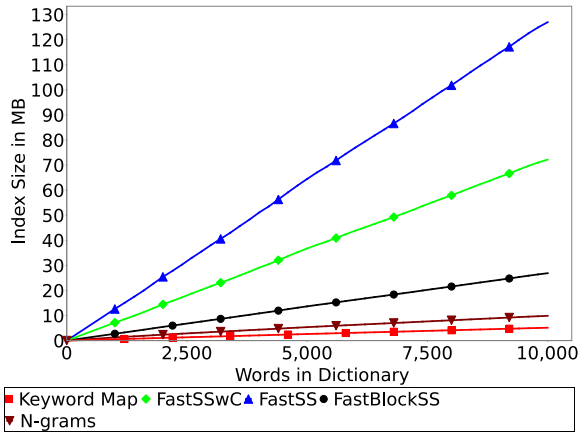

Figure 12: Index size for a randomly generated dictionary

dictionary. This is despite the fact that the number of words in the English dictionary is four times the number of words in the random dictionary, and the total volume of indexed text is twice as large as in the randomly generated dictionary. Specifically, the dictionary size is 165 KB for the randomly generated dictionary and 388 KB for the English dictionary. The increased index size for the random dictionary is due to the increased word length, and a larger number of possible variants that are stored. The keyword map is the smallest index for random text.

Figure 13 illustrates the relationship between word length and index size in the random dictionary for FastSS and FastBlockSS for 100 words of in-

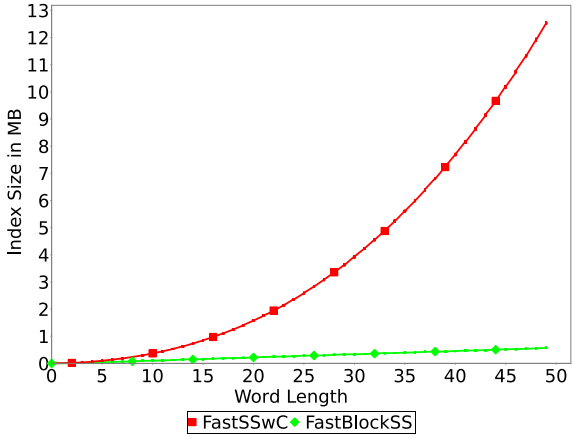

Figure 13: Space usage with increasing word lengths

creasing length. FastBlockSS uses the concept of blocks to avoid redundancy and uses less space that FastSSwC.

Figures 14 and 15 show index creation time in ms for the English and the random dictionary for offline algorithms that pre-calculate the dictionary. An index to an English dictionary can be built in less than 8 seconds. 10,000 random words can be indexed in under 8 seconds. FastBlockSS is the fastest with respect to index creation time, because redundant blocks are not stored. The keyword map and n-grams algorithm build indexes much faster than FastSS algorithms.

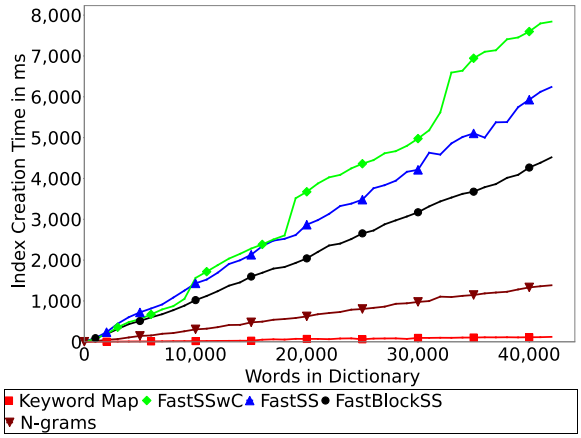

Figure 14: Index creation time for an English dictionary

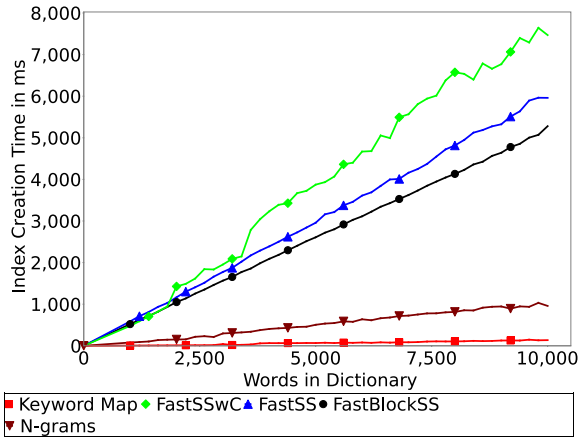

Figure 15: Index creation time for a randomly generated dictionary

Overall, FastSS and its variants perform faster than dynamic programming, NR-grep, keyword map, neighborhood generation and n-grams. The benefit is a faster search time, but it is combined with the drawback of having to manage a large index and an initial index creation time.

4.3 Large Database Measurements

To measure and compare FastSS with a large dataset, 53,177 English Wikipedia articles and meta pages[2] with at least 1000 characters each have been indexed in a MySQL database (MySQL 4.0.24)[22] using FastSS, with k=1, 2, 3, and n-grams, with n=2,3,4. The user selects in a web GUI, which runs with Apache 2.0.54-5sarge1 [8], the method to use, which is either FastSS or n-grams, and the edit distance k. A PHP script (PHP Version 5.1.6 without acceleration) [6], accessed via the GUI, measures the time for a lookup for both methods with k=1, k=2, and k=3. In the measurements reported here, the first cold test run is ignored, the following 10 warm test runs search for the first 100 words in the dictionary.

N-grams do not support searching for short words. Therefore, only longer words have been tested for both methods. A word of length 7 with 2-gram and k=3 cannot be exhaustively searched using n-gram filtering as there may not be any common n-grams. The fallback strategy to search the complete dictionary would take too long. From the first 100 words from the dictionary, 16 words were skipped for k=1, 61 words were skipped for k=2, and 84 were skipped for k=3. Effectively, for k=1 84 test words were used, for k=2, 39 words, and for k=3, 16 words were timed.

4.3.1 Implementation

The implementation of FastSS and n-grams using a database to store the index and data are described in the following. The indexes for 2-grams, 3-grams, and 4-grams are stored in a database, using the following relations, where FK stands for foreign key referencing the word id in the `words` table, see below, and INT for integer.

```
2gram (gram CHAR(2), wid INT FK)
3gram (gram CHAR(3), wid INT FK)
4gram (gram CHAR(4), wid INT FK)
```

The index for FastSS with k=1, k=2, k=3 is stored in the database in the following relations.

```
precalc1 (precalc CHAR(20), wid INT FK)
precalc2 (precalc CHAR(20), wid INT FK)
precalc3 (precalc CHAR(20), wid INT FK)
```

N-gram and precalc relations reference the primary key (PK) called `id` in relation words.

```
words (id INT PK, word CHAR(20))
```

The remaining information about the text units is stored in relation `wordlist`, which encodes the document id (tid), word id (wid), word position, and the length of the article or chapter.

```
wordlist (tid INT, wid INT, position INT)
```

The position of the n-gram and the position of the deletions for FastSS are not considered, and both approaches use DP.

4.3.2 Performance

Index sizes for 53,177 English Wikipedia articles are shown in Table 3 for FastSS and in Table 4 for n-grams. The dictionary created from Wikipedia articles and metapages contains 465,498 words, utilizing 18.5 MB.

Table 5 shows the average time for one lookup of FastSS and n-grams in a large dataset. A similarity search for both methods with k=1 is faster than k=2 which in turn is faster than a similartity search with k=3. FastSS clearly outperforms n-grams.

Table 3: Index size for FastSS in MySQL

FastSS, k=1	FastSS, k=2	FastSS, k=3
117.3 MB	495.3 MB	1.5 GB

Table 4: Index size for n-grams in MySQL

2-gram	3-gram	4-gram
40.8 MB	39.7 MB	58.8 MB

Table 5: Lookup time for similar words in a subset of the English Wikipedia and meta pages

	k=1	k=2	k=3
FastSS	0.01 s	0.07 s	0.57 s
N-grams	0.29 s	3.13 s	35.06 s

5 Demonstrations

FastSS indexes deletions. This index can be stored in an SQL database, or can be kept in memory. Both versions can be downloaded from the FastSS project site[27]. The similarity search demo applications run on Debian with the 2.4.29 kernel, PHP Version 5.1.6 without acceleration, Apache 2.0.54-5sarge1, and SQLite 3.2.8. The machine has an Intel(R) Pentium(R) 4 CPU 3.60 GHz with 16 KB level 1 cache, 2 MB level 2 cache and 1 GB RAM.

5.1 Moby Dick Demonstration

Fast Similarity Search (FastSS) is used to find similar words in 135 chapters from the book *Moby Dick; or The Whale* by Herman Melville [20]. If a similar word is found, the result page is shown with the chapters where the word has been found and how fast the lookup was. This application does not consider ranking. It performs a fast similarity search and the results are displayed in the order they are found. The index and the chapters are stored in an SQLite database [7] of size 32 MB. The size of the text itself is 1 MB.

The application, see Figure 16, is available online [27]. The GUI has a query field and two buttons. After the similarity or exact search button has been pressed, the lookup time along with not more than 40 matches will be shown. This restriction has been made because a similarity search with

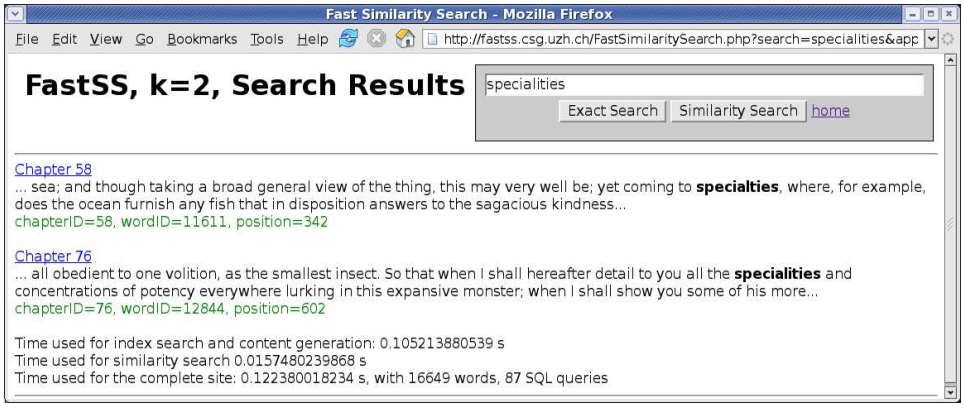

Figure 16: Screenshot: FastSS demo web page with similarity search in Moby Dick.

a large edit distance and a short query produces a high load, due to many matches being found. A search for the word *dew* in an English dictionary with approximately 42,000 entries results in 153 similar words for the edit distance equal or less than 2.

Two query options are provided in the application. One is an exact search, the other is a similarity search that uses FastSS with up to two mismatches. When we enter the query *test*, the exact search returns 4 results containing the word *test*, while the similarity search returns 31 results containing the words *meet, get, that, these, pent, must, just, left, Let, most, set, feet, yet, Tell*. The similarity search stops at chapter 1, because, otherwise, it would exceed the limit of 40 matches.

5.1.1 Implementation

The implementation of FastSS using a database to store the index and data are described using the following relations.

```
words (id INT PK, word TEXT)}
precalc (id INT PK, precalc CHAR(32),
   wid INT, deletion INT)
```

The word *test* is stored in the relation *words*. *precalc.deletion* contains the number of deletions, and *precalc.wid* references *words.id*. The index for the chapter information is stored in relation `wordlist`.

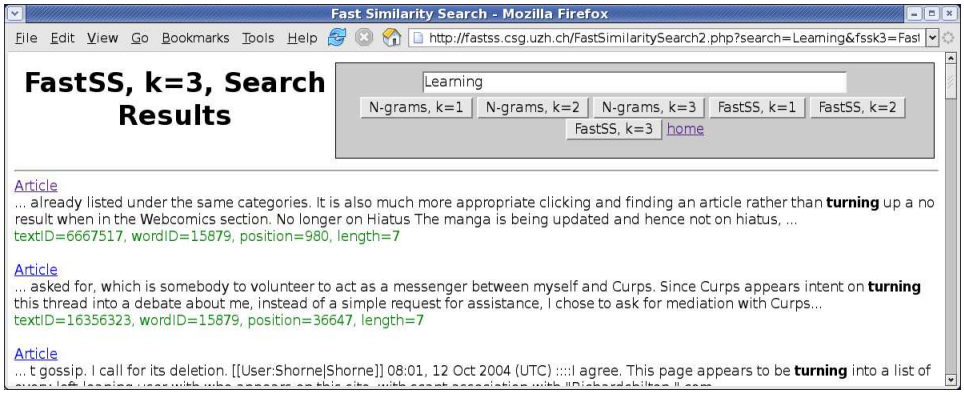

Figure 17: Screenshot: FastSS demo web page with similarity Search in English Wikipedia and meta pages.

```
wordlist (cid INT, wid INT, position INT,
   pre TEXT, original TEXT, post TEXT)
```

The ids of the similar words (*wid*), which have been found using the index, are looked up to find the corresponding chapter id (*cid*). The columns *position, pre, original* and *post* are used to display the result.

5.2 Wikipedia Demonstration

FastSS is used in this demonstration, see Figure 17, to find similar words in 53,177 English Wikipedia articles and meta pages stored in a MySQL database. If a similar word is found, the result page is shown with an overview of articles found. This application does not consider ranking.

The GUI [27] shows six buttons. Three buttons search with FastSS and k=1, k=2, and k=3. The other three buttons search with n-grams and k=1, k=2, and k=3. After pressing one of this six buttons, the lookup time along with not more than 40 matches will be shown. The implementation has been presented in Section 4.3.1.

6 Discussion and Future Work

The size of the neighborhood defined by insertions, deletions, and replacements, U_{idr}, is defined by the length of the dictionary word m, the alphabet size s, and ED, k. The space complexity is $O(m^k s^k)$ [26] since each of the k positions selected out of m positions can be filled by s letters. This is

considerably larger than the size of the deletion neighborhood U_d which is $O(m^k)$.

Although the search using neighborhood generation has sub linear time complexity in the size of the target data, and it is appropriate and efficient for DNA and protein sequences [21], it is the slowest algorithm in the evaluation performed. The large number of neighbors when using a large alphabet and long words makes the algorithm slow and reflects the size of the neighborhood, as predicted by theory. For an alphabet of 26 characters, neighborhood generation generated 23,883 unique neighbors for the word *test* with ED=2. These neighbors have to be created first and secondly looked up in the index, which contributes to poor performance.

N-grams are slower in the measurements shown in Section 4 than FastSS, as the list of candidates returned by n-grams is larger. The higher the number of mismatches k to be searched for with n-grams, the fewer common blocks between a query and the n-grams are found, and the more candidates are present. The result will be a slower search as these candidates have to be checked with dynamic programming. N-grams could be evaluated faster if a faster dynamic programming algorithm, such as NR-grep [24] is used. FastSS could be speeded up as well by additionally indexing delete positions. The indexing of all delete positions and combinations of delete positions would result in a faster search with FastSS, as no candidates would have to be checked with dynamic programming.

The experimental evaluation shown here also indicates that using U_{idr} to drive the generation of variants does not make sense in natural language as many of those variants are not likely to exist. In U_{idr} generation there is more processing to generate the variants but a smaller index containing only the dictionary words. In a deletion-based neighborhood U_d the index is larger, but the neighborhood is smaller and can be generated significantly faster.

A reduction of index size using *stemming* could reduce the search time. The drawback is that stemming is language-dependent. For every language a different stemming algorithm has to be used. Another reduction method is to skip frequent words such as “the” or “of”. Further reduction of the dictionary size can be achieved by compression. Lossless compression can be used as well as lossy compression, but with lossy compression results have to be double checked with dynamic programming. FastSSwC and FastBlockSS check the candidates with DP, therefore lossy compression would integrate best with FastSSwC and FastBlockSS.

Edit operations modeled in FastSS and its variants are insertions, deletions, and updates. A possible extension could be adding more operations,

for instance a transposition of characters that are next to each other [11]. This kind of operation can be added without modifying the index structure or without adapting the algorithm significantly. One would expect similar performance and index size.

One drawback of FastSS is that the index can be large. For an edit distance 2, the English dictionary has been stored in an index that is up to 250 times larger than the dictionary. The index size for FastSS and its variants can vary from 10 to 100 MB, while the dictionary size is 388 KB. FastSS has been implemented in both in-memory and disk models. For large dictionaries, like the one gathered from Wikipedia, the database index is a more appropriate solution. A demo application [27] stores all information in a relational database, and demonstrates a server scenario.

Future work will focus on the index and algorithm tuning, to speed up the search.

7 Summary and Conclusions

This paper presents a fast similarity search algorithm for large dictionaries, which is faster than other existing algorithms. The equivalence of a deletion-based neighborhood and edit distance is demonstrated, and substantial experimental evidence backs the claim that FastSS outperforms all algorithms compared with respect to the search time. These findings show that pre-calculating and indexing the dictionary with only a subset of possible mutation operations reduces the time complexity significantly. Additionally, it is faster in practice. Although only a subset of mutation operations is used, the algorithm returns results that are equal to the edit distance by combining deletions.

The algorithms outlined here could be used by a search engine in an Intranet, a spell checker, or by a desktop search engine. However, for a real-world application that improves the quality of a search, the edit distance has to be adaptable, and a percentage identity measure can be a viable alternative. The quality of the search will not be increased when searching a short word with a high edit distance. A search for the word such as *dew* with the edit distance less or equal 2, equivalent to a 33% identity, results in 153 words, while a search for the word *database* with the same edit distance, results in 2 similar words. Ranking is also necessary to return best matches first.

Another aspect is whether the tradeoff between the index size and the increased lookup time is affordable for a real-world application. There is no

general answer to this question, but due to the fact that FastSS is suitable for databases and disk space is cheap, in scenarios with many lookups and few updates this solution is very appropriate.

References

[1] Apache Lucene. http://lucene.apache.org/, 2007.

[2] Enwiki dump progress on 20070206: Articles, templates, image descriptions, and primary meta-pages. http://download.wikimedia.org/enwiki/20070206/enwiki-20070206-pages-articles.xml.bz2, 20.3.2007.

[3] IBM Software - DB2 Net Search Extender - Product Overview. http://www.ibm.com/software/data/db2/extenders/netsearch/, last visited: 10. 11. 2006.

[4] Jazzy The Open Source Spell Checker. http://jazzy.sourceforge.net, 2006.

[5] Joomla! http://www.joomla.org, 2006.

[6] PHP: Hypertext Preprocessor. http://www.php.net/, 2007.

[7] SQLite homepage. http://www.sqlite.org, 2006.

[8] The Apache HTTP Server Project. http://httpd.apache.org/, 2007.

[9] TYPO3. http://typo3.com, 2006.

[10] Zope.org. http://www.zope.org, 2006.

[11] A. Amir, E. Eisenberg, and E. Porat. Swap and mismatch edit distance. *Algorithmica*, 45(1):109–120, 2006.

[12] R. Baeza-Yates and G. Navarro. A Hybrid Indexing Method for Approximate String Matching. *JDA*, 1:205–239, 2001.

[13] N. Blachman. Google Guide Making Searching Even Easier. http://googleguide.com/print_understand_results.html, last visited: 10. 11. 2006.

[14] A. Califano and I. Rigoutsos. FLASH: A Fast Look-up Algorithm for String Homology. In *ISMB*, 1993.

[15] A. L. Cobbs. Fast approximate matching using suffix trees. In Z. Galil and E. Ukkonen, editors, *Combinatorial Pattern Matching, 6th Annual Symposium*, volume 937 of *Lecture Notes in Computer Science*, pages 41–54, Espoo, Finland, 5-7 July 1995. Springer.

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

[17] D. Gusfield. *Algorithms on strings, trees and sequences: computer science and computational biology*. Cambridge University Press, 1997.

[18] H. Hyyrö and G. Navarro. A Practical Index for genome Searching. In *SPIRE'03*, LNCS:2857, pages 341–349.

[19] V. I. Levenstein. Binary codes capable of correcting insertions and reversals. *Sov. Phys. Dokl.*, 10:707–10, 1966.

[20] H. Melville. *Moby-Dick; or, The Whale*. Harper and Brothers, 1851.

[21] E. Myers. A sublinear algorithm for approximate key word searching. *Algorithmica*, 12(4/5):345–374, 1994.

[22] http://www.mysql.com/.

[23] D. S. N. Koudas, A. Marathe. Flexible String Matching Against Large Databases in Practice. In *VLDB*, pages 1078–1086, 2004.

[24] G. Navarro. NR-grep: A Fast and Flexible Pattern Matching Tool, Technical Report TR/DCC-2000-3. Technical report, University of Chile, Departmento de Ciencias de la Computacion, Santiago, 2000. http://www.dcc.uchile.cl/∼gnavarro.

[25] G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.

[26] G. Navarro, R. Baeza-Yates, E. Sutinen, and J. Tarhio. Indexing Methods for Approximate String Matching. *IEEE Data Engineering Bulletin*, 24(4):19–27, 2001.

[27] These authors. Fast Similarity Search. http://fastss.csg.uzh.ch, 2006.

[28] E. Ukkonen. Approximate string matching over suffix trees. *CPM93*, 684:228–242, 1993.

[29] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173, 1974.