Bachelor's thesis on

Fuzzy lexical matching

Marc Schoolderman

supervisor: Kees Koster
second reader: Marc Seutter

August 2012

Radboud University Nijmegen

Faculty of Science

Abstract

Being able to automatically correct spelling errors is useful in cases where the set of documents is too vast to involve human interaction. In this bachelor's thesis, we investigate an implementation that attempts to perform such corrections using a lexicon and edit distance measure.

We compare the familiar Levenshtein and Damerau-Levenshtein distances to modifications where each edit operation is assigned an individual weight. We find that the primary benefit of using this form of edit distance over the original is not a higher rate of correction, but a lower susceptibility to *false friends*. However, deriving the correct weights for each edit operation turns out to be a harder problem than anticipated.

While a weighted edit distance can theoretically be implemented effectively, a deeper analysis of the costs of edit operations is necessary to make such an approach practical.

Contents

1 Introduction

Human beings have little difficulty in interpreting texts with lexical errors in them; in fact it is usually hard for us to spot these, even if we are looking for them. On the other hand, we also find it difficult to correctly apply arbitrary rules – like verb conjugation, or the spelling of proper names. For computers, the reverse is true in both cases.

This distinction is actually helpful – the computer's ability to spot errors where we do not has long been exploited by spell-checkers. In its most familiar form, this is an interactive process. In some applications, we want computers to perform corrections unaided. For example, when digitizing existing archives, it would be prohibitive to manually remove all errors produced by an OCR-process.

In information retrieval we want to find information regardless of whether our spelling of a search phrase matches the one in a target document.

In this bachelor's thesis, we describe a system for automatically and efficiently correcting a class of errors by using a lexicon and approximate — or *fuzzy* — matching, and evaluate its effectivity.

1.1 Problem statement

With respect to fuzzy matching, we will try to answer the following:

> *Is it possible to efficiently implement and adapt the edit distance measure to improve word recognition when looking up approximate matches in a lexicon?*

This question has two different aspects. Firstly, we need an efficient way to store a lexicon and find exact matches in this, that is also easily adaptable for fuzzy matching. Since we assume errors to be infrequent, such an implementation will automatically be efficient *if* we can find a method of approximate matching that is not prohibitively expensive. Secondly, given such an implementation, how can we actually benefit from it?

This gives rise to a number of smaller questions, such as:

- Which algorithms and data structures can we use?
- Can we efficiently implement these?
- How should we modify the edit distance measure?
- How do we determine that, so doing, we have improved word correction?
- Can we use our implementation outside an experimental setup?

The layout of this thesis is as follows. The rest of this chapter mentions the motivation for this research. Chapter 2 presents techniques and a theoretical background, giving a toolbox for our implementation which is discussed in chapter 3. In chapter 4 our implementation will be applied against two simulated data sets and measure the effectiveness of fuzzy matching in this setting. Chapter 5 concludes this thesis and presents some further challenges that we do not address.

1.2 Applications of fuzzy matching

Fuzzy lexical matching has an obvious application in the construction of *spelling correction* mechanisms in familiar settings such as word processing. This is usually an interactive process, where the system generates a list of correction candidates and the user has to select the best one. We will therefore list other applications, most of which demand the *non-interactive* correction of errors due to the sheer size of the correction task.

Query-based search As noted in [5], a significant proportion (10-15%) of queries sent to a search engine contain errors. Users of present-day search engines are of course already familiar with this kind of correction, for example, a Google search for 'obma' assumes we are looking for 'Obama' and in fact shows these results for us.

But this form of correction only addresses one side of the search. The document collections searched for will also contain errors, and we should correct for these as well. Furthermore, these errors cannot be handled by an interactive spelling correction mechanism.

Document classification Every computer user will be familiar with the arms race between spam and spam-filtering to come up with new and creative ways to deliberately misspell words on the one hand and catch these on the other. For example, one website reports receiving 79 variations of the word 'Viagra' in only 12 days, and calculates that – using the tricks employed by spammers – there are over 10^{21} possible variations.[1]

Natural language processing The AGFL system[2] can correctly parse many documents using an ambiguous grammar– but at present all encountered word forms have to match its lexicon exactly. Being able to process natural language while tolerating errors would also be beneficial to *automated document translation*, which at present fails (usually badly) in their presence.

OCR correction Reynaert[26] reports about a project where a collection of historical Dutch newspapers has been digitized.[3] Except for errors induced by the OCR process itself, such projects also have to deal with changes in spelling that happen over time – especially in a language such as Dutch where spelling changes get introduced quite often. It is clear that in this case an interactive spelling mechanism is not an option, but also that digitizing such collections of text is quite valuable — both for preserving this information and making it more readily accessible.

Bio-informatics The BLAST program[4] allows one to search a genome for biological sequences. This tool also has to deal with the problem of judging whether two DNA-sequences are 'similar'. A feature shared with spelling correction or OCR post-correction is that in this application some differences in sequences are likely to be of less importance than others.

In this bachelor's thesis, we will focus on *non-interactive spelling correction*. Its application area therefore lies within the first three of the above categories, and could form a part of an OCR correction system as well.

[1] `http://cockeyed.com/lessons/viagra/viagra.html`
[2] `http://www.agfl.cs.ru.nl/`
[3] Available online at `http://kranten.kb.nl/`
[4] `http://blast.ncbi.nlm.nih.gov/`

2 Background

Before we can talk about correction, we need to be specific about the types of errors we want to correct. Kukich[16] provides a detailed classification of errors; we will reduce this somewhat.

First, we can distinguish between *non-word errors*, which are detectable at the lexical level – e.g. *definate* versus *definite*, and *real-word errors* which can only be detected by also looking at context such as syntax, semantics, or overall structure.

For example, out of context we have no reason to reject *ships*, but the sentence *'a ships was adrift'* is syntactically wrong. In another case dropping the *d* in *adrift* we might get *'a ship was a rift'* – which is syntactically correct, but nonsense.

We can further (loosely) classify non-word errors by the process which generates them into *transfer errors* and *cognitive errors*.

Errors induced by OCR will usually corrupt (multiple) characters that are orthographically similar – e.g. replacing *D* with *O*, or *ri* for *n*. Similarly, typos are more likely to consist of substitutions of letters close to each other on the keyboard or transpositions (*the*→*teh*). In all these cases the correct spelling of the word is available, but lost in transfer.

Cognitive errors occur when a user is not familiar enough with the proper spelling of a word, as in *definate* versus *definite*, or mistakenly uses a phonetically similar word-form (*they're* versus *their*).

Cognitive errors are more difficult for a computer to correct, since they are likely to result in a real-word error, and are more complex. For example, Pedler[24] finds that in a sample of dyslexic text, 39% of all errors differ in more than one letter, 8% are misplaced word boundaries, and 17% of errors are real-word errors.

We propose another class of errors: *variant errors*. These occur when the language of the source document is written in a different variant of the language of the lexicon. Such variations naturally occur; for example American vs. British English. In Dutch, various spelling reforms have resulted in a number of spellings being in simultaneous use.

In this thesis, we are only interested in non-word errors that are correctable at the typographical level. Furthermore, the word boundary problem will be considered a separate (albeit interacting) problem, and we will not attempt to solve it here. There are good reasons for this – in many cases splitting a word or joining two words will result in real-word errors, which we cannot correct without context information. Also, there are unavoidable ambiguities even using exact matching. For example, 'car wash' and 'carwash' are both common spellings of the same compound noun; but whether to identify the first form as a compound noun again requires contextual information.

2.1 Preliminaries

Conceptually, fuzzy lexical matching is the action of *searching* a lexicon for entries *similar to* a given word. We have already mentioned that we want to use a form of edit distance to determine word similarity. The first priority is therefore to ascertain whether can find an efficient method to calculate (multiple) edit distances.

The second priority is a good searching algorithm. Without it, the only way to perform fuzzy matching is to compute – given a corrupted word – an edit distance for *every* entry in our lexicon, which is clearly not acceptable if we want to efficiently use large, realistic lexicons.

2.2 Edit distance

The concept of an edit distance goes back to [17] in the context of binary codes. The idea is very simple: allow one string to match another by also allowing certain *edit operations*: replacing one character by another, inserting a new character, or deleting an existing one.

By counting the minimum number of operations necessary to transform one word into another we obtain a distance measure. For example, transforming *apple* to *able* requires two edit operations (deleting one p, transforming another p into a b). Today, this measure is usually referred to as the Levenshtein distance (or simply edit distance).

A simple definition of this distance measure is shown in figure 2.1.

```haskell
levenshtein :: String -> String -> Int
levenshtein s1 s2 = ed s1 s2
  where
    ed (a:as) (b:bs) = minimum [ ed as (b:bs) + 1
                               , ed (a:as) bs + 1
                               , ed as bs     + if a == b then 0 else 1
                               ]
    ed as bs = length as + length bs
```

Figure 2.1: Naïve implementation of Levenshtein distance in Haskell

This distance measure is a proper metric in the mathematical sense. By this we mean that the edit distance (ed):

1. is always non-negative: $\mathrm{ed}(x, y) \geq 0$
2. is zero if and only if two strings match exactly: $\mathrm{ed}(x, y) = 0 \Leftrightarrow x = y$
3. is symmetric: $\mathrm{ed}(x, y) = \mathrm{ed}(y, x)$; this practically follows from the definition and the fact that deletions and insertions are inverses of each other.
4. satisfies the triangle inequality: $\mathrm{ed}(x, z) \leq \mathrm{ed}(x, y) + \mathrm{ed}(y, z)$

Another classical paper[6] introduced transpositions of *adjacent* letters as a valid operation, for example allowing *teh* → *the* as a single operation. This is commonly referred to as the Damerau-Levenshtein distance. It is important to note that transposition can be achieved in the Levenshtein distance by an insertion and deletion, or equivalently, two substitutions; so the Damerau-Levenshtein distance will never be greater than the Levenshtein distance. Also, the usage of 'adjacent' is rather vague. Do we allow *owl* →*low* as two transpositions? Or what about, for example, *nei* → *in*? In both cases, we might end up transposing letters that were not adjacent in the source or target word. The original paper only allowed one edit operation, and so does not address this.

One problem with unrestricted transpositions is that it makes an efficient implementation complicated. Also, it is hard to see the benefit of assigning low values to complicated sequences of edits – as we will see in chapter 4, even low counts of edit operations can unrecognizably alter a word. Therefore, in this paper we will use the term Damerau-Levenshtein distance to refer to only a restricted modification (shown in a naïve implementation in figure 2.2) which only allows transposition of two characters that are adjacent in the original and the target word.

Note that this distance measure no longer satisfies the triangle inequality; for example, *emil*→*elm* has distance 3, even though *emil*→*eml* and *eml*→*elm* can each separately be performed by two single operations. For our purposes, this turns out to be of no consequence.

Obviously these implementations have exponential running times, so these can only be used for short words.

```haskell
damerau_levenshtein :: String -> String -> Int
damerau_levenshtein s1 s2 = ed s1 s2
  where
    ed as@(a:a':as') bs@(b:b':bs') | a == b' && b == a'
        = minimum (ed as' bs'+1 : edits as bs)
    ed as bs
        = minimum (edits as bs)

    edits (a:as) (b:bs) = [ ed as (b:bs)+1
                          , ed (a:as) bs+1
                          , ed as bs + if a == b then 0 else 1
                          ]
    edits as bs = [length as + length bs]
```

Figure 2.2: Naïve implementation of Damerau-Levenshtein distance in Haskell

```c
int levenshtein(const char *src, const char *dest)
{
    size_t x_max = strlen(src);
    size_t y_max = strlen(dest);
    int matrix[x_max+1][y_max+1];

    /* initialize the sides of the matrix */
    for(size_t y=0; y <= y_max; y++)
        matrix[0][y] = y;
    for(size_t x=0; x <= x_max; x++)
        matrix[x][0] = x;

    /* systematically fill the matrix */
    for(size_t y=1; y <= y_max; y++)
        for(size_t x=1; x <= x_max; x++)
            matrix[x][y] = minimum_of_3(
                matrix[x]  [y-1] + delete_cost(src [y-1]),
                matrix[x-1][y]   + insert_cost(dest[y-1]),
                matrix[x-1][y-1] + match_cost (src[x-1], dest[y-1])
              );

    return matrix[x_max][y_max];
}
```

Figure 2.3: $O(nm)$ implementation of edit distance using dynamic programming in C99

2.3 Dynamic programming

A useful technique to compute recursive algorithms (such as an edit distance) in polynomial time is dynamic programming. The general idea is that we compute each intermediate result exactly once.

A simple way to achieve this is to use *memoization*; after each recursive call, store the result in a table, and on subsequent calls re-use that value. This does however incur extra bookkeeping in having to maintain the memoization table.

A different, more elegant, approach is to look at the recurrence and identify a useful order in which all sub-problems are solved before they are needed to compute results that depend on them. This can be done by allocating a n-dimensional array where each cell corresponds to one invocation of the original recursive function. By initializing portions of this array with initial values and then using the recurrence repeatedly we can fill the entire array until we have computed the cell representing our answer.

Such an array-based approach for computing edit distances is presented in [29]. A concrete implementation of this approach is shown in figure 2.3. The idea is that each cell `matrix[`i`][`j`]` should be made to contain the edit distance needed to transform the first i characters of *src* into the first j characters of *dest*. During initialization, we can immediately fill in the top row and left-hand side column, since these correspond with a sequence of insertions and deletions (respectively). Following initialization we can fill in the rest of the matrix in a left-to-right, top-to-bottom fashion.

The matrix that results from matching the string *definite* to *deity* is shown in figure 2.4. Note that by examining the contents of the matrix it is also possible to deduce a sequence of edits to transform a word into another. Also note that there is more than one way of doing this, in this case because we have a choice about which i to delete.

		d	**e**	**f**	**i**	**n**	**i**	**t**	**e**
	0	1	2	3	4	5	6	7	8
d	1	**0**	1	2	3	4	5	6	7
e	2	1	**0**	**1**	**2**	**3**	4	5	6
i	3	2	1	1	**1**	**2**	**3**	4	5
t	4	3	2	2	2	2	3	**3**	4
y	5	4	3	3	3	3	3	4	**4**

Figure 2.4: Dynamic programming matrix showing the Levenshtein distance from *definite* to *deity*. The grayed path contains the sequences of edit operations obtaining the minimal distance.

It should be clear that given two strings of length n and m, this algorithm requires $O(nm)$ iterations. It also appears to require $O(nm)$ memory, but by observing that we only really need to store one row at a time, this can be brought down to $O(n)$.

Right now we can already construct a crude form of fuzzy lexical matching by brute force – simply computing the edit distance to every word in our lexicon. If our lexicon consists of N words of maximal length m, this would have a running time of $O(nm \cdot N)$.

Of course, large portions of the dynamic programming matrix do not contribute to a possible solutions. Various methods of computing the matrix more intelligently have been devised over the years. Only looking at the complexity figure is not the whole picture. Some algorithms have equal worst-case complexity but are faster in practice; others may have a better complexity but still perform slower in practice due to a high constant overhead. A general overview and comparison for some of these methods can be found in [20].

2.4 Levenshtein automata

If we are using the previous algorithm to compute the edit distance from both *definite→deity* and subsequently *definite→deified*, it is easy to see that there is no need to recompute a large portion of the matrix. A clever way is to compute the matrix resulting from *definite→dei* once and use it to perform the computation for both suffixes separately.

Observe that the computation of the i'th row of the dynamic programming matrix only requires the i'th character of the destination string. Therefore, we can refactor the algorithm in figure 2.3 as a state machine determined by the string `src`, processing characters from `dest` one at a time, where the state is represented by a row of `matrix`. Having fed the entire string into this machine we can read the corresponding edit distance from the state. Thus, we can duplicate the state of the machine after processing the common prefix *dei-* and avoid recomputing it.

If we fix a maximal edit distance k, we can also say that such a state machine is in an *accepting state* whenever the value in last column of its state is $\leq k$. In this way we have a finite state machine whose language consists of all strings within k edit operations of `src`.

2.4.1 Bit-parallelism – the `agrep` approach

The previous method used the dynamic programming matrix to define a state machine. It is also possible to directly define a non-deterministic finite state machine that acts as a Levenshtein automaton as follows. Given a alphabet Σ and a string $S \in \Sigma^*$, construct a finite state machine accepting S containing $|S| + 1$ states, and stack copies of these on top of each other, with progressive layers representing higher edit distances, adding transitions representing the various edit operations. By this we mean that if $s_{d,i}$ is the state in which we have matched the first i characters of S at distance d, there should be

1. a transition to $s_{d+1,i+1}$ for every $c \in \Sigma$; representing substitution
2. a transition to $s_{d+1,i}$ for every $c \in \Sigma$; representing deletion of c
3. an empty transition to $s_{d+1,i+1}$; representing the insertion of the $(i + 1)$'th character of S [1]

A diagram of such a state machine is shown 2.5.

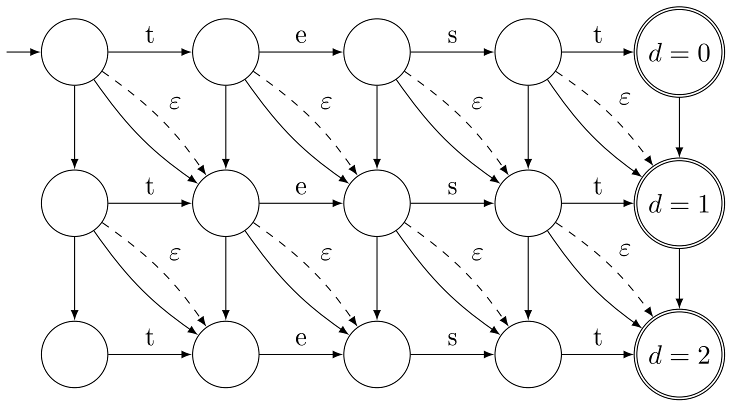


Figure 2.5: Non-deterministic finite state automaton accepting strings matching the word *"test"* with Levenshtein distance 2

[1] We use the convention that we want to edit S into some target string T, which is fed into the machine. These roles of course can be reversed

An efficient approximate string *searching* algorithm using this approach is implemented in the `agrep`[2] tool and presented by Wu and Manber[30]. In this approach, each row of the state machine is represented by a string w_i of $|S| + 1$ bits; with a set bit indicating that a state is active. The transitions are then applied in a single operation using only bit operations. For instance, the empty transitions can be implemented as the operation $w_i := w_i \ \underline{\texttt{or}}\ (w_{i-1}\texttt{<<}1)$. The horizontal transitions depend on the strings S, and are stored as bit masks that need to be computed only once.

Formally, the operation required to update the state after reading the character c is as follows; using the convention that $w_i = 0$ if $i < 0$, define define

$$\text{mask}_{c,j} \overset{def}{=} [j < |S| \text{ and } S_j = c] \tag{2.1}$$

$$w'_i = ((w_i \ \underline{\texttt{and}}\ mask_c)\texttt{<<}1) \ \underline{\texttt{or}}\ w_{i-1} \ \underline{\texttt{or}}\ (w_{i-1}\texttt{<<}1) \ \underline{\texttt{or}}\ (w'_{i-1}\texttt{<<}1) \tag{2.2}$$

In many cases, $|S|$ is less than the word size on modern computers (typically 64 bits), and these bit operations take constant time. The end result is that for a string of length m, we can determine whether it matches S within edit distance k in $O(km)$ operations, using k machine words for state. Set up of the automaton requires only initialization of the bit mask. This costs $O(n)$ operations and $|\Sigma|$ machine words.[3]

An advantage of this algorithm for string searching is that it is fairly extensible. For example, adding transpositions is not much work. Also, only minimal changes are needed to support a limited subset of regular expressions – for example, changing the machine in figure 2.5 to accept the regular expression "`tes*t`" can be done by adding just a few transitions. Some examples of extensions are already provided in the original presentation[30].

For approximate matching, another advantage is that it is easy to determine the *minimal* edit distance at which the automaton may still enter an accepting state – determining the lowest i so that $w_i \neq 0$ suffices. We can do this in constant time time while calculating the next state w'_i.

2.4.2 Other approaches using bit-parallelism

Since its introduction, optimizations to the above solution have been proposed. One such approach is presented by Baeza-Yates and Navarro[2]. They observe that, since there is an empty diagonal transition, if a state is active, then so are all subsequent states on the same diagonal. Instead of encoding the rows of the machine state directly by a bit representation, they encode the *diagonals* by listing (for each diagonal) the minimum row value on which each diagonal is active. Furthermore, they then encode these values in a bit string using a unary representation, and show that once again they can employ bit parallelism to compute transitions efficiently.

For small patterns S, their solution gives a $O(n)$ algorithm to match a string of length n allowing for k errors (compared to $O(kn)$ above). However, the disadvantage is that even using 64-bit machine words, this only works for $|S| \leq 14$, which is quite limited.

A different technique employing bit-parallelism is due to Myers[19]. This method constructs an automaton based on the dynamic programming matrix. Starting from the observation that all horizontal and vertical differences in the matrix are at most ± 1, a number of clever transformations are performed resulting in an $O(n)$ algorithm as long as $|S|$ is less than the machine word size. A single drawback of this method is that there is no obvious way to find (in constant time) the minimal edit distance at which the automaton might still accept the input. This is not a problem for for approximate *searching*, but does prevent its use in an informed search(see section 2.6.3).

[2] `ftp://ftp.cs.arizona.edu/agrep/`

[3] This implies that the worst case complexity for fuzzy lexical matching using this method has complexity $O(n + km \cdot N)$, compared to $(nm \cdot N)$ needed to calculate the dynamic programming matrix N times — certainly an improvement if k is low

2.5 Generalizing the edit distance using confusions

An observation already made in [6] is that most spelling errors are one of the four operations already described. This is what makes an edit distance a useful distance measure.

However, a drawback of edit distance is that it treats all edit operations as equally important. So inserting a q or changing a e into an k are deemed as likely as deleting an e or changing a n into an m, since all of these have 'cost' 1.

Second, the edit distance is too rough a measure. In order to correct for common mistakes we may need two or three edits. But given a reasonably large lexicon there will already be many words matching any given word within two edit operations. This especially problematic since we need a large lexicon to perform lexical matching.

One way to remedy this is to assign different costs to each edit operation. This is a fairly easy change. If Σ is our alphabet, it requires us to construct a $(|\Sigma|+1)\times(|\Sigma|+1)$ matrix `d`; where `d[`α,β`]` contains the cost of replacing α with β, with α and β either characters or the *empty* character $\varnothing$. For example, the cost of inserting the letter a can then be looked up as `d[`$\varnothing$`,'a']`, and we should expect that for any arbitrary character α, `d[`α,α`] = 0`. We will call this a *weighted edit distance*.

Another possible way to assign costs to edit operations is shown in [12] and [22]. Instead of one cost matrix, we can use four:

- `sub[`α,β`]` – the cost of substituting β for α
- `rev[`α,β`]` – the cost of transforming a substring $\alpha\beta \rightarrow \beta\alpha$. In other words, this assigns a cost to the act of transposing two characters.
- `add[`α,β`]` – the cost of inserting β after α
- `del[`α,β`]` – the cost of deleting β after α

A significant improvement here is that defining costs this way allows for a limited form of *context* to guide the cost of an edit operation. For example, using these tables it is easy to assign a lower cost to deleting the superfluous l in *definitelly* than deleting a l in general. We will call this the weighted edit distance *with context*.

Both weighted variations are easy to implement using dynamic programming. However, implementing a *weighted edit distance* using bit-parallelism means we also need transitions than can go down more than one row. In the `agrep` approach this would increase the cost of feeding a character into the automaton from $O(k)$ to $O(k^2)$ steps – and since we need k to be larger if we are using a weighted edit distance, this approach quickly looses its appeal.

2.5.1 Other extensions to the edit distance

In [23], the weighted edit distance as described above is also combined with a set of *constraints* that specify bounds on the number and type of operations. For example, disallowing string transformations which involve more than k insertions. In a generalized Levenshtein distance this may be of use to set a reasonable limit on low-cost edit operations. However, the algorithm has an higher algorithmic complexity than those for an unconstrained edit distance – $O(nm^2)$ compared to $O(nm)$ for the dynamic programming solution – and it raises the question what a set of reasonable constraints should look like.

A similar modification is that of a *normalized edit distance*[18]. In this case the cost of a sequence of edit operations is divided by the length of that sequence (with an exact match also counted as an operation). This has the effect of penalizing large corrections on smaller words and tolerating more corrections on larger words. However, as before, the algorithm to compute this is less efficient.

The idea of generalizing the edit distance can also be taken to its logical conclusion, by lifting the restriction that all edit operations concern *characters*. In this case, α and β may be arbitrary strings, and the cost matrix assigns costs to substitutions of the form $\alpha \rightarrow \beta$. This approach is explored in [4]. The drawback of this approach is (again) that calculating the edit distance becomes even more costly – $O(n^2m^2)$ – and that we need a rich set of edits and associated costs.

2.6 Best-first search

We already mentioned one way to perform fuzzy matching so far – simply calculating the edit distance to every word in our lexicon. This is obviously not very efficient. A better approach is to use a search strategy so that we only inspect the subset of our lexicon that is 'likely' to match. This prompts us to look in the direction of an *informed search strategy.* A conceptually simple but very useful strategy is the best-first search[11].

A well-known example of best-first search is the A* algorithm[8]. This algorithm finds an optimal path between two nodes in a graph, using a heuristic $f(x)$ that estimates the cost to go from any given node to the desired destination. It does this by maintaining an *open set* consisting of all nodes that we can travel to from nodes already visited (the *closed set*). Initially the *open set* will only contain the starting node. At each step the algorithm selects a node from the *open set* with a minimal value of $f(x)$ to visit next.

If these nodes are points are in a Euclidean space (for example, a map), a useful heuristic is the sum of the distances necessary to travel to the node x via the graph, and the distance needed to travel from x to the end point directly in a straight line.

The general procedure for best first search is shown in figure 2.6. Note that explicitly maintaining the *closed set* is also referred to as negative memoization[11], and can be considered a form of dynamic programming. If it is known that the graph (or search space) we are processing does not contain cycles (for example, when we are searching through a tree) we can dispense with this step.

This search algorithm can also be used in different contexts. For example, we can also use it without specifying any explicit goal node to simply find a shortest path to *any* node. Another possibility is to use it to calculate the dynamic programming matrix of figure 2.4. We can do this by starting in the top right corner and at each step calculate the values of the three neighbouring cells to the left, bottom and diagonally. In this case we can be sure that all cells with values higher than 5 will never be calculated, and if we use a good heuristic function we might only need to calculate a small portion of this matrix surrounding the grayed path. Choosing the heuristic poorly, however, would mean we still calculate the majority of the matrix and pay for the added overhead of our search strategy and might be slower than the algorithm in 2.3.

In a more advanced application, [11] uses a best-first search strategy to produce a text formatting algorithm that (on average) is faster than the one used by TEX.

2.6.1 Formal analysis

Important properties of the A* algorithm are already proved in [8] and [9]. However, this is dealt with mostly in the context of finding optimal *paths* in a graph where external information is available that can be used to steer the search algorithm.

If we are not interested in finding shortest paths, and have no extra information, a best-first search strategy can still be useful. Therefore we will take a slightly more abstract look.

Let the search space consists of a finite set of states H, where we also have a transition relation $(\rightarrow)$ between states. Let $\rightarrow^*$ denote the transitive closure of this relation. Let O_i and C_i denote the *open set* and *closed set* at the beginning of step i of the algorithm, and $\sigma_i \in O_i$ the node selected after this step, so that $f(\sigma_i) = \min f(O_i)$, with $f : H \rightarrow \mathbb{R}$ our heuristic function. The closed set contains all nodes selected previously, so $C_i = \{\sigma_k : k < i\}$. We can then define O_0 to be our initial set of states (usually a singleton set), and $O_{i+1} = (O_i \cup \{\tau : \sigma_i \rightarrow \tau\}) \setminus C_{i+1}$. Clearly this algorithm must terminate at step i if $O_i = \varnothing$. Let the set $I \subset \mathbb{N}$ denote the indices of all non-terminating steps, i.e. the set of indices for which σ_i is defined.

We further define a set of target states $T \subset H$ that are reachable from our initial states (that is, for each $\gamma \in T$, $\sigma \rightarrow^* \gamma$ for some $\sigma \in O_0$), and an actual cost (or distance) function $D : T \rightarrow \mathbb{R}$. The objective of a best-first search is to find the states of T in increasing order of their distance value. As a consequence we immediately have a simple way to find a minimal result in T: simply take the first target state encountered.

```
bool bestfirst_search(Node start, Node goal)
{
    priority_queue<Node> open;
    set<Node> closed;

    open.push(start);
    while(!open.empty()) {
        Node cur = open.top();
        open.pop();

        if(cur == goal)
            return true;

        if(closed.insert(cur).second) {
            for(Node n : cur.next)
                open.push(cur);
        }
    }

    return false;
}
```

Figure 2.6: Skeleton for performing best-first search (C++). The result value only specifies whether the search succeeded or not. Note that by replacing the `priority_queue` by `stack` we get a simple depth-first search; replacing it with `queue` gives a breadth-first search.

In order for this to succeed we need two requirements on the heuristic function f:

1. f must be monotonic, that is that if $\sigma \to \tau$, then $f(\sigma) \leq f(\tau)$
2. f must be equal to D for nodes in T, i.e. $f(\gamma) = D(\gamma)$ for all states $\gamma \in T$.

   As a consequence, f may never over-estimate the eventual cost for nodes not in T. If $f = D$, then obviously this requirement holds automatically.

Lemma 1. *For $i, j \in I$, if $i < j$ then $f(\sigma_i) \leq f(\sigma_j)$.*

Proof. It suffices to show that $f(\sigma_i) \leq f(\sigma_{i+1})$.

Since f is monotonic, $f(\sigma_i) \leq \min f(\{\tau : \sigma_i \to \tau\})$. Combining this with $f(\sigma_i) = \min f(O_i)$, we get $f(\sigma_i) = \min f(O_i \cup \{\tau : \sigma_i \to \tau\}) = \min f(O_{i+1} \cup \{\sigma_i\})$, and so $f(\sigma_i) \leq \min f(O_{i+1}) = f(\sigma_{i+1})$. □

To finish our proof of correctness, we also need to prove that the first target state found is a minimal result in T. To do this we only have to show that every reachable state will eventually selected. This part of the proof does not depend on f. We begin with a simple lemma.

Proposition. *The search procedure selects each node only once. That is $i \neq j \Rightarrow \sigma_i \neq \sigma_j$ for $i, j \in I$.*

Proof. Suppose $i \neq j$. Without loss of generality, assume that $i < j$. Then $\sigma_i \in C_j$, whence $\sigma_i \notin O_j$. But since $\sigma_j \in O_j$ it must be that $\sigma_i \neq \sigma_j$. □

Lemma 2. *If $\tau \in O_i$, then there is a $k \geq i$ so that $\tau = \sigma_k$.*

Proof. Suppose that no such k exists. Then for all $j \geq i$, $\tau \in O_i$. This means that our algorithm never terminates, and $I = \mathbb{N}$. But then, because of the previous proposition, the set $\{\sigma_i : i \in I\} \subset H$ is infinite. But H is finite – contradiction. □

Corollary. *If for a given τ there is a $\sigma \in O_0$ so that $\sigma \to^n \tau$, then there is a i so that $\tau = \sigma_i$.*

Proof. By induction on n. If $n = 0$ then $\sigma = \tau$, so $\tau \in O_0$, and so there exists i so that $\tau = \sigma_i$.

Now suppose that $\sigma \to^n\!\to \sigma' \to \tau$. Then by the induction hypothesis there is a i so that $\sigma' = \sigma_i$. But that means that $\tau \in O_{i+1}$, and so there is a $j \geq i + 1$ for which $\tau = \sigma_j$. □

Corollary. *The first target state found using best-first search has minimal distance.*
Formally: let $k = \min\{i \in I : \sigma_i \in T\}$. Then $D(\sigma_k) = \min D(T)$.

Proof. Because γ is reachable there is an i so that $\gamma = \sigma_i$. But then we have $k \leq i$ by definition of k, and so $f(\sigma_k) \leq f(\gamma)$ by Lemma 1. But then also $D(\sigma_k) \leq D(\gamma)$. □

We have now proven the correctness of this method, since we have shown that every target state will be selected in the order of their actual cost.

Infinite search spaces

From a theoretical viewpoint, it should be mentioned that the proof above relies on the search space H being finite. In fact a best-first search does not work *in general* if H is infinite. But it can be made to work on infinite search spaces if we impose additional restrictions so Lemma 2 can be proven again. A general way to ensure this is to require that for each $y \in \mathbb{R}$, the set $\{\tau \in H : f(\tau) \leq y\}$ *is* finite – we will however not digress further into this.

2.6.2 Best-only search

A slight modification of best-first is the best-only search. In this case we stop looking at states when their heuristic value is worse than the best target state found so far. Practically, this can be implemented by maintaining a cut-off threshold as in a branch-and-bound algorithm. Semantically, this extends the closed set as follows:

$$C_i = \{\sigma_k : k < i\} \cup \{\alpha \in H : \text{there is a } \sigma_k \in T, k < i \text{ such that } f(\alpha) > f(\sigma_k)\}$$

This also implies that the search algorithm will terminate as soon as there are no more states with a promising heuristic value. And, since the first target state selected will already have the best value, this means that the search space greatly reduces after the first target state found.

However, note that combining any other search strategy with the above cut-off would still ensure $i < j \Rightarrow D(\sigma_i) \geq D(\sigma_j)$ for $\sigma_i, \sigma_j \in T$. So a best-first search strategy is not necessary to construct a best-only search if we are willing to discard a few sub-optimal results, and can compute f easily for arbitrary states.

If we are only interested in a best result if it is *unique*, we will call this an *unambiguous best-only* search. In such a search, we also lower the threshold whenever we find two target states having the same D-value. In a pure best-first search, lowering the threshold after we already found a target state will result in immediate termination, in other search strategies it merely prunes the search space.

2.6.3 The heuristic function

In order for a best-first search to work, the function f needs to be easily computable. In the A* algorithm, this is achieved by defining f as the sum of a path-cost function, g, which gives the minimum cost needed to reach a state (with $g(\gamma) = D(\gamma)$ for states in T), and an estimator (say, h) that should not over-estimate the needed cost to reach the target state (with $h(\gamma) = 0$ for $\gamma \in T$).

If we do not know in advance what our target state is, defining $h(x)$ is rather non-trivial. But it is always safe to define $h(x) = 0$. The path-cost function g can however be constructed incrementally at each step. That is, the value of $g_0(\sigma)$ is 0 if $\sigma \in O_0$, and ∞ otherwise. We then recursively define

$$g_{i+1}(\tau) = \min\{g_i(\tau), g_i(\sigma_i) + \mathcal{V}(\sigma_i \to \tau)\}$$

Here $\mathcal{V}(\sigma_i \to \tau) \geq 0$ denotes the cost of taking this transition, assumed to be infinite if no such transition exists. If we modify the best first-search to use $g_i(\sigma_i) = \min g_i O_i$ instead of f, we will prove a variant of Lemma 1 to show that this modification is still sound.

Proposition. *If $i < j$, then for each $k \geq i$, $g_k(\sigma_i) \leq g_k(\sigma_j)$.*

Proof. By induction it follows that for each $k \geq i$, $g_i(\sigma_i) = g_k(\sigma_i)$. Similarly, $g_j(\sigma_j) = g_k(\sigma_j)$ for $k \geq j$. If $i \leq k < j$, then it is easy to see that $g_j(\sigma_j) \leq g_k(\sigma_j)$. Combining these two facts we find that $g_j(\sigma_j) \leq g_k(\sigma_j)$ for all $k \geq i$.

So it suffices to show that $g_i(\sigma_i) \leq g_j(\sigma_j)$. We do this by showing that $g_i(\sigma_i) \leq g_{i+1}(\sigma_{i+1})$.

Since $\sigma_{i+1} \in O_{i+1}$, at least one of $\sigma_{i+1} \in O_i$ or $\sigma_i \to \sigma_{i+1}$ holds. We know that if $\sigma_{i+1} \in O_i$, then $g_i(\sigma_i) \leq g_i(\sigma_{i+1})$, since the algorithm selected σ_i over σ_{i+1} at step i. But then it is easy to see that $g_i(\sigma_i) \leq \min\{g_i(\sigma_{i+1}), g_i(\sigma_i) + \mathcal{V}(\sigma_i \to \sigma_{i+1})\}$. $\square$

2.6.4 Example

We can compute any weighted edit distance between two strings n and m using a best-only search and an incremental heuristic.

Let the search space be $H = \mathbb{N} \times \mathbb{N}$, where the tuple $(i, j) \in H$ is the state in which we have matched the first i characters of n to the first j characters of m. Obviously $O_0 = \{(0, 0)\}$, and $T = \{(|n|, |m|)\}$.

Now, the transition relation corresponds to the edit operations allowed. For example, $(i, j) \to (i + 1, j)$ would correspond to a delete on the $(i + 1)$'th character of n. Finally, $\mathcal{V}(e)$ is simply the cost associated with the edit operation corresponding to the transition e.

2.7 Summary

In closing, we see that there are at least two useful ways to add costs to an edit distance, and various efficient ways to compute edit distances. Also, we have a a search strategy which can serve in many ways like a Swiss army knife.

3 Implementation

Part of our problem lies in showing how to construct an efficient way to match input words in a lexicon, allowing for edit operations to occur. Since we are focusing on natural language applications, we should assume that in most cases words will not contain errors and that therefore an exact match suffices. If we do this efficiently, we can immediately optimize our approximate matching method by simply not using it often, and always trying to find an exact match first.

3.1 Constructing a lexicon

In earlier times, even the problem of finding exact matches required innovative solutions such as Bloom filters [3] due to memory constraints. In recent times even modest computers have enough memory to keep large lexicons in memory, and we have well-known data structures to do this.

Following [10], we consider three efficient candidate structures to for this task. Hash tables, binary trees and trie structures. Of these, hash tables are considered the most efficient structure, and allow for a space/time trade-off. However, hash entries are by design randomly distributed over a hash table, and performing a single edit operation on a string will produce a very different hash value than it had originally – in fact there is no relation between similarity and location in the data structure. This means that the only way to perform a fuzzy match using a hash table would be a brute force search, which is not acceptable.

Binary search trees *do* preserve key ordering, but these are still not much help – the relative position of a keys in a binary search tree is dictated by the need to keep the tree as a whole balanced, not by key similarity. On top of this, binary trees are slower than other solutions when used for exact matching since at each node a complete string comparison would have to be performed.

A trie structure seems a promising choice: these are fast, and the position of a key in a trie is directly related to the contents of the key itself – allowing the retrieval of similar keys more easily.

3.2 Tries

Knuth[13] provides us with an overview of the family of trie structures. In its simplest form, given an alphabet Σ, a *trie* is a $|\Sigma|$-ary tree storing strings of characters. Whereas in a binary search tree at each node the decision to go left or right would (in our case) depend on the outcome of a string comparison, in a trie we have more than two outgoing edges, with each outgoing edge of a node corresponding to a single character. To find a string S in a trie, we start at the root of the trie, and choose the outgoing edge corresponding to the first character of S. We then arrive at a new node and repeat this process for the second character in S, and so on, until either one of two things happen:

- We have exhausted all characters in S. In this case, if the node found is represents a stored key, we have found S. If not, our search has failed.
- We arrive at a node where no outgoing edge matches the next character in S. In this case too S is not contained in the trie.

We call this a *simple trie*. A graphical representation can be seen in figure 3.2.

```haskell
data Trie = Trie [(Char,Trie)] Bool

contains :: String -> Trie -> Bool
contains []     (Trie _ isKey) = isKey
contains (c:cs) (Trie subtries _)
 = case lookup c subtries of
     Just trie -> contains cs trie
     Nothing   -> False
```

Figure 3.1: A simple trie lookup implemented in Haskell

Compact tries Various optimizations to this basic structure are possible. For instance, we can see from figure 3.2 that a simple trie uses many intermediate nodes that can only lead to a single leaf node. These essentially encode the suffixes of words. An compact trie removes these by storing the suffixes directly; in figure 3.2 we have indicated this by underlining the part of the word that needs to be stored in the node itself.

Patricia tries This optimization still leaves nodes that only have one outgoing edge. A different technique called the Patricia trie removes such nodes altogether by collapsing all simple chains of nodes. Note that a Patricia trie is still a $|\Sigma|$-ary tree, even though each outgoing edge is now labelled by a sequence of characters — since no outgoing two edges from a single may start with the same character. This means that to store the words *freedom* and *freeze*, we are forced to add them as children of an internal node representing the common prefix *free-*, as shown in figure 3.2.

Digital search trees However, the number of nodes can be reduced further. In a compact trie, only leaf nodes contain stored suffixes. If we lift this limitation, we can in fact use *every node* in the trie to represent a word.

This corresponds to constructing what Knuth[13] calls a $|\Sigma|$-ary digital search tree. In this case, we traverse the tree structure in exactly the same manner as we would in a compact trie – the difference being that each node should be treated as an internal node and 'leaf' node at the same time. For example, to find the word *freeze* in figure 3.2, we start at the root, noting that it is not identical to *fact*, and take the edge corresponding to first letter, *f*. We then find that the suffix *-reeze* matches the remainder of our search string, and are done.

Note that Knuth[13] only describes *binary* digital search trees in detail. In such a search tree, each node only has two outgoing edges, with the next *bit* of the word searched for controlling whether to go left or right. However, such a search tree would have drawbacks similar to an ordinary binary tree when used for fuzzy lexical matching.

In the number of nodes required, a digital search tree is the most parsimonious of the trie variants, since we only need a single node per word. Another advantage is that we can re-use the data structure required to implement a compact trie. In fact the only difference lies in the insertion algorithm. Finally, we have a degree of freedom on where to place a word in the search tree. This could be used, for example, to place words that are likely to be more common nearer to the root of the tree, and perhaps even to dynamically optimize the shape of the trie during lookup.

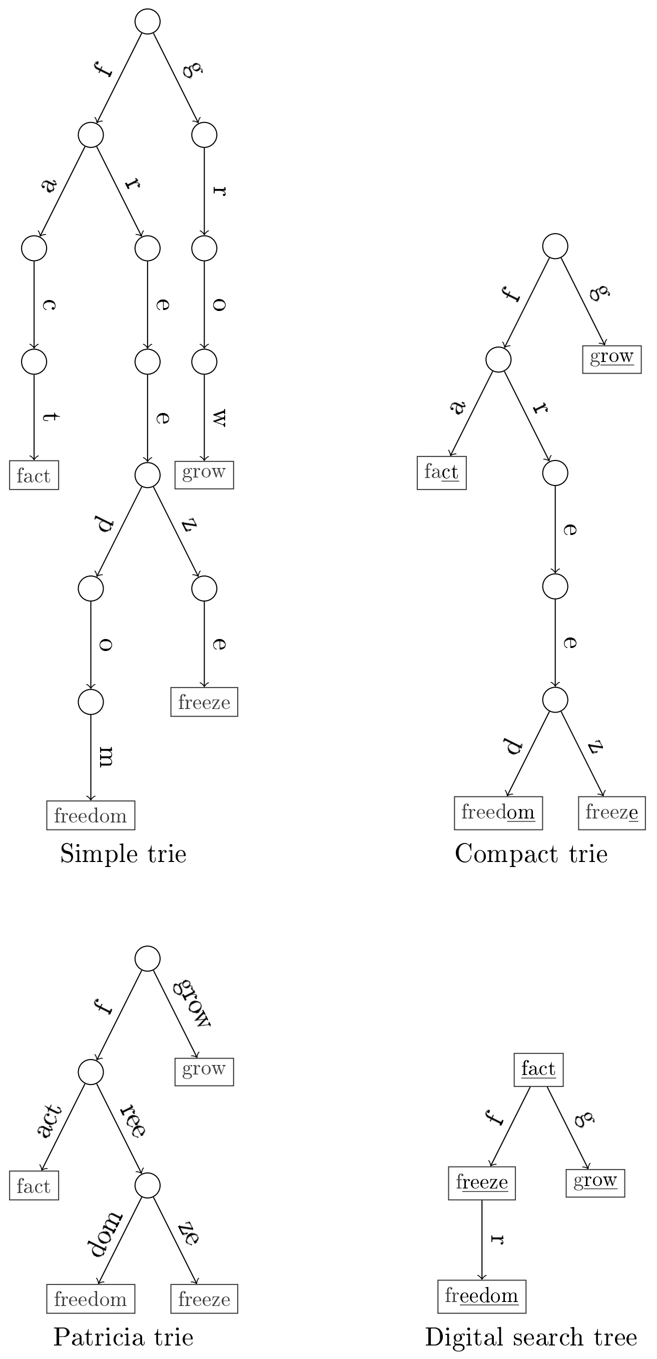


Figure 3.2: Examples of tries containing the words: fact, freedom, freeze, grow

3.3 Theoretical complexity

Assuming our alphabet Σ is finite, we can determine the theoretical complexity of looking up a string of length n in a trie.

Proposition. *Finding an exact match in a trie has complexity $O(n)$.*

Proof. Given that every node in a trie has at most $|\Sigma|$ outgoing edges, and each character corresponds to taking at most one edge, we see that an upper bound is $O(n \cdot |\Sigma|)$ character comparisons. However, since we can assume Σ to be fixed, $|\Sigma|$ is a constant, and so the true complexity is $O(n)$. □

Note that this proof does not depend on what type of trie we are using.

However, in the application we are interested in, we can be more precise about the true cost of this figure, by making a simple observation:

Assumption (Property of languages)**.** *The probability of two words sharing a common prefix P of length n decreases rapidly as n gets larger.*

For a trie structure, this means that beyond the first few levels, the arity of our nodes should rapidly become low, essentially a constant. We can verify this quickly. If we load the RIDYHEW word list[28] containing over 450000 English word forms as a simple trie, we can calculate the average arity of internal nodes at the various levels. As can be seen in figure 3.3 – after the first 4 levels, there is a sharp drop-off in the arity of internal nodes. In fact we can see that nearly all nodes in the trie are of low arity. Notable exceptions to this are illustrative as well. For example, the nodes corresponding to the prefix ‘centimilli’ and ‘centimicro’ have a high arity – but these are very rare words. From this we conclude that in the average case, looking up a word in a trie will in actual fact take much less than $n \cdot |\Sigma|$ character comparisons.

In terms of memory complexity, a counting exercise shows that a simple trie containing N words of maximal length m has $O(m \cdot N)$ nodes. On the other hand, a digital search tree will have exactly $N + 1$ nodes, each containing a string of $O(m)$ characters.

In comparison, hash tables have *amortized* $O(m)$ lookup; and in a balanced binary search tree we can expect a worst case of $O(m \log N)$ comparisons. However, if we take the natural language property discussed above into account, this can be reduced to an average complexity of $O(m+\log N)$ — since any string comparison between two arbitrary strings will on average fail after a constant number of steps. This – perhaps surprisingly – means that the complexity of a binary search tree is not bad, especially if we are often looking up long words in modestly-sized dictionaries.

3.4 Practical performance

Theoretically, we have seen that all tries have equal complexity. Practically, there are large differences. We can have a significant impact on the efficiency and memory footprint by choosing one variant over the other. Also, there is considerable degree of freedom in representing the outgoing edges in each node of the trie.

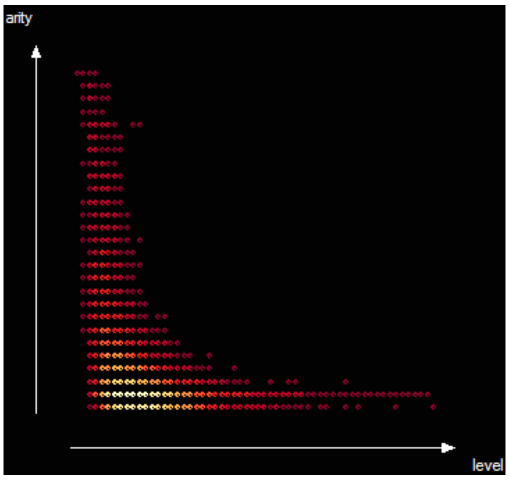
The easiest way to represent a simple trie in C++ is the following structure:

```
struct trie {
    trie *next[256];
    bool is_key;
};
```

While ostensibly quick – finding the next node in a trie takes only a single lookup, consider again that even small dictionaries require many nodes. On a 64-bit machine the above structure requires over 2 kilobytes per node. Looking at the number of nodes of figure 3.3, it becomes clear that storing a dictionary this way would require over 2 gigabytes.

But there are better alternatives instead of using an direct array. We list two:

level	nodes	average arity
0	1	26.0
1	26	14.3
2	373	10.4
3	3888	5.6
4	21667	2.8
5	59966	1.7
6	104004	1.3
7	140092	1.1
8	156525	1.0
9	152787	0.9
10	134789	0.8
11	108934	0.7
12	81352	0.7
13	56709	0.7
14	38035	0.6
15	24526	0.6
16	15353	0.6



Arity of nodes (y-axis) plotted against level inside the trie (x-axis), Brighter colors indicate more frequent occurrences.

Figure 3.3: Arity of nodes in a simple trie structure storing the RIDYHEW word list.

Pointered structures By storing the outgoing edges in a *binary tree*, we end up with a data structure also known as a ternary search tree:

```
struct trie {
    trie *left, *right;
    trie *next;
    char c;
    bool is_key;
}
```

In this structure the `left` and `right` pointers point to siblings of the current node, whereas `next` is the node we may take if we can match the current character `c`.

Instead of a binary tree, we can also simply obtain a simple *linked list* by omitting the `left` pointer from this structure – this may gain us more by saving on memory.

Dynamic arrays The downside of the previous structures is that they involve pointer chasing. A way around this is to represent the association list as a single contiguous object, by using a (resizable) array. For instance, an implementation using the standard C++ `vector` container looks very similar to the definition in Haskell:

```
struct trie {
    vector<pair<char,trie*>> next;
    bool is_key;
}
```

The downside is that, compared to pointered structures, we pay an overhead for maintaining a dynamic array. Common implementations of `vector` store a pointer to a dynamically allocated array and two additional integers indicating the actual and reserved size of that array. The `pair` objects also require internal padding since a character and pointer have different sizes.

Both of these issues can be remedied by implementing a *custom* container for association lists instead of relying on the C++ library. Regardless, the need to allocate the array itself still means that this approach uses more memory compared to a linked list.

The structures shown so far describe *simple tries*. However, representing *compact tries* or a *digital search tree* can be done similarly by replacing the **bool** `is_key` field by a **const char** `*search_key` pointing to the relevant suffix of the key corresponding to the current node, or `NULL` if there is no such key [1]

Implementing a *Patricia trie* can be done by replacing the single character stored in each structure by a string. The obvious way to do this is to replacing the occurrences of **char** `c` by **const char*** `prefix`, and allocating the string dynamically.

A more efficient approach, suggested by [14], is to put an upper limit on the length of prefixes used in the Patricia trie. The advantage of this is that we can store the elements of the string directly in each node. This ensures a more optimal usage of memory. In C++, a structure implementing this could look as follows:

```
struct patricia_trie {
    patricia_trie *right;
    patricia_trie *next;
    char prefix[N];
    bool is_key;
}
```

This *restricted* form of Patricia trie obviates the need for allocating dynamically and saves storing a pointer, at the cost of requiring more nodes. However, since most prefixes in a Patricia trie will be quite small, this should be a good trade-off.

3.4.1 Optimization considerations

When using a binary tree to store the edges, an obvious optimization is to ensure that such a tree is *balanced*. However, in the light of figure 3.3, this may not gain us much beyond the first two levels of the trie.

On the other hand, an association list – whether implemented as linked list or a dynamic array – can be combined with a *move-to-front* optimization in which an element, if chosen, is moved to the front of the list having the effect of placing nodes that are more likely to be visited towards the front of the list. Alternatively, the list can be sorted beforehand on the weight of the subtries (or a similar heuristic) to achieve the same effect.

Finally, when using a *dynamic array*, it is also possible to perform a binary search instead of a linear search, if we ensure the array is sorted on character value.

3.4.2 Allocation strategies

All structures shown are recursive, and so depend on dynamic allocation to make them work. However, dynamic allocation itself incurs overhead; experiments using `gcc 4.4` on a 64-bit Linux machine shows that the amount of bytes actually needed to allocate an object of n bytes appears to be:

$$\text{allocated}(n) = 16 \cdot \max(2, \lceil (n+8)/16 \rceil)$$

This means that by allocating each structure separately, they will end up at least 32 bytes away from each other. This hurts spatial locality, and thereby the effectiveness of caching.

Also, the *physical* location that each structure takes in memory is (usually) determined by the order in which they are entered into the lexicon, not their relative position. This may mean that similar strings can end up in vastly different sections of memory, which is even more detrimental to spatial locality. This sensitivity to insertion order is demonstrated below.

We can tackle these obstacles by taking a two-step approach to handling our lexicon. Building a lexicon can be done via the obvious process of repeatedly inserting entries into it.

[1] Instead of just the suffix, we can also store the full key at each node – this wastes some memory, but can be useful. Note that small suffixes may be shared across multiple nodes and so only need to be allocated once.

Before using a lexicon, however, we should first apply *serialization* to store it in an easily machine readable form that ensures that

1. we know beforehand how may `struct`'s we should allocate, so we can do this in a single allocation, minimizing overhead and maximizing the denseness of our data structure.
2. the physical location in memory of each `struct` is determined by its logical position in the trie.

By using this approach we can achieve a dramatic improvement. The next table shows the times for loading the RIDYHEW word list in both lexicographical and randomized order, either by directly inserting each word in a simple trie, or by reading it in serialized form.

read type	load time		memory footprint
	lexicographical order	*random order*	
direct read	233ms	687ms	35103kb
using serialization	<100ms	<100ms	26327kb

3.4.3 Comparison of trie implementations

Empirical data is necessary to decide what data structure is optimal, and what representation to use. Since testing all combinations is prohibitive, we restrict ourselves to the following

- Compare linked lists, binary trees and dynamic arrays when applied to a simple trie
- Compare the various tries using the same representation of outgoing edges

In each case the test consisted of compiling the RIDYHEW list (in a randomized order), and measuring the time needed to look up 10 million samples randomly drawn from it. Each test was run a number of times, the table lists the median values. We will also note the memory requirements. All programs were written in C++, compiled using `gcc 4.4` using `-O3`. The benchmarks were performed on an Intel Xeon E5310 running Linux in 64bit mode.

Table 3.1 shows the results. The primary implementations considered were those using a plain linked list, a linked list sorted on the size of their subtries – so entries likely to be taken are nearer to the front – and a plain binary tree. For comparison, we also test an implementation in which the trees were AVL-balanced. This had no noticeable improvement.

Also compared were implementations using dynamic arrays, implemented using both a simple linear search in a `vector` sorted on the size of the subtries and a binary search in a `vector` sorted lexicographically. It is noteworthy that this last approach performed significantly worse than the others. In order to measure the overhead a standard `vector`, we also implemented a more compact representation – this had identical performance but a much reduced memory footprint.

We conclude that the simplest technique (that of a linked list with an optimized search order) already performs well, and has the smallest memory footprint. Binary trees are also a safe (and perhaps more robust) choice. Using a different representation does not seem to be beneficial.

	lookup time (ms)	in-memory size of lexicon (kb)
Linked list	2110ms	26327kb
Linked list (sorted)	1206ms	26327kb
Binary tree	1136ms	35103kb
Binary tree (AVL-balanced)	1129ms	35103kb
Vector (linear search)	1320ms	52655kb
Vector (binary search)	1995ms	52655kb
Compact array (linear)	1320ms	38394kb

Table 3.1: Comparison of simple trie implementations

We now compare different trie variations, using a linked list to store the edges between nodes as above. The results are shown in table 3.2. For each trie variant, we list the number of nodes, the time taken to look up 10 million samples (as before), and the in-memory size. For comparison, we also list some results on hash tables with k buckets (using the Fowler/Noll/Vo hash function[21]) and a basic AVL tree. While it is clear that hash tables are the fastest data structure, tries perform more than adequately and allow for a highly compact representation.

It is, however, surprising that a 'compact' trie is actually less compact than a simple trie. A likely causes of this is that the RIDYHEW lexicon does not provide enough opportunity for the tail optimization to be useful. On artificially large word lists, a compact trie shows a significant improvement in memory footprint over the simple trie.

	#nodes	lookup time (ms)	in-memory size of lexicon (kb)
Simple trie	1123327	1206ms	26327kb
Compact trie	869016	1350ms	27527kb
Digital search tree	459027	1225ms	15880kb
Unrestricted Patricia trie	572957	1512ms	19561kb
Restricted Patricia trie, $N = 4$	642035	1084ms	15047kb
Hash table $k = (2^{14})$		1598ms	25687kb
Hash table $k = (2^{16})$		933ms	27415kb
Hash table $k = (2^{18})$		677ms	33648kb
AVL tree		1290ms	35870kb

Table 3.2: Evaluation of trie variants

In conclusion, a digital search tree or restricted Patricia trie seems to be the best choice for an overall data structure, combined with linked lists or binary trees for the internal representation of edges. But a simple trie might in fact also suffice, depending on the lexicon. In any case, performance is more than adequate.

It is important to note these results apply only to the RIDYHEW list. For larger lexicons we expect – on the basis of the theoretical advantages – that a simple trie is not sufficient. For similar reasons, using linked lists seems less preferable when we have better options. But clearly, even for a large English lexicon such as RIDYHEW the results do not conform to this expectation.

3.5 Fuzzy matching in a lexicon

Having discussed only exact matching so far, we now turn to a core research question. A naïve solution to perform fuzzy matching in a *simple trie* is exemplified by the fragment in figure 3.4. This function takes the directly recursive approach — similar to the code in figure 2.1 — to test whether the trie contains a key matching the string within Levenshtein distance d.

As before, this solution is not very efficient, and potentially visits each node in the trie many times. An attempt at modifying this function to return more information than a simple boolean answer will at best result in a function only usable in very small toy examples.

3.5.1 Using finite state machines

A more promising, and in fact much easier technique is to use the finite state machine of section 2.4.1; if we assume that we have access to these via the functions `nfaSetup`, `nfaFeed` and `nfaAccepts`, a Haskell implementation could be as simple as:

```haskell
fuzzy_match' d string trie = match (nfaSetup d string) trie
  where
    match state root@(Trie subtries isKey)
      = nfaAccepts state && isKey
          ||
        or [match (nfaFeed c state) trie | (c,trie) <- subtries]
```

In a finite state machine described in section 2.4.1, each application of `nfaFeed` is an $O(k)$ operation. It is easily seen that the function visits each node in the trie only once. Since the number of nodes in a simple trie containing N words of maximal length m is $O(m \cdot N)$, this function has the same worst-case complexity as a brute-force search in a simple list of words.

3.5.2 Adapting automata for best-first search

As discussed in section 2.4.1, it is easy to reason about the state of a finite state machine in the `agrep` approach. In particular, we can easily see what the optimal edit distance is at which an automaton may still produce a match by determining which rows of the automaton still contain active states.

And since all transitions in the automaton at best keep the edit distance the same, this means that we can use this measure to construct a heuristic function $\hat{f}$ operating on the state $s = \{w_i\}_{0 \leq i \leq k}$ of the automaton:

$$\hat{f}(s) = \min_i \{w_i \neq 0 : w_i \in s\}$$

This function is monotonic, so can be used to construct a best-first search by setting $f(s) = \hat{f}(s)$ if the automaton is not in any accepting state and defining $f(s)$ to be the edit distance matched if it is. Since these values can be computed easily, we can use it to compute a best-only result as described in section 2.6.2.

A downside is that we have no efficient automaton that computes a generalized edit distance.

3.5.3 Adding memoization to the naïve solution

Another approach is to improve the naive solution shown in figure 3.4 using a best-first search with an incremental heuristic as described in section 2.6.3. In this case the search space $H = \mathcal{N} \times \{i \in \mathbb{N} : i \leq |S|\}$ is the product of the set of nodes in the trie and the set of valid indices into the string S that we want to match. The state (ν, i) captures that we have reached ν in the trie and have discarded the first i characters of S.

```haskell
fuzzy_match :: Int -> String -> Trie -> Bool
fuzzy_match d _ _ | d < 0
 = False

fuzzy_match d []      root@(Trie subtries isKey)
 = isKey || or [fuzzy_match (d-1) [] trie | (_,trie) <- subtries]

fuzzy_match d (c:cs) root@(Trie subtries isKey)
 = case lookup c subtries of
        Just trie -> fuzzy_match d cs trie
        Nothing   -> False
      || ed cs root
      || or [ed cs trie     | (_,trie) <- subtries]
      || or [ed (c:cs) trie | (_,trie) <- subtries]
 where ed = fuzzy_match (d-1)
```

Figure 3.4: Naïve solution of fuzzy matching in Haskell

Our set of goal states is $T = \{(\nu, |S|) : \nu \in \mathcal{N},\ \nu \text{ represents a stored key}\}$. The state transitions correspond to the edit operations, as before. That is, given a state (ν, i), the set of next states is $\{(\nu, i+1)\} \cup \{(\mu, i) : \mu \text{ is a child of } \nu\} \cup \{(\mu, i+1) : \mu \text{ is a child of } \nu\}$. which correspond to deleting the i'the character in S, inserting a character from the trie before the i'the character in S, and matching/replacing the i'the character of S with a character in the trie.

The cost function $\mathcal{V}((\nu, x) \to (\mu, y))$ then simply corresponds to the cost assigned to these edit operations, which is an easy function to compute. So we have all the required tools to construct a best-first search in the trie.

The complexity of this search can easily be seen to be the limited by the size of the search space. That is, if we have a trie containing N words of maximal length m, matching a string S of length n has worst-case complexity $O(n \cdot mN)$, which is the same as that of a brute-force search implemented using the dynamic programming algorithm of figure 2.3. However, we should expect to visit each node much less than m times, and we expect the average case complexity to be closer to $O(n+mN)$. Compare this with $O(n+kmN)$ for the approach of the previous section. We also need to maintain a negative memo-table, which in the worst case may consist of $O(mN)$ entries, but usually much less.

Note that the idea of using best-first search in a trie to solve edit distances is not new; an early mention of it is found in [7].

3.5.4 Comparison of techniques

We implement both techniques in C++ using a simple trie. We chose this data structure for three reasons:

- We have seen that is is reasonably efficient, and very compact.
- The choice of structure should not have as large an impact as the size of the lexicon itself.
- Implementing a best-first search is easiest on a simple trie.

We then perform a *best-only* and an *unambiguous best-only* search. To control this search we use a priority queue consisting of $k+1$ buckets (each implemented using C++ `vector`) as our open list, one for each possible edit distance. In this way we can select the next best possible continuation and add continuations to the open list in (amortized) constant time. The closed list can either be represented by a `set` or `unordered_set`; this choice did not seem important.

When performing a best-only search using automata, we largely follow the best-first search strategy, but greedily open a node μ if the cost of the transition from node ν to μ in the trie did not increase the heuristic value of $\hat{f}$. This should prevent opening too many states that are irrelevant. Also, using automata we do not need to explicitly maintain the closed set, as we are guaranteed to visit every node just once.

search strategy	max. distance	Levenshtein automaton	Best-first
best-only	$d = 1$	1.8s	5s
unambiguous best-only	$d = 1$	1.3s	4s
best-only	$d = 2$	11.5s	53s
unambiguous best-only	$d = 2$	6.4s	35s
best-only	$d = 4$	66s	8m
unambiguous best-only	$d = 4$	26s	264s

Table 3.3: Time taken to correct Norvig's data set using an unweighted Levenshtein distance

To measure performance, we use the RIDYHEW word list and the list of misspellings obtained from [22], used in section 4.2.1. During this test, we look up around 47000 words (most of which with small or medium errors) using best-only and unambiguous best-only search using a unmodified Levenshtein distance, for a maximum edit distance $d \in \{1, 2, 4\}$.

The results are shown in table 3.3. In a direct comparison, it is clear that the best method to perform fuzzy matching using a standard Levenshtein distance is a best-first search combined with Levenshtein automata.

However, as we will see in the next section, the usefulness of Levenshtein distance with $d > 1$ is questionable. Also, by modifying the weights of each edit operation the search space will change as well – many obscure parts of the lexicon will be pruned even before the search has begun.

3.6 Summary

In this chapter we have investigated building an actual implementation of fuzzy lexical matching in C++. For use as a data structure, either a *digital search tree* or *restricted Patricia trie* using an association list or binary tree to store pointers to child nodes seems the best option.

Fuzzy matching using a unweighted edit distance can be implemented using Levenshtein automata and an informed search strategy. However, these become unwieldy for a weighted edit distance. Applying a *best-first search* directly also results in solution that — although slower in the case of a unweighted edit distance — is still viable.

4 Effectivity of fuzzy matching

In this chapter we will try to determine whether an implementation of a weighted edit distance based on a confusion matrix is an improvement over an unweighted Levenshtein distance. A first impression is that this should be the case. Many spelling errors consist of a single edit operation. Secondly, a majority of errors are of the same type.

In the sets of corrupted spellings examined in this chapter, we have found that common errors involve the vowels *e*, *i*, *a*, which often get inserted, deleted or replaced with each other. Errors involving consonants are usually simple insertions and deletions, and seem focused around the letters *r,l,n,s* and *t*. If we construct a set of weights for an edit distance that takes advantage of these facts, it is reasonable to expect some improvement in our ability to detect and correct spelling errors.

4.1 Framework for evaluation

An often-used measure of effectivity for spelling correction is the accuracy: how many incorrect words were detected and corrected successfully? However, a richer picture can be obtained if we treat a spelling corrector as a binary classifier. A framework for this is presented in [25].

This framework operates by partitioning a test set into two groups: words that we want to see corrected (the *target* group) and words that should be left as-is. Secondly, the subset of words that are being acted upon by an automatic correction mechanism are considered to be *selected*.

The division between *target* and *non-target* is not necessarily very sharp. For example, if an out-of-lexicon word becomes corrupted, we have no hope of correcting it using a lexicon-based correction system information. In fact, there is a significant risk of 'correcting' uncorrupted out-of-lexicon words.

Given the scope of our implementation, we will consider such cases to be *non-target*. That is, we want out-of-lexicon words to be left as-is.

Looking at the intersections of the *(non-)target* and *(non-)selected* groups, we get four familiar classes:

true positives (TP) *target* and *selected*; i.e. words that contained errors that have been corrected successfully

true negatives (TN) *non-target* and *not selected*: words correctly kept as-is. These are primarily uncorrupted words we can find by an exact match, but also – as outlined above - out-of-lexicon words that successfully evaded correction

false positives (FP) *non-target* and *selected*. These are the cases in which a system erroneously 'corrects' a word. By design the only case in this class are out-of-lexicon words that our system corrects to an in-lexicon word ('false friends')

false negatives (FN) *target* and *not selected*; this is the remaining class of in-lexicon words that were corrupted and either not corrected, or corrected to the wrong word

Using these definitions, the *recall* $R = TP/(TP + FN)$ is the ratio of errors caught; the *precision* $P = TP/(TP + FP)$ is the ratio of *selected* words where we have successfully performed a correction. These two can be combined into the *F measure* by taking the harmonic mean: $F = 2RP/(R + P)$.

However, we will also have to be more precise about what it means for a word to be in the *selected* set. This too is not a sharp definition. For example: if we attempt to correct the word 'entorpy';

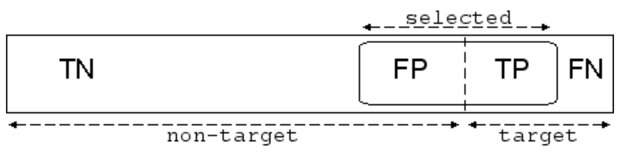


Figure 4.1: Schematic representation of spelling correction task (due to [25]

should it count as a True Positive if we are presented with a list of 100 possible corrections at the same edit distance, of which 'entropy' is but one? If we are simply interested in the capacity to retrieve the correct form, it should; if on the other hand we are interested in the capacity of our system to unambiguously select a correction, this is a False Negative. The framework of [25] deals with this by classifying correction tasks into levels; each building upon the other, in the sense that higher levels can at best perform as well as lower levels. We will follow this general idea; and so the definition of the *selected* set will vary between tests.

4.2 Obtaining actual confusion data

Until now, we have not bothered with actually deriving a confusion matrix needed for a weighted edit distance. This turns out to be rather problematic; we failed to find any ready-to-use matrices in the public domain. One suggested approach is to use an iterative process[12]: start with a regular edit distance, run a spell checker, update the confusion table and repeat. Another involves building a stochastic model[27]; but deriving edit costs again is a iterative process. Both approaches are rather involved, and tend to focus more on assigning a probability to word-correction pairs, instead of constructing a simple edit distance. For example, in the approach taking by [27], no two strings have an edit distance of 0.

A more straight-forward approach is taken by [1]: letting $C(i, j)$ denote the relative frequency with which the letter i should get substituted by j, they derive the associated substitution cost as $S(i, j) = \log \frac{C(i,i)}{C(i,j)}$ (for $i \neq j$).

We will take an even simpler view. We observe that the most frequently needed correction step should have the smallest edit distance (1), and all other edit operations should get scores relative to that operation. Given a list of simple edit operations O, let N_{op} denote the number of times that the operation $op \in O$ was needed to correct an error. Then we can derive the penalty for op as

$$P(op) \overset{def}{=} 1 + \log \frac{\max_{op' \in O} N_{op'}}{N_{op}} \tag{4.1}$$

As an exception, substituting a character for itself is not considered an edit operation, and so the 'penalty' for substituting a character by itself is fixed at $P(c \to c) = 0$.

The rationale for this formula is mainly that it is simple. But, we can also see that $\max_{op'} N_{op'}/N_{op}$ is proportional to the expected number of edit operations we need to perform to see a single occurrence of op. Adding the logarithms of these is, of course, related to multiplying these expected values. In other words, defining the penalties in this way is not entirely arbitrary.

Using definition 4.1, we have reduced the problem to simply finding a frequency count of needed corrections, given a set of correct words and misspellings.

4.2.1 Chosen data sets

As test material, we have used two different sources. The first – which we will call Norvig's data set – is taken from [22]. This provides us with a large list of words with associated misspellings, collected from various sources (amongst others, misspellings found on Wikipedia). The same source also provides us with counts of spelling correction edits, including one character of pre-context in the case of insertions and deletions. From this list of correct words, we have filtered out the ones in which the correct word contained a contraction (*hasn't*, *that's*) or was a possessive form – since we feel most of these should not be handled at the vocabulary level.

A drawback of this list of misspellings is that it contains many of far-fetched misspellings – for example, *suxsefel* for *successful*, or *mutril* for *mutual*, and it treats every misspelling as equally common.

Therefore, we also have used the EPO1A corpus[15] — consisting of abstracts of patent applications — to generate a list of word corrections ourselves. We have done this by (using a large English vocabulary) selecting all out-of-vocabulary lower-case words which occur only once (*hapax legomena*) or twice (*dis legomena*), as candidate-misspellings. This produced 1873 *hapax*, out of which we identified 544 as misspellings, and 417 *dis legomena*, out of which 40 could be identified as corrupted. These word lists were small enough to correct by hand, giving us some measure on the occurrence of edit corrections needed in a real-world texts. Like [6], we found that the vast majority of errors found consisted of a wrong, missing or inserted letter, or transposition of two adjacent letters. In some cases, a bigram had to removed; the most common example of this is a superfluous *-ly* suffix in words such as *stepwisely* or *edgewisely*.

As in Norvig's data set, we find that most corruptions fall into a small group of single letter operations; figure 4.2 lists the frequency of each simple edit operation necessary to correct a misspelt word. We also see that the 16 most common corrections are all insertions and deletions, and that these account for about half of all errors – clearly, adding a contextual information in these cases will help to make a weighted edit distance more discerning.

A similar table listing the frequency of single letter corrections, also taking into account one single character of pre-context, and allowing for swaps of adjacent characters is listed in the appendix.

36 ∅→*i*	9 *e*→*i*	4 ∅→*h*	2 *s*→*c*	2 ∅→*x*	1 *r*→*t*	1 *l*→*i*	1 *e*→*b*
32 ∅→*e*	9 ∅→*m*	3 *p*→∅	2 *r*→*y*	2 ∅→*f*	1 *r*→*p*	1 *l*→*b*	1 *d*→*n*
30 ∅→*r*	8 *m*→*n*	3 *n*→*a*	2 *r*→*o*	1 *y*→*i*	1 *r*→*m*	1 *j*→*h*	1 *d*→*j*
30 ∅→*l*	8 ∅→*d*	3 *l*→*a*	2 *r*→*n*	1 *y*→*e*	1 *r*→*l*	1 *j*→*d*	1 *c*→*t*
22 *i*→∅	6 *t*→*i*	3 *i*→*u*	2 *p*→*e*	1 *y*→*a*	1 *r*→*i*	1 *j*→*a*	1 *c*→*s*
20 *n*→∅	6 *o*→*e*	3 *i*→*c*	2 *o*→*u*	1 *v*→*i*	1 *r*→*c*	1 *i*→*v*	1 *c*→*r*
20 *l*→∅	6 *i*→*t*	3 *e*→*u*	2 *o*→*s*	1 *v*→*f*	1 *r*→*a*	1 *i*→*r*	1 *c*→*n*
19 *e*→∅	6 *e*→*o*	3 *e*→*r*	2 *o*→*r*	1 *u*→*r*	1 *p*→*t*	1 *i*→*n*	1 *c*→*d*
19 ∅→*s*	6 *a*→*i*	3 *d*→∅	2 *o*→*m*	1 *u*→*l*	1 *p*→*r*	1 *i*→*f*	1 *b*→*p*
18 ∅→*t*	5 *u*→∅	3 *c*→*i*	2 *o*→*l*	1 *u*→*a*	1 *p*→*o*	1 *h*→*t*	1 *b*→*l*
14 ∅→*a*	5 *c*→∅	3 *a*→*n*	2 *o*→*a*	1 *t*→*s*	1 *p*→*a*	1 *h*→*e*	1 *b*→*e*
13 *s*→∅	5 ∅→*p*	3 *a*→*l*	2 *o*→∅	1 *t*→*r*	1 *o*→*t*	1 *h*→∅	1 *a*→*y*
12 *r*→∅	5 ∅→*g*	2 *y*→*r*	2 *m*→*o*	1 *t*→*p*	1 *o*→*p*	1 *g*→*t*	1 *a*→*u*
12 *a*→∅	4 *u*→*i*	2 *x*→*s*	2 *l*→*r*	1 *t*→*o*	1 *n*→*i*	1 *g*→*d*	1 *a*→*r*
12 ∅→*n*	4 *r*→*e*	2 *u*→*t*	2 *l*→*o*	1 *t*→*h*	1 *n*→*e*	1 *f*→*v*	1 *a*→*p*
11 ∅→*c*	4 *n*→*m*	2 *u*→*s*	2 *l*→*f*	1 *t*→*b*	1 *n*→*d*	1 *f*→*l*	1 *a*→*m*
10 *e*→*a*	4 *n*→*g*	2 *u*→*o*	2 *g*→*q*	1 *s*→*z*	1 *m*→*r*	1 *f*→*i*	1 *a*→*j*
10 *a*→*e*	4 *m*→∅	2 *u*→*e*	2 *f*→∅	1 *s*→*l*	1 *m*→*e*	1 *e*→*s*	1 ∅→*z*
9 *t*→∅	4 *g*→*n*	2 *t*→*u*	2 *e*→*p*	1 *s*→*e*	1 *m*→*a*	1 *e*→*n*	
9 *i*→*e*	4 ∅→*u*	2 *s*→*u*	2 *b*→∅	1 *s*→*d*	1 *l*→*u*	1 *e*→*m*	
9 *i*→*a*	4 ∅→*o*	2 *s*→*o*	2 ∅→*y*	1 *r*→*u*	1 *l*→*s*	1 *e*→*h*	

Figure 4.2: Frequencies of single letter corruptions — without pre-context — in the *hapax* and *dis legomena* of the EPO1A corpus.

4.3 Experiments performed

Since the starting point for our implementation was an unweighted edit distance, it is only appropriate to also use this as a baseline against which we want to see an improvement. We will use both the standard Levenshtein distance[17] as well as the Damerau-Levenshtein distance[6] which also allows for transpositions of two *adjacent* characters.

As our attempted improvement, we will consider a weighted edit distance which lists a cost for substituting, inserting and deleting a single character (i.e. which can be represented using a single $|\Sigma| \times |\Sigma|$ matrix), as well as a weighted edit distance which also uses one character of context, as described in section 2.5.

These four systems will be tested in a number of settings. In each setting we will supply a lexicon, a list of correct words and corresponding misspellings, and for each word type (correct and corrupted) perform a best-only search in our lexicon. Thus, only words found at the minimum distance count as a correction candidate.

Basic correction capability Given a lexicon consisting *only* of the correct word forms, measure Recall. At this level, a word is *selected* if it is in the list of candidates. This is the most favorable condition – ideally, we should have a Recall of 1. Precision will be 1 by design.

Basic error detection Use RIDYHEW as a lexicon, which has a comprehensive but incomplete coverage, measure Recall and Precision. *Selected* words are as in the previous test. The effect of a wider lexicon should provide more background 'noise'.

Frequency-based correction As in the previous case, but from the list of correction candidates pick the most frequently occurring word (according to an externally provided frequency count).

Unambiguous correction As in the second case, but only consider a word *selected* if it is an unambiguous match – i.e. no alternative matches at the same edit distance exist. We can use the *unambiguous best-only* search of section 2.6.2 to do this more efficiently than the frequency-based selection.

There is an increasing level of difficulty in these tests; for example, if a word fails to be corrected during frequency-based correction, it will obviously also fail to be an unambiguous match. Note that only the last two tests actually perform a *one-to-one* correction of words.

We also need to measure how robust our correction mechanism is when encountering correct out-of-lexicon words. To do this we use a probabilistic test:

Robustness Split a word list randomly into a lexicon (97%) and a test-set (3%); attempt to look up the test-set in the lexicon and measure the False Positive Rate, $FP/(FP + TN)$.

Since none of the words of the test-set will be in the lexicon, Recall will obviously be 0. Note that we can measure the False Positive Rate in two ways: either as we would in the Frequency-based correction task (in which case any correction candidate immediately results in a False Positive), or as in the Unambiguous correction-task (in which case we are still okay if we find more than one correction candidate).

By repeating this test a number of times and averaging the result, we will get an indication about how robust a system is against so-called *false friends*.

4.4 Results

Norvig's data set First, we present the results of the aforementioned tests applied to the first list of misspellings, using the associated confusion data. The maximum edit distance for correction candidates in all cases was set to the arbitrary large value of 64 to give an upper bound on recall.

task	measure	TP	TN	FP	FN	Recall	Precision
Base correction	Levenshtein	24563	7754	0	14754	0.625	1
Error detection	Levenshtein	16515	7270	1346	21940	0.429	0.925
Frequency-based correction	Levenshtein	13501	7234	1507	24829	0.352	0.9
Unambiguous correction	Levenshtein	6744	8083	533	31711	0.175	0.927
Base correction	Damerau-Levenshtein	24982	7754	0	14335	**0.635**	1
Error detection	Damerau-Levenshtein	17188	7270	1346	21267	0.447	0.927
Frequency-based correction	Damerau-Levenshtein	14117	7234	1507	24213	0.368	0.904
Unambiguous correction	Damerau-Levenshtein	7073	8080	536	31382	0.184	0.93
Base correction	Weighted	22693	7754	0	16624	0.577	1
Error detection	Weighted	14052	7270	1346	24403	0.365	0.913
Frequency-based correction	Weighted	13489	7234	1507	24841	0.352	0.9
Unambiguous correction	Weighted	10343	7615	1001	28112	0.269	0.912
Base correction	Weighted w. context	23284	7754	0	16033	0.592	1
Error detection	Weighted w. context	13730	7270	1346	24725	0.357	0.911
Frequency-based correction	Weighted w. context	13405	7234	1507	24925	0.35	0.899
Unambiguous correction	Weighted w. context	11037	7557	1059	27418	0.287	0.912

EPO1A data Next, we perform the same tests on the misspellings obtained from the EPO1A corpus, and the associated confusion data.

task	measure	TP	TN	FP	FN	Recall	Precision
Base correction	Levenshtein	592	533	0	0	1	1
Error detection	Levenshtein	530	519	29	47	0.919	0.948
Frequency-based correction	Levenshtein	449	501	65	110	0.803	0.874
Unambiguous correction	Levenshtein	285	536	12	292	0.494	0.96
Base correction	Damerau-Levenshtein	592	533	0	0	1	1
Error detection	Damerau-Levenshtein	565	519	29	12	0.979	0.951
Frequency-based correction	Damerau-Levenshtein	485	501	65	74	0.868	0.882
Unambiguous correction	Damerau-Levenshtein	317	536	12	260	0.549	0.964
Base correction	Weighted	592	533	0	0	1	1
Error detection	Weighted	506	519	29	71	0.877	0.946
Frequency-based correction	Weighted	480	501	65	79	0.859	0.881
Unambiguous correction	Weighted	416	527	21	161	0.721	0.952
Base correction	Weighted w. context	590	533	0	2	0.997	1
Error detection	Weighted w. context	503	519	29	74	0.872	0.945
Frequency-based correction	Weighted w. context	479	501	65	80	0.857	0.881
Unambiguous correction	Weighted w. context	450	526	22	127	0.78	0.953

It is immediately clear that the EPO1A set is better suited to edit-distance based word correction: this much is clear from the fact that the correct word is practically always amongst the correction candidates found using a best-only search in three out of four cases.

The Recall value decreases with each task, as should be expected – if a word is a true positive in the *Unambiguous correction* task, then it will also be selected by the *Frequency-based* task (since it is the only result at that edit distance), and the *Error detection* task. Finally, the lexicon used in the

Error detection task has less coverage than the one used in the *Base correction* task, and therefore more opportunities for false negatives.

It should also be noted that the *Base correction* recall of the Levenshtein distance provides an upper bound for the weighted edit distance, and similarly the Damerau-Levenshtein distance is an upper bound for the weighted edit distance with context. This may be surprising, but is also to be expected: the average cost of edit operations goes up, yet the maximum edit distance stays the same. From this it immediately follows that the maximally obtainable recall is quite low (0.635) in the case of Norvig's data set.

Note that the precision values stay consistently high — this is primarily because we are using lexicons with high coverage. The primary thing to note in this column is that the score for *Frequency-based* correction is a first indication that it is more susceptible to false friends.

The false friends problem Given enough edit operations, we can transform every word into every other word. To prevent this we need to find an acceptable maximum allowed edit distance. We do this by repeating the above test using a maximum edit distance d, determining both the reduction in Recall, and the 'False friends ratio', defined as the False Positive Rate in the *Robustness test.* We also compare unambiguous best-only correction with frequency-based correction. These tests have only be run on the EPO1A data set.

	Unambiguous		Frequency-based	
task	Recall	False friends	Recall	False friends
Levenshtein $d = 1$	0.45	0.10	**0.70**	**0.16**
Levenshtein $d = 2$	0.49	0.20	0.80	0.48
Levenshtein $d = 4$	0.49	0.33	0.80	0.89
Damerau-Levenshtein $d = 1$	0.53	0.10	**0.83**	**0.17**
Damerau-Levenshtein $d = 2$	0.54	0.21	0.86	0.49
Damerau-Levenshtein $d = 4$	0.54	0.33	0.87	0.88
Weighted $d = 2$	0.59	0.07	0.69	0.08
Weighted $d = 4$	**0.71**	**0.22**	0.84	0.31
Weighted $d = 8$	0.72	0.49	0.86	0.72
Weighted w. context $d = 2$	0.18	0.01	0.18	0.01
Weighted w. context $d = 4$	**0.64**	**0.06**	0.69	0.06
Weighted w. context $d = 8$	0.69	0.18	0.75	0.20

Using an unweighted edit distance, increasing the maximum edit distance beyond 1 gives a negligible improvement on recall, but increases the number of false friends dramatically. In these cases, using $d = 1$ and a frequency-based selection seems the best choice.

A good value of d for weighted edit distances seems to be $d = 4$; as there is little improvement in Recall but a rise in false friends after that. More importantly, the unambiguous best-only correction method performs nearly as well as the frequency-based selection on this test. This is helpful, since we can then use the faster *unambiguous best-only search.*

We will now repeat the benchmarks measuring performance on Norvig's data set (seen earlier in table 3.3) to these three methods. To compute a Damerau-Levenshtein distance we use the finite state automaton approach described before; to compute weighted edit distances we use a best-first search and memoization, as described in chapter 3.

correction task		time needed
(Frequency based) Damerau-Levenshtein	$d = 1$	2.5s
(Unambiguous) Weighted	$d = 4$	25s
(Unambiguous) Weighted w. context	$d = 4$	4.4s

In conclusion, we find that a weighted edit distance *with context* can be very effective at correction tasks. It produces fewer false friends — and can have good performance. On the other hand, there seems to be little point in using a weighted edit distance *without context.*

5 Conclusions

We have outlined a method for performing fuzzy matching against a lexicon using a standard edit distance and a form of weighted edit distance. By implementing these in the course of this research, we have determined answers to the questions posed in our original problem statement:

Which algorithms and data structures can we use? Variations on tries, especially *digital search trees* and *restricted Patricia tries* are useful data structures. Levenshtein automata implemented using bit-parallelism are an efficient way to compute an unweighted edit distance. Dynamic programming and best-first search are possible techniques to implement a weighted edit distance.

Can we efficiently implement these? Trie structures allow for a very compact representation. A Levenshtein automaton can be applied to such a trie structure using an informed search strategy; we have found a combination of best-first and greedy search to be the most efficient. Using best-first search to compute a weighted edit distance in a trie directly is also feasible.

How should we modify the edit distance measure? The approach taken in this thesis is to assign weights to edit operations. Because insertions and deletions are the most common operations, the weight of these operations must also depend on contextual information.

How do we determine that, so doing, we have improved word correction? Using the evaluation framework provided by [25], we can meaningfully use the notions of Recall, Precision and False Positive Rate to evaluate any spelling correction system thoroughly.

Can we use our implementation outside an experimental setup? Yes, if we restrict ourselves to a unweighted edit distance. We are, however, not yet satisfied that the benefits of a weighted edit distance outweigh the costs of implementing it in a practical setting.

While we have shown that a weighted edit distance can perform better than a standard edit distance, this does come at a price – finding a useful list of spelling errors is hard. Constructing these data sets by hand is time consuming — processing a data set to obtain the *hapax legomena* can be automated, but each spelling error has to be detected and corrected by hand. In some cases the intended word is unclear, or requires contextual information. It is also very tedious work — it took 8 hours to construct a list of misspellings and corrections for the EPO1A corpus.

Yet, having a good data set is essential for construction a reliable confusion matrix. During construction, we have observed that a good confusion matrix serves two purposes – first, it improves the recall and precision of spelling correction. But it also helps *shape* the search space so that a best-first search can be run at an adequate speed. A bad confusion matrix therefore means we will have to spend more time waiting for worse results.

Because of this, we consider this implementation of weighted edit distances promising, but still only useful in an experimental setup.

5.1 Further challenges

During this research, many problems were encountered that we did not address, but which are important to fuzzy lexical matching.

- Our construction of confusion matrices was largely based on intuition, and only intended to demonstrate the feasibility of a weighted edit distance. Also, we did not split our data sets into a training and test set, which implicitly means our experiments suffer from overfitting. To construct better matrices, a more rigorous analysis than performed in this thesis is called for.
- The start of the art seems to be moving towards a more general $n \rightarrow n$ substitution model of edit distance[4], which can also model common mistakes such as $ph \rightarrow f$, OCR errors where two letters are merged into one, or perhaps use even more context to help correct specific cases such as recognizing the suffix *-ize* whenever the lexicon considers *-ise* correct.
- OCR correction usually produces word boundary errors as well, where spaces get introduced in between words or are dropped. This adds an extra challenge of finding a correct splitting into words of an entire sentence. Combining this with fuzzy matching might be possible, but a statistical approach may also be necessary[22].
- In some cases, a word is obviously misspelt (such as the word *froward* whenever *forward* is intended), but cannot be corrected by a purely lexical system since the corrupted word is also a valid word – but does not make sense in the context where it occurs.
- Instead of a fixed list of words, we might want to use an open lexicon. For example, one which can recognize any arbitrary chemical formula, or valid postal codes. This at the very least requires extensions to the trie used to implement the lexicon – and will probably involve modelling it as a very large deterministic finite state automaton.

Frequency of corrections in EPO1A

This appendix lists the frequency with which single edit operations were necessary to correct misspelt words found in the EPO1A corpus as described in 4.2.1. This list is similar to the one in figure 4.2, but counts transpositions separately and includes one character of pre-context in the case of insertions and deletions. No insertions or deletions were ever required at the start of a word.

10 *ll*→*l*	3 *ur*→*u*	2 *rn*→*r*	2 *e*→*ec*	1 *u*→*i*	1 *os*→*so*	1 *i*→*io*	1 *e*→*eg*
10 *l*→*ll*	3 *si*→*s*	2 *rl*→*r*	2 *de*→*d*	1 *ts*→*t*	1 *or*→*ro*	1 *i*→*id*	1 *e*→*ee*
10 *e*→*a*	3 *s*→*st*	2 *re*→*r*	2 *dd*→*d*	1 *tp*→*pt*	1 *op*→*po*	1 *ht*→*th*	1 *e*→*ea*
10 *a*→*e*	3 *s*→*se*	2 *re*→*er*	2 *d*→*de*	1 *te*→*t*	1 *oo*→*o*	1 *hs*→*h*	1 *ds*→*d*
8 *u*→*ur*	3 *rr*→*r*	2 *r*→*n*	2 *d*→*dd*	1 *ta*→*t*	1 *oi*→*o*	1 *hh*→*h*	1 *dn*→*nd*
8 *m*→*n*	3 *ri*→*r*	2 *pr*→*p*	2 *ci*→*c*	1 *t*→*to*	1 *oc*→*o*	1 *he*→*eh*	1 *d*→*dl*
8 *m*→*mm*	3 *r*→*ra*	2 *pl*→*p*	2 *bb*→*b*	1 *t*→*tl*	1 *o*→*ou*	1 *ha*→*h*	1 *ct*→*c*
8 *i*→*a*	3 *p*→*pl*	2 *p*→*pt*	2 *b*→*bl*	1 *t*→*ta*	1 *o*→*ot*	1 *h*→*ht*	1 *cs*→*sc*
6 *r*→*re*	3 *ng*→*gn*	2 *p*→*pi*	2 *a*→*au*	1 *t*→*s*	1 *o*→*on*	1 *h*→*hi*	1 *cr*→*rc*
6 *o*→*e*	3 *na*→*n*	2 *ou*→*o*	2 *a*→*ar*	1 *t*→*b*	1 *o*→*oe*	1 *h*→*ha*	1 *ci*→*ic*
6 *n*→*ne*	3 *n*→*ni*	2 *on*→*o*	1 *zi*→*z*	1 *st*→*s*	1 *nt*→*n*	1 *gu*→*g*	1 *c*→*t*
6 *in*→*i*	3 *li*→*l*	2 *ol*→*o*	1 *z*→*zi*	1 *so*→*os*	1 *ne*→*en*	1 *gn*→*ng*	1 *c*→*n*
6 *e*→*o*	3 *l*→*li*	2 *ol*→*lo*	1 *yp*→*y*	1 *s*→*z*	1 *nc*→*n*	1 *gn*→*g*	1 *c*→*d*
5 *t*→*tr*	3 *iu*→*ui*	2 *o*→*os*	1 *ya*→*ay*	1 *s*→*so*	1 *n*→*nn*	1 *ge*→*g*	1 *c*→*ct*
5 *ss*→*s*	3 *it*→*i*	2 *o*→*oi*	1 *y*→*yl*	1 *s*→*sl*	1 *n*→*na*	1 *g*→*t*	1 *c*→*ce*
5 *s*→*ss*	3 *ii*→*i*	2 *o*→*a*	1 *y*→*yi*	1 *s*→*d*	1 *mr*→*rm*	1 *g*→*gr*	1 *c*→*cc*
5 *s*→*si*	3 *i*→*in*	2 *na*→*an*	1 *y*→*yd*	1 *s*→*c*	1 *mp*→*m*	1 *g*→*gn*	1 *c*→*ca*
5 *r*→*rr*	3 *i*→*e*	2 *n*→*ns*	1 *y*→*i*	1 *rt*→*tr*	1 *mi*→*m*	1 *g*→*gi*	1 *b*→*p*
5 *nn*→*n*	3 *g*→*ge*	2 *n*→*nl*	1 *y*→*e*	1 *ro*→*or*	1 *ma*→*am*	1 *g*→*ga*	1 *at*→*a*
5 *i*→*ic*	3 *en*→*e*	2 *n*→*nd*	1 *xt*→*x*	1 *ri*→*ir*	1 *m*→*mp*	1 *g*→*d*	1 *as*→*a*
5 *ee*→*e*	3 *e*→*i*	2 *mo*→*om*	1 *xc*→*x*	1 *ra*→*ar*	1 *m*→*mo*	1 *f*→*v*	1 *an*→*na*
5 *b*→*bi*	3 *c*→*ci*	2 *me*→*m*	1 *x*→*xc*	1 *r*→*ry*	1 *m*→*me*	1 *f*→*fy*	1 *an*→*a*
5 *a*→*i*	3 *al*→*la*	2 *la*→*l*	1 *wi*→*w*	1 *r*→*rs*	1 *m*→*ma*	1 *f*→*fl*	1 *aj*→*ja*
4 *ti*→*it*	2 *yr*→*ry*	2 *l*→*r*	1 *vl*→*v*	1 *r*→*rh*	1 *lt*→*l*	1 *eu*→*ue*	1 *ai*→*ia*
4 *t*→*ti*	2 *y*→*yn*	2 *l*→*le*	1 *vc*→*v*	1 *r*→*rd*	1 *ls*→*sl*	1 *es*→*se*	1 *ai*→*a*
4 *t*→*te*	2 *y*→*yc*	2 *it*→*ti*	1 *v*→*ve*	1 *r*→*l*	1 *lf*→*fl*	1 *es*→*e*	1 *a*→*ax*
4 *ra*→*r*	2 *x*→*s*	2 *is*→*i*	1 *v*→*f*	1 *r*→*e*	1 *le*→*l*	1 *er*→*re*	1 *a*→*at*
4 *p*→*pp*	2 *v*→*vi*	2 *il*→*i*	1 *uu*→*u*	1 *q*→*qu*	1 *lb*→*bl*	1 *em*→*me*	1 *a*→*an*
4 *o*→*or*	2 *ut*→*tu*	2 *ie*→*ei*	1 *us*→*u*	1 *pr*→*rp*	1 *l*→*i*	1 *ei*→*e*	1 *a*→*al*
4 *o*→*ol*	2 *u*→*ut*	2 *ic*→*ci*	1 *ur*→*ru*	1 *pp*→*p*	1 *l*→*f*	1 *ed*→*e*	1 *a*→*ai*
4 *n*→*nt*	2 *u*→*ul*	2 *i*→*ia*	1 *uo*→*ou*	1 *po*→*p*	1 *jd*→*dj*	1 *ec*→*e*	1 *a*→*ad*
4 *n*→*ng*	2 *tr*→*t*	2 *he*→*h*	1 *ul*→*u*	1 *pa*→*ap*	1 *j*→*h*	1 *eb*→*be*	
4 *n*→*m*	2 *ti*→*t*	2 *g*→*q*	1 *ul*→*lu*	1 *p*→*pr*	1 *iv*→*vi*	1 *ea*→*e*	
4 *mm*→*m*	2 *t*→*tt*	2 *ff*→*f*	1 *ue*→*eu*	1 *p*→*ph*	1 *iu*→*i*	1 *e*→*u*	
4 *i*→*is*	2 *t*→*th*	2 *f*→*fi*	1 *ua*→*au*	1 *p*→*pe*	1 *in*→*ni*	1 *e*→*ex*	
4 *ei*→*ie*	2 *su*→*us*	2 *er*→*e*	1 *u*→*us*	1 *p*→*pa*	1 *if*→*fi*	1 *e*→*et*	
4 *e*→*es*	2 *se*→*s*	2 *ep*→*pe*	1 *u*→*un*	1 *ou*→*uo*	1 *ie*→*i*	1 *e*→*em*	
4 *e*→*er*	2 *s*→*sf*	2 *e*→*en*	1 *u*→*ua*	1 *ot*→*to*	1 *i*→*iz*	1 *e*→*el*	

Bibliography

[1] K. Audhkhasi and A. Verma. Keyword search using modified minimum edit distance measure. In *IEEE International Conference on Acoustics, Speech and Signal Processing*, volume 4, pages 929–932, april 2007.

[2] R. A. Baeza-Yates and G. Navarro. A faster algorithm for approximate string matching. In *Proceedings of the 7th Annual Symposium on Combinatorial Pattern Matching*, CPM '96, pages 1–23, 1996.

[3] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[4] E. Brill and R. C. Moore. An improved error model for noisy channel spelling correction. In *Proceedings of the 38th Annual Meeting on Association for Computational Linguistics*, ACL '00, pages 286–293, 2000.

[5] S. Cucerzan and E. Brill. Spelling correction as an iterative process that exploits the collective knowledge of web users. In *EMNLP'04*, pages 293–300, 2004.

[6] F. J. Damerau. A technique for computer detection and correction of spelling errors. *Communications of the ACM*, 7(3):171–176, Mar. 1964.

[7] M. R. Dunlavey. Technical correspondence: on spelling correction and beyond. *Communications of the ACM*, 24(9):608, Sept. 1981.

[8] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, Jul 1968.

[9] P. E. Hart, N. J. Nilsson, and B. Raphael. Correction to "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". *SIGART bull.*, (37):28–29, Dec 1972.

[10] S. Heinz, J. Zobel, and H. E. Williams. Burst tries: A fast, efficient data structure for string keys. *ACM Transactions on Informations Systems*, 20(2):192–223, April 2002.

[11] P. Jones. *Best First Search & Document Processing Applications.* PhD thesis, Katholieke Universiteit Nijmegen, 2000.

[12] M. D. Kernighan, K. W. Church, and W. A. Gale. A spelling correction program based on a noisy channel model. In *Proceedings of the 13th conference on Computational linguistics - Volume 2*, COLING '90, pages 205–210, 1990.

[13] D. E. Knuth. *The Art of Computer Programming, volume 3: Sorting and Searching.* Addison Wesley, second edition, 1998.

[14] C. Koster. Indexation and fuzzy lexical matching. Unpublished.

[15] C. Koster, M. Seutter, and J. Beney. Classifying patent applications with winnow. In *Proceedings Benelearn 2001*, pages 19–26, 2001.

[16] K. Kukich. Techniques for automatically correcting words in text. *ACM Computing Surveys*, 24(4):377–439, Dec. 1992.

[17] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10(8):707–710, Feb. 1966. Doklady Akademii Nauk SSSR, V163 No4 845-848 1965.

[18] A. Marzal and E. Vidal. Computation of normalized edit distance and applications. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(9):926–932, Sep 1993.

[19] G. Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM*, 46:1–13, 1999.

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

[21] L. C. Noll. Fowler/Noll/Vo non-cryptographic hash algorithm. `http://www.isthe.com/chongo/tech/comp/fnv`.

[22] P. Norvig. Natural Language Corpus Data: Beautiful Data. `http://norvig.com/ngrams/`, 2008.

[23] B. J. Oommen. Constrained string editing. *Information Sciences*, 40(3):267–284, Dec 1986.

[24] J. Pedler. *Computer Correction of Real-word Spelling Errors in Dyslexic Text.* PhD thesis, Birkbeck, University of London, 2007.

[25] M. Reynaert. All, and only, the errors: more complete and consistent spelling and ocr-error correction evaluation. In *Proceedings of the Sixth International Conference on Language Resources and Evaluation (LREC'08)*, May 2008.

[26] M. Reynaert. Non-interactive ocr post-correction for giga-scale digitization projects. In *Proceedings of the 9th international conference on Computational linguistics and intelligent text processing*, CICLing'08, pages 617–630, 2008.

[27] E. S. Ristad and P. N. Yianilos. Learning string-edit distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(5):522–532, May 1998.

[28] C. Street. Ridiculously Huge English Wordlist (RIDYHEW). `http://www.codehappy.net/wordlist.htm`, 2003.

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

[30] S. Wu and U. Manber. Fast text searching: Allowing errors. *Communications of the ACM*, 35(10):83–91, Oct. 1992.