TEXT ENCODING AND RECALL SPEEDUP FOR CORRELATION MATRIX MEMORIES

Tomáš Beran[1,2], Tomáš Macek[2] and Miroslav Skrbek[1]

[1]Department of Computer Science and Engineering, Czech Technical University
Karlovo náměstí 13, 121 35 Prague 2, Czech Republic
[2]T. J. Watson Research Group, IBM Czech Republic s.r.o.
The Park 2294/2, 148 00 Prague 4 Chodov, Czech Republic

xberant@fel.cvut.cz(Tomáš Beran)

Abstract

In this article we describe a part of our search engine based on Correlation matrix Memories. We focus on a part (we refer it as a letter-word matcher) of our search engine that takes a single word from an input query and looks for its word representative (word label). Words are searched within a static lexicon that is trained from a collection of documents. Although our letter-word matcher provides exact matching, approximate matching and stemming, we pay here attention on the exact matching only. Our search engine is based on the Correlation Matrix Memories (CMMs). CMMs are type of binary neural networks. They are capable of both exact and approximate matching. An advantage of CMMs is its very fast learning process. We proposed two encoding methods of input patterns designed to reduce memory consumption of CMMs. Both methods give some level of error rate in comparison with a standard approach. The first method allows to reduce memory more than 7 times. There is a tradeoff between memory requirement and error rate value. We also tested an n-gram approach for memory consumption and the error rate. We suggest three methods of speeding up a software simulation of the CMM recalling process. Combining all three we achieved a significant speedup of a standard method.

Keywords: Correlation matrix memory, Binary neural networks, Text searching, N-gram approach, Speedup of recall.

Presenting Author's Biography

Tomáš Beran. He finished his MSc degree in 1999 at the Czech Technical University in Prague on the topics of languages and translations. His interest in artificial neural networks leads to the implementation of an optical music recognition system based on binary neural networks, presented in his MSc thesis. He continued to use such neural networks in his doctoral study on text processing. He joined the IBM T. J. Watson Research group in Prague in 2002 where he has been part of the team developing the speech recognition engine for embedded devices.

ISBN 978-3-901608-32-2

1 Introduction

This article deals with an application of binary correlation matrix memory (CMM) for exact text matching task over a static lexicon of words. It is a part of our search engine that seeks word representatives based on input query. CMMs are type of binary neural networks that are capable of perfect matching or matching admitting errors. Advantages of CMMs are also fast learning process and easy implementation.

Here, we focus on two problems. First, we evaluate two encoding methods to reduce memory consumption. Both methods introduce some level of error rate. Secondly, we suggest three methods of speeding up a software simulation of the recall process of CMMs.

We briefly describe binary correlation matrix memories in section 2. Then, we present evaluation of two encodings methods in sections 3 and 4. Section 5 describes acceleration methods of the recall process of CMM.

2 Text Matching by CMMs

Our search engine produces a list of relevant documents based on an input query. Its structure is shown in Fig. 1.

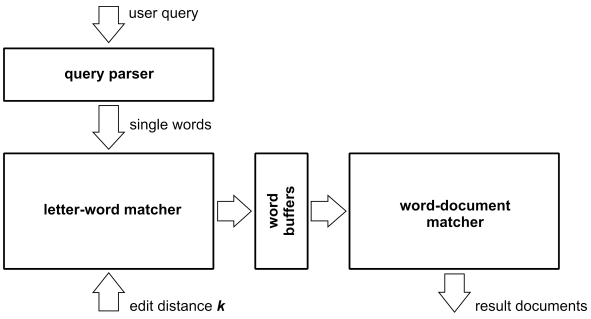

Fig. 1 The structure of our search engine.

The input query consists of a sequence of words that a user considers to be important for searching desired documents. The engine contains two main parts: *a letter-word matcher* and *a word-document matcher*.

The first part provides a search on the letters of an input query words for word representatives within a static lexicon. It allows to perform an exact match as well as an approximate match. An input for this part is letters of a single input word. The query parser separates single words from the input query. The second part provides a search for relevant documents based on the list of recalled words from the first part. In this article, we pay attention on the letter-word matcher part. We also focus on the exact match only.

2.1 Correlation Matrix Memory

Correlation Matrix Memories (CMMs) are type of associative memories (for a more detailed description see [1] or [2]). They can be thought of as a kind of neural networks. Fig. 2 shows an example of CMM and learning and recalling process.

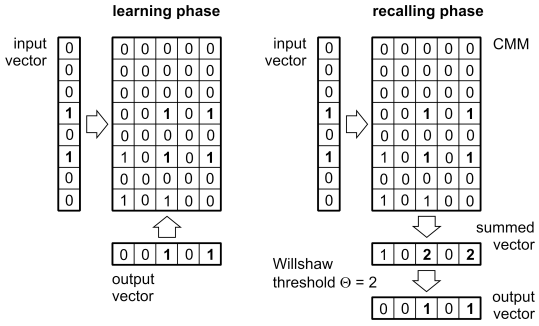

Fig. 2 Learning and recalling processes of CMM.

We use binary CMMs, so all weights (matrix cells), as well as input and output patterns, are binary. Therefore, they are very fast in training and recalling. Their simple architecture allows to implement them in hardware easily. An example of a hardware accelerator for this kind of neural networks is described in [3]. Analogous to other types of neural nets, they also have the ability to deal with noisy input patterns. They produce reasonable output on input patterns which were not in the training set, but are similar enough to others within the set.

The *learning process* is very fast in comparison to other neural nets since a CMM learns each association in one step (one pass through the matrix). During this step, the matrix cells $M_{i,j}$ which correspond to set bits in both input and output vector are set to one $I_i = 1 \wedge O_j = 1$. The results are *or*-ed to the matrix in each learning step. Thus an association pattern can partially overlap other one(s). Such overlapping can cause CMM to recall associations that have not been trained. Due to *or*-ing, the matrix will always find the stored association, but it can also recall other ones that were not in the training set, resulting in *false positive* errors. The false positive rate depends mainly on the mutual orthogonality of input patterns in the training set [1].

In our approach, we encode letters and words orthogonally, i.e. each letter sets single bit position in the input vector and each word is represented by one bit in the output vector. Thus, we do not have to deal with the false positives. This simplification makes recalling faster, because no validation is required on the output. On the other hand, more memory is required for a given number of associations.

In the *recalling process*, the input vector is applied to the input of the CMM. Matrix rows corresponding to set bits in the input vector are summed. The result is an integer vector, which is then thresholded to get the binary output vector. Commonly, we use the *Willshaw method* [4] that sets the threshold Θ equal to the number of set bits in the input vector $\Theta = |I|$. Using this value results in an *exact match*. Decreasing it results in a *partial match* based on the Hamming distance.

2.2 Letters and Words Encoding

A common method of letters and words encoding is shown on Fig. 3. As already mentioned, letter and word

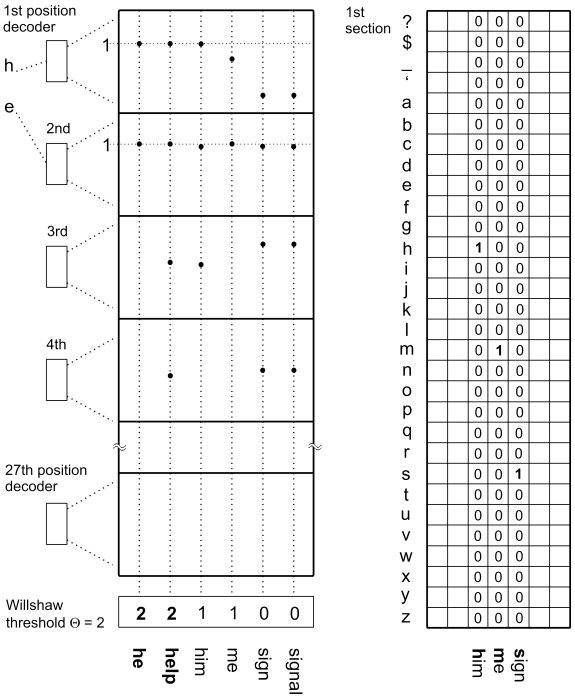

Fig. 3 Letters and words encoding, basic method.

patterns are encoded orthogonally. Each letter sets one bit in the input vector [5]. We count with 30 letters: 'A' to 'Z', underscore, apostrophe, and two special symbols. Each letter of the input word string represents one chunk with exactly one bit set. The chunks for all positions are concatenated to form the input pattern. One word position corresponds to one *horizontal section* of the matrix (see right side of Fig 3). On the output, each word from the lexicon is associated with one bit position of the output vector.

The matrix must be dimensioned so that the number of horizontal section are greater or equal to the length of the longest word in the lexicon [6]. In the collection of the Shakespeare works, we found the longest word to be 27 characters long yielding 810 rows. However, this approach leads to a prefix type of error on output words of different lengths. For example in Fig. 3, the word 'he' is correctly recalled whereas the word 'help' is not a correct answer.

2.3 Parallel Matrices

The prefix type of error could be overcome by comparing the word length after the recall [6]. However, we have used one matrix per word length resulting in 27 matrices for the Shakespeare collection. Overall, this saved 73 % of memory in comparison with a single matrix (594 KB vs. 2.17 MB). In our solution we used the input word length to select the correlation matrix to use. We could also save time by exploiting parallel processing of several query words with different lengths.

2.4 Testing Parameters

We used the following machine in our tests: Intel Pentium 4 1.6 GHz, 1 GB DDR SDRAM 333 MHz. We built a lexicon from a collection of the Shakespeare works [7]. It contains 24 k unique words. The longest word has 27 characters. The test set contains 47 k words where half of the words were randomly corrupted.

3 Shared Sections

The idea of a shared sections method is based on reusing one horizontal section with more letter positions. We use one matrix with hs sections, where hs is less than the maximum length of a word in the collection. Letters of an input word are spread in "modulo way" among the shared sections. An example of generating a binary vector from the word 'enter' for three sections is shown on Fig. 4. Letter 'e' is overlapped

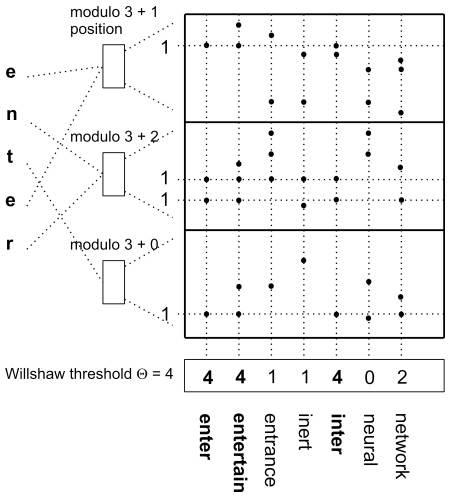

Fig. 4 Shared section method, $hs = 3$.

in the first section for the first and the fourth positions. Letters 'n' and 'r' is placed in the second section. And, letter 't' goes to the third section. Generally, a letter i is written to a section j when:

$$j = \begin{cases} i \bmod hs & \text{for} \quad i \bmod hs > 0 \\ hs & \text{for} \quad i \bmod hs = 0 \,. \end{cases} \tag{1}$$

Obviously, we introduce a new type of error due to sharing sections by more positions. In the example in Fig. 4 there are two falsely recognized words: 'entertain' and 'inter'. However, the tradeoff between the error rate and memory consumption becomes scalable via hs value.

3.1 Error Rate Definition

The error rate value reflects the amount of faults (in our case false positives) generated by a given method over a given collection. It is expressed as a percentage value from the maximum possible faults.

Let m be the number of words in the lexicon L, $|res_{w_i}|$ the number of recalled words for an input word w_i. Then, the error rate er is given by:

$$er = 100 \cdot \frac{\sum_{w_i \in L} |res_{w_i}|}{m(m-1)} \,. \tag{2}$$

Because a word is not its own false positive, the divider is $m - 1$ in Eq. (2).

3.2 Number of Bits Checking

For the shared section method, the number of bits set in the input vector can be less than the number of letters in the input word $|I| \leq |w|$ (for the case of overlapping same letters). Therefore, besides the length check handled by parallel matrices, we also check the results by comparing the number of bits along with the length check. It gives us further error rate reduction.

In the example from Fig. 4, the misrecognized word 'entertain' can be corrected by the length check (by the parallel matrices method), because they differ in length. But the word 'inter' cannot be corrected in this way as it has the same length. However, 'enter' and 'inter' differs in the number of bits set in the input vector. Thus, it can be fixed with the bit checking.

The improvement in error rate when using bits checking is shown in Tab. 1. The table also compares impact on memory and error rate for different numbers of hs.

Tab. 1 Shared section method results.

hs	1	2	3	5	10
ma	82.5	165	247	402	583
mr	7.2	3.6	2.4	1.48	1.02
fa^l	3.94	0.21	0.11	0.018	3.6e-4
fa^{lb}	0.32	0.03	0.01	2.7e-4	0
er^l	0.018	9.3e-4	5.1e-4	8.1e-5	1.6e-6
er^{lb}	0.0014	1.3e-4	4.3e-5	1.2e-6	0

ma is the absolute memory consumption (in KB) for parallel matrices with hs sections. mr is a decreasing factor of memory consumption in comparison to common method without shared sections (594 KB). fa^l is the average number of faults per word in the collection with the length check. fa^{lb} is same for length and bits check. er^{lb} and er^{lb} are the corresponding error rates.

Results shows memory reducing factor up to 7.2. In that case the matrices needs about 82 KB for the lexicon. But the average number of faults per word is relatively high 0.3. We think the reasonable tradeoff is $hs = 3$ with 2.4 reduction factor and low average number per word 0.01. The bits checking method reduces the error rate approximately by factor 10.

4 N-Gram Encoding

In this section, we evaluate an n-gram encoding method. We tested n-grams of order $o \in \langle 2, 5\rangle$ for memory consumption and the error rate. The n-gram method with CMMs has been already used in [6] to carry out the approximate match. We used different encoding scheme, we replaced letter symbols with n-gram symbols. Due to a big number of possible n-grams (see Tab. 2), we always use the shared section method with $hs = 1$ to keep memory consumption as low as possible. Errors caused by the shared section method (discarding position information) is reduced by the fact that n-grams already contains local letter position information. We also used both length bits checking methods.

N-grams are generated for each word position starting from 1 up to the position whose the end of the n-gram reaches the end of the word. Therefore, neighboring n-grams share letters. For example, the word 'sign' generates the following n-grams:

- three bigrams 'si', 'ig' and 'gn',
- two trigrams 'sig' and 'ign',
- one 4-gram 'sign' and
- one 5-gram 'sign?'.

The number of n-grams for word w and order o is given by $|w| - o + 1$. When the order of an n-gram is bigger than the word length a reserved symbol '?' completes the n-gram.

Tab. 2 shows the number of all possible n-grams for a given order and a alphabet size $|\Sigma| = 30$. These numbers are enormous, especially for order bigger than 2. Therefore, we use so called *active n-grams* only, which is a subset of all n-grams. Active n-grams are those ones, which appeared at least once in a collection. The numbers of active n-grams from the Shakespeare collection are shown in the Tab. 2.

Tab. 2 Numbers of all n-grams and active n-grams.

order	bigrams	trigrams	4-grams	5-grams
all	900	27000	810000	24300 k
active	506	4587	17415	30384

4.1 Boundary Letter

The boundary letter method extends the input word with a special symbol '$' (reserved in our alphabet) at the beginning of the word and at the end. The "boundary letter" puts the information about which n-gram is a starting one and which is a last one. As shown in our tests, it significantly contributes to reducing the error rate.

For example, the word 'sign' becomes '$sign$'. It corresponds to following n-grams:

- '$s', 'si', 'ig', 'gn', 'n$' for bigrams, or
- '$si', 'sig', 'ign', 'gn$' for trigrams, or
- '$sig', 'sign', 'ign$' for 4-grams, or
- '$sign$' for 5-grams.

The number of n-grams for a given order generated from a word w of order o is $|w|-o+3$. Tab. 3 compares the results of the error rate and memory consumption for n-gram methods for different orders. It also shows the error-rate improvements when using boundary letter enhancement.

Tab. 3 N-gram method results.

o	bigrams	trigrams	4-grams	5-grams
ma	1.36	12.3	46.8	81.4
fa^{lb}	8.9e-4	0	0	0
fa^{lbb}	8.9e-5	0	0	0
er^{lb}	4e-6	0	0	0
er^{lbb}	4e-7	0	0	0

ma is the absolute memory requirement (in MB) for active n-grams and $hs = 1$. fa^{lb} is the average number of faults per a word in a collection with length and bits check. fa^{lbb} is fa^{lb} with boundary letter enhancement. er^{lb} and er^{lbb} are the corresponding error rates.

Results shows that the proposed n-grams method gives mostly zero error rate even for $hs = 1$. Negligible number of errors appeared for bigrams only. However, the major problem is the memory consumption. For bigrams it is about twice as big as for the common method with parallel matrices.

4.2 Reduction of Active N-Grams

In order to reduce more memory consumption, we also tried further reduce the number of active n-grams. We created a table that contains the list of n-grams sorted according to their occurrence in the collection. We tested how many n-grams from the begin of the table we can use to keep the error rate at reasonable level. We also tested what happen if we moved a window of active n-grams into the middle of the table. The error rates for 200 most frequent bigrams are shown in Tab. 4.

Tab. 4 Reduction of active n-grams.

window shift	0 %	1 %	5 %
fa	0.028	0.033	0.18
er	4e-4	5e-4	2.8e-3

The results show that the most important n-grams are the most frequent ones. Unfortunately, we get about the same memory requirement as for the common method, but with some level of error rate.

5 Speedup of Recall

In this section we propose three methods to speedup the recalling process. They are suitable for software implementation of the correlation matrix memories. We tested and compared them with commonly used method described in subsection 2.2.

5.1 Columns Pruning

Our first proposed method, named *columns pruning*, is based on omitting matrix columns that can not longer affect the result list from the further processing.

Let us give an example. For the exact match, we use the Willshaw threshold (a hard threshold set to the number of bits set in the input vector). A winning column must then match all bits set in the input vector. Therefore, when we find a first mismatch we can omit a column where a mismatch occurred from the further processing. An example of the method when recalling the word 'helm' is shown in Fig. 5. We found a 84.7 % reduction (6.5 times) in processed matrix cells on the Shakespeare collection in comparison with the common method.

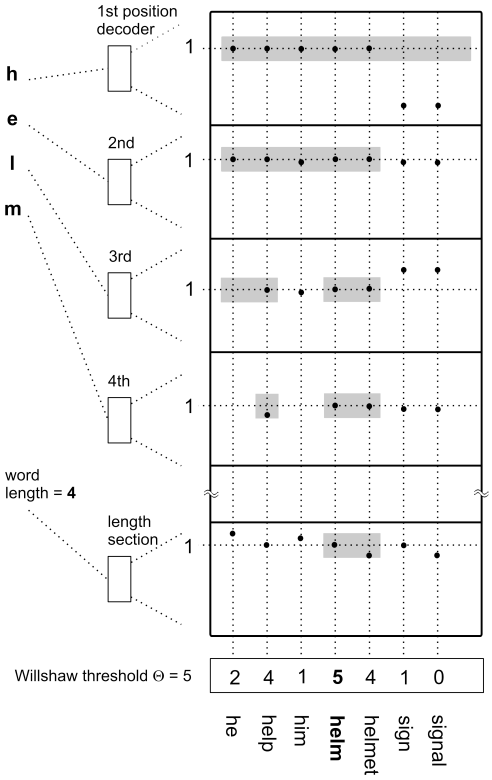

Fig. 5 Columns pruning method.

5.2 Index Matrix Representation

The second method is called *index matrix*. It reduces the number of processed matrix cells during recalling. We create a duplication of CMM (on a trained matrix), but in a different matrix representation. The index matrix stores only the active cells (matrix cells with value 1). Rows are composed from a list of active cell indexes. This method aims to omit processing of inactive matrix cells in the first processed row.

The indexes are offsets from the previous active cell in the row. We can do it in this way, because we always process the whole row. It makes the index values lower and saves the amount of memory required for the index matrix. We encode indexes in a similar way as for UTF-8 Unicode character encoding. Index can be represented as one, two or three bytes depending on its value. The highest bits in the first byte determine the number of bytes needed to represent the value. Most of the values fit into 1 byte range (about 90 %), so this is worth to encode it in this way.

In our test case, the index matrix is about 3.7 times smaller then its binary counterpart. While the binary matrices require about 594 KB, their index equivalents consume about 162 KB.

The index matrix method gives us further reduction of number of processed matrix cells. The reduction factor is 8 (for same test case) in comparison to columns pruning method. Combining both methods we save about 98 % of accessed cells, that is 52.4 times overall factor.

5.3 Hash Section

Previous two methods reduced the number od accessed matrix cells significantly. Recalling process quickly prunes false columns and converges into a proper one. Columns are keeping in processing as long as they match input letters. We process input letters sequentially, so those columns that has longest prefixes of an input recalled word are discarded at latest. The last proposed method was designed to recognize and discard those columns which have same prefix and differ in latter character(s).

We use a simple hash function from a string (see the following algorithm). First, we count an integer value v_{sum} based on all characters from the input word. Then, we apply modulo m on the sum v_{sum} to get the result value in range$\langle 0, m-1\rangle$.

1: $v_{sum} \Leftarrow 0$
2: **for all** $i = 1$ to $|w|$ **do**
3: $\quad v_{sum} \Leftarrow v_{sum} << 4 + w_i$
4: **return** $v_{sum} \mod m$

w_i is the ASCII value of a letter on i-th position in the word w. Value m is chosen to be a value of power of 2 minus one, and also to be the first bigger value then a quarter of the number of the original matrix rows M_i.

$$j = \arg\min_{j \geq 1}(2^j) : \; m = 2^j - 1 \; \wedge \; m \geq \frac{M_i}{4} \qquad (3)$$

We insert a new section before the first letter section. This section occupies r rows from the matrix. Each word has a single bit set in this section. We make the index matrix only for this section because we always start recalling with this section.

5.4 Speed Test

Tab. 5 shows experimental results for proposed methods. We have measure the speed by the average number of recalled associations per second. We have measured the exact match on the mentioned Shakespeare collection (see subsection 2.4).

Tab. 5 Speed comparison of proposed methods.

technique	mtd-a	mtd-b	mtd-c	mtd-d
assoc./sec	5202	43219	160528	1439147
speedup	1.0 x	8.3 x	30.9 x	276.7

mtd-a is the common recalling method, *mtd-b* is the columns pruning method, *mtd-c* marks the index matrix method and *mtd-d* is the hash section enhancement.

We achieved about 270 times speedup combining all three proposed methods comparing to the common method. The test showed approximately same speedup factor values as it has been predicted by counting the number of accessed matrix cells (see subsection 5.1) for the columns pruning and the index matrix methods. We also measured speed of the *binary search* method on the same test. It was about 900 thousand associations per second. So our proposed methods is about 60 % faster then the binary search.

6 Conclusion

In this article, we proposed the parallel matrix method that decreases the memory requirements by 73 % compared to standard implementation. The method resolves the prefix type of error and allows parallel processing of query words with different length.

We proposed shared section method that is able to further reduce memory requirement up to 7.2 times. However, this method gives imperfect results with some level of the error-rate. Our test showed that the reasonable tradeoff between the memory consumption and the error rate is for three horizontal sections. It reduces memory by factor 2.4 while still having reasonable average number of false positive per word 0.01. Proposed bits checking method reduces error rate by factor 10.

We evaluated the n-gram approach. Combining length and bits check with boundary letter method resulting mostly in zero error rate. But clear drawback of this method is the memory consumption, that is at least twice as big as for the basic method with parallel matrices. We also proved by a test that the most important n-grams concerning error rate are the most frequent ones.

We proposed three techniques for acceleration of the recall of software simulation of CMM. Combining all tree methods we achieved 270 factor speedup on the lexicon with 24 k words (overall 1.4 million associations per second). Comparing to a binary search method we process about 60 % more input queries at the same time.

This research is partially supported by the research program "Transdisciplinary Research in the Area of Biomedical Engineering II" (MSM6840770012) sponsored by the Ministry of Education, Youth and Sports of the Czech Republic.

7 References

[1] Aaron Turner. Modelling local transition functions in cellular automata using associative memories. Master's thesis, University of York, 1997.

[2] Mick Turner and Jim Austin. Matching performance of binary correlation matrix memories. *Neural Networks*, 10(9):1637–1648, 1997.

[3] Michael Weeks, Victoria J. Hodge, and Jim Austin. A hardware-accelerated novel IR system. In *PDP*, pages 283–, 2002.

[4] D. J. Willshaw, O. P. Buneman, and H. C. Longuet-Higgins. Non-holographic associative memory. *Nature*, 222(5197):960–962, June 1969.

[5] Victoria J. Hodge and Jim Austin. An evaluation of standard retrieval algorithms and a binary neural approach. *Neural Networks*, 14(3):287–303, 2001.

[6] Victoria J. Hodge and Jim Austin. A comparison of standard spell checking algorithms and a novel binary neural approach. *IEEE Trans. Knowl. Data Eng.*, 15(5):1073–1081, 2003.

[7] The complete works of William Shakespeare: http://shakespeare.mit.edu/.

[8] Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.

