A practical index for approximate dictionary matching with few mismatches

Aleksander Cisłak† and Szymon Grabowski‡

† Technische Universität München, Department of Informatics,
Boltzmannstr. 3, D-85748 Garching, Germany, a.cisla@tum.de
‡ Lodz University of Technology, Institute of Applied Computer Science,
Al. Politechniki 11, 90–924 Łódź, Poland, sgrabow@kis.p.lodz.pl

Abstract. Approximate dictionary matching is a classic string matching problem, with applications in, e.g., online catalogs, geolocation (mapping possibly misspelled location description to geocoordinates), web searchers, etc. We present a surprisingly simple solution, based on the Dirichlet principle, for matching a keyword with few mismatches and experimentally show that it offers competitive space-time tradeoffs. Our implementation in the C++ language is focused mostly on data compaction, which is beneficial for the search speed (e.g., by being cache friendly). We compare our solution with other algorithms and we show that it performs better for the Hamming distance. Query times in the order of 1–2 microseconds were reported for one mismatch for the dictionary size of a few megabytes on a medium-end PC. We also demonstrate that a basic compression technique consisting in character substitution can significantly reduce the index size for the DNA data (below the dictionary size), while still keeping the query time relatively low.

1 Introduction

Dictionary string matching, defined as the task of checking if a query string occurs in a collection of strings given beforehand, is a classic research topic. In recent years, an interest in approximate dictionary matching can be observed, where the query and one of the strings from the dictionary may only be similar in a specified sense (e.g., differ by at most \( k \) mismatches) rather than equal. Approximate dictionary matching is considered a hard problem, since most useful string similarity measures are non-transitive. On the other hand, matching with tolerance for errors is also a very convenient functionality, e.g., for online catalogs, stores, web searchers, etc.

As indexes supporting approximate matching tend to grow exponentially in \( k \), the maximum number of allowed errors, it is also a worthwhile goal to design efficient indexes supporting only a small \( k \). In this paper we focus on the problem of dictionary matching with one mismatch. Formally, for a collection \( \mathcal{D} = \{d_1, \ldots, d_m\} \) of \( m \) strings \( d_i \) of total length \( n \), over a given alphabet \( \Sigma \), \( I(\mathcal{D}) \) is an approximate dictionary index supporting matching with one mismatch, if for any query pattern \( P \) it returns all strings \( d_j \) from \( \mathcal{D} \) such that \( \text{Hamming}(P, d_j) \leq 1 \).
2 Related work

The solutions for the approximate dictionary matching problem can be basically divided into two classes: worst-case space and query time oriented, and heuristical ones. Notable results from the first class comprise the $k$-errata tree by Cole et al. [5] using $O(n \log^k n)$ space and $O(|P| \log \log n + \text{occ})$ query time, where the maximum allowed edit distance is $k$, and the dictionary by Belazzougui [1] designed for one error, with the $O(n(\log n \log \log n)^2 \log |\Sigma|)$ bits of space and $O(|P|)$ query time.

Many algorithms from the latter class are more practical, often require linear space (e.g., are based on the suffix tree), but they do not have strong worst case performance guarantees. See the extensive survey from Boytsov [3] for more results and details.

Many popular techniques are filters. Roughly speaking, they divide the vocabulary into many (disjoint or overlapping) clusters and a query word is then assigned to one or several clusters containing candidate strings to find the distance to. Bocek et al. [2] presented a simple technique for string neighborhood generation based on deletions, which was a generalization of the Mor and Frankel [9] idea for one error. To check if two strings, $S_1$ and $S_2$, are matching with up to $k$ errors, we delete all possible ordered subsets of $k'$ symbols, for all $0 \leq k' \leq k$, from $S_1$ and from $S_2$, and iff the intersection of the resulting lists of strings is non-empty, then $S_1$ and $S_2$ may be in edit distance at most $k$ (verification is then required). Let us give an example: if $S_1 = \text{abbac}$ and $k = 2$, then its neighborhood is: $\text{abbac, bbac, abac, abbc, abba, abb, abc, aba, abc, aac, bba, bbc, bac and bac}$ (of course, some of the resulting strings repeat and may be removed). If $S_2 = \text{baxy}$, then its respective neighborhood for $k = 2$ will contain, e.g., the string $\text{bac}$, but the following verification will show that $S_1$ and $S_2$ are in edit distance greater than 2. If, however, $ed(S_1, S_2) \leq 2$, it is impossible not to have in the neighborhood of $S_2$ at least one string from the neighborhood of $S_1$, hence we will never miss a match.

A classical solution for dictionary matching with one error is called the permuterm index [8]. The idea is to take every string $s \in D$, append a terminator symbol $\$$ to it, and then consider all cyclic rotations of $s\$$$. If the newly created strings are inserted to a trie, we can perform prefix searches on them, which now translate to arbitrary (exact) substring searches. Note that the subtrie rooted in the node labeled with the string $s_{j+1,...,\text{len}(s)}\$$s_1,...,j-1$, where $\text{len}(s)$ is the length of $s$, has as its leaves all the strings from $D$ which differ to $s\$$ only at one position $(j)$, appropriately rotated, plus $s\$$ itself (also rotated by $j$ symbols). The trie can be replaced with any data structure supporting prefix searches.

The main problem with the standard permuterm index is its space usage, as the number of strings inserted to the data structure is $m$ multiplied by the average string length (which may be often around 10 or more in practice). Ferragina and Venturini [7] gave a modification of the LF-mapping known from FM-indexes [6] to support the functionality of the permuterm index. As this compact solution is based on the well-known “compressibility” of the BWT transform (in
this case, the BWT is applied to a concatenation of all the strings from $D$), its name, \textit{compressed permuterm index}, is justified.

The algorithm perhaps most similar to the one presented in this work was given by Shi and Widmayer [10]. They introduce the notion of \textit{home strings} of a given $q$-gram, which is the set of strings in $D$ that contain the $q$-gram in the exact form. The value of $q$ is set to $|P|/(k+1)$. In the search phase, they break $P$ into $k + 1$ disjoint $q$-grams and use a careful candidate inspection order to speed up finding the matches with up to $k$ edit distance errors.

\section{Our algorithm}

The algorithm that we are going to present in this section is uncomplicated and based on the Dirichlet principle, ubiquitous in approximate string matching techniques. We partition each word $d_i \in D$ into $k + 1$ disjoint pieces of length $|d_i|/(k + 1)$ and each such piece is a key in a hash table $T$. The values in $T$ are lists of words having the corresponding key of one of their pieces. In this way, every words occurs on exactly $k + 1$ lists, which seemingly bloats the space use. Fortunately, the case of $k = 1$ is rather mild. Instead of storing full words on the respective lists, we only store their “missing” prefix or suffix. As the order of words on a list does not matter, we first populate the list with the words without their prefix and then with the words without the suffix; additionally we store the position on the list (as a 16-bit index) where the latter part begins. In this way, we traverse only a half of a list, on average, during the search. This kind of trick does not seem to pay off for $k$ larger than 1. In this case, we do not care for the word order on a list, but we store $\lceil \log_2(k + 1) \rceil$ bits with each word for telling which part of the word is the list key.

As regards the implementation, our main focus was on data compactness. In the hash table, we store the buckets which contain word parts as keys (e.g., \texttt{tab}) and pointers to the lists, which store the “missing” parts of the word (e.g., \texttt{le}, \texttt{let}, \texttt{oo}). These pointers are always located right next to the keys, which means that unless we are very unlucky, a specific pointer should already be present in the CPU cache during the traversal. Strings are represented by multiple characters with a prepended 8-bit counter which specifies the length. The memory layouts of these substructures are fully contiguous.

Moreover, we apply a basic compression technique. We find the most frequent $q$-grams (for a fixed $q$ given at index build time; e.g., $q = 2$ and $q = 4$ are reasonable for English text and DNA, respectively) in the word collection and replace their occurrences on the lists with unused symbols (e.g., byte values 128, \ldots, 255 for English).

\section{Experimental results}

Experimental results were obtained on the machine equipped with the Intel i5-3230M processor running at 2.6 GHz and 8 GB RAM.
We investigated various hash functions, and the differences in query times turned out to be relatively small. We can see in Figure 1 that the fastest function was Hsieh’s SuperFastHash\(^1\) (*nomen est omen*).

![Figure 1](image-url)  
**Fig. 1.** Hash functions vs query time \((k = 1)\), for the list of 4261 common typos and the English language dictionary of size 2736 KB.

Reducing the value of the load factor did not provide a speedup in terms of the query time, as demonstrated in Figure 2. This can be explained by the fact that even though the relative reduction in the number of collisions was substantial, the absolute difference did not exceed a few collisions per list. Moreover, when the hash factor were lower, memory locations of the lists could be possibly closer to each other, which might have had a positive effect on cache utilization.

In Figure 3 we can see a linear decrease in the index size and an expected exponential increase in query time, as the maximum error value \(k\) for the Hamming distance increased.

Just as expected, \(q\)-gram coding provided a reduction in index size, at the cost of increased query time. \(Q\)-grams were generated separately for each dictionary as a list of 100 most common \(q\)-grams for that particular dictionary. For the English language dictionaries\(^2\), more favorable results were reported for 2-grams rather than 3-grams. We can see the speed-to-space relation in Figure 4. In the case of DNA dictionaries presented in Figure 5, best results were observed for 2-grams and 4-grams, where the former provided a slightly better compression rate.

---

\(^1\) https://code.google.com/p/smhasher/source/browse/branches/chandlerc_dev/SuperFastHash.cpp

\(^2\) English language dictionaries are available from the *iamerican* and *iamerican-insane* Ubuntu packages, and under the following link: http://www.math.sjsu.edu/~foster/dictionary.txt.
Fig. 2. Query time and index space \((k = 1)\) vs the load factor, for the list of 4261 common typos and the English language dictionary of size 2736 KB.

Fig. 3. Query time and index space vs the error value \((k)\), for the list of 4261 common typos and the English language dictionary of size 794 KB.
at the cost of the query time. For both q-gram methods in the case of DNA data, the resulting index turned out to be up to 30% smaller than the input dictionary. The queries were generated randomly by introducing noise into samples taken from dictionary entries, and their length was equal to the length of k-mers in the dictionary. Up to 3 errors were inserted, each with a 50% probability.

![Graph showing query time and index space vs dictionary size](image)

**Fig. 4.** Query time and index space vs dictionary size ($k = 1$), with and without q-gram coding. The list of 4261 common typos and the English language dictionaries were used.

Tested on the English language dictionaries, our algorithm performed favorably when compared to methods proposed by other authors. Others consider the Levenshtein distance as the edit distance, whereas we use the Hamming distance, which puts us at the advantageous position. Still, the provided speedup is significant, and we believe that the more restrictive Hamming distance is also an important measure of practical use. As regards the results reported for the Mor–Fraenkel algorithm and Boytsov’s Reduced alphabet neighborhood generation, we were not possible to accurately calculate the size of the index (both implementations by Boytsov, available on the Internet\(^3\)). For this reason, we used rough ratios based on index sizes reported by Boytsov [4] for similar dictionary sizes. It is worth noticing that the times for the brute-force algorithm are not listed, since they were roughly 3 orders of magnitude longer than the ones presented. Consult Figure 6 for details.

\(^3\) [http://searchivarius.org/personal/software](http://searchivarius.org/personal/software)
Fig. 5. Query time and index space vs dictionary size ($k=1$), with and without $q$-gram coding. Queries with random noise (up to 3 errors) and the dictionaries of DNA 20-mers (k-mers of length 20) obtained from the genome of Drosophila melanogaster (fruit fly) were used.

We also evaluated different word splitting schemes. For instance for $k=1$, one could split the word into two parts of different sizes, e.g., $6 \rightarrow (2, 4)$ instead of $6 \rightarrow (3, 3)$. Unfortunately, unequal splitting methods caused slower queries when compared with the regular one.

## 5 Conclusions

We have presented an index for dictionary matching with mismatches, which performed best for the Hamming distance of one. Its functionality could be extended by storing additional information in the lists that contain the “missing” parts of the words. This could be for instance a mapping of words to positions in the document, which would create an inverted index supporting approximate matching.

Moreover, the space usage could be further reduced by using a different character encoding. For the DNA (assuming 4 symbols only) it would be sufficient to use 2 bits per character, and for the “pure” English alphabet 5 bits (there are 26 letters, which in a simplified text can be augmented only with a space character, a few punctuation marks and a capital letter flag). Such an approach would be also beneficial for space compaction, and it could have a further positive impact on cache usage.
Fig. 6. Query time vs index space for different methods. We used the Hamming distance, and the other authors used the Levenshtein distance for $k = 1$. The list of 4261 common typos and the English language dictionaries of size 794 KB, 2736 KB, and 5800 KB were used.

References

1. D. Belazzougui. Faster and space-optimal edit distance "1" dictionary. In *Combinatorial Pattern Matching*, pages 154–167. Springer, 2009.
2. T. Bocek, E. Hunt, B. Stiller, and F. Hecht. Fast similarity search in large dictionaries. Technical Report ifi-2007.02, Department of Informatics, University of Zurich, 2007.
3. L. Boytsov. Indexing methods for approximate dictionary searching: Comparative analysis. *Journal of Experimental Algorithmics (JEA)*, 16:1–1, 2011.
4. L. Boytsov. Super-linear indices for approximate dictionary searching. In *Similarity Search and Applications - 5th International Conference, SISAP 2012, Toronto, ON, Canada, August 9-10, 2012. Proceedings*, pages 162–176, 2012.
5. R. Cole, L.-A. Gottlieb, and M. Lewenstein. Dictionary matching and indexing with errors and don’t cares. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pages 91–100. ACM, 2004.
6. P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proceedings of the 41st IEEE Annual Symposium on Foundations of Computer Science*, pages 390–398, 2000.
7. P. Ferragina and R. Venturini. The compressed permute.m index. *ACM Transactions on Algorithms (TALG)*, 7(1):10, 2010.
8. C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.
9. M. Mor and A. S. Fraenkel. A hash code method for detecting and correcting spelling errors. *Communications of the ACM*, 25(12):935–938, 1982.

10. F. Shi and P. Widmayer. Approximate multiple string searching by clustering. In *Genome Informatics Workshop*, Tokyo, Japan, 1996. Universal Academy Press.