Matchsimile: A Flexible Approximate Matching Tool for Personal Names Searching

Gonzalo Navarro   Ricardo Baeza-Yates

Dept. of Computer Science, University of Chile,

Blanco Encalada 2120, Santiago, Chile.

`{gnavarro,rbaeza}@dcc.uchile.cl`.

João Marcelo Azevedo Arcoverde

Matchsimile Ltda - CTO,

Rua Ribeiro de Brito, 1002/1103, CEP 51.021-310, Recife-PE, Brazil.

`jmarcelo@matchsimile.com.br`

Abstract

In this paper we present the architecture and algorithms behind Matchsimile, *an approximate string matching lookup tool especially designed for human and company names searches against a large textual database. Part of a larger information retrieval environment, this specific engine accepts an input text file with a set of personal and company names and a set of restrictions for the search. After a batch processing, the engine outputs another text file containing the occurrences that match each record of the input names file, according to its search parameters. Beyond the similarity search capabilities applied on each word that forms a name, the tool considers a set of personal names formation rules for their words such as combination, abbreviation, character mapping, duplicity detections, ordering, word omission and insertion, among others. This engine is used in a succeeded commercial application (also named* Matchsimile*), which uses this tool to allow lawyers names searches against many official law journals publications.*

1   Introduction

Living in a world surrounded by errors and mistakes, the overwhelming existing search technologies does not address the human tendency to be inexact. They were designed to mainly focus on exact matching searches capabilities. Many situations tend to fail when those algorithms are applied, instead of performing approximate string matching searches in a textual database, where they have higher chances to suffer some kind of (undesirable) corruption, or even when the valuable information can suffer modifications the way that they can appear, in function of their nature.

Computational biology, image analysis, speech processing, medical diagnosis and legal texts constitute motivation examples for this kind of search. In the scope of the last one we have found a particular situation focused on personal names searches, where exists formation rules that can write the same piece of information in many ways. In addition, we have to add typos, spelling errors, OCR errors, etc.

Matchsimile will find a person name, a company name or a simple geographical address even if the words that form the name present errors among their characters. Suppose the following example for a hypothetical fellow named `"Juan Abigahil Eslopênio de Capriolli"`. This name is formed by five distinct words that can easily suffer modifications such as words duplicity, abbreviations, omissions, insertions and transpositions. Thus, the following occurrence triggered by *Matchsimile* would be correctly evaluated for the above example: `"Caprioli, Juam A. Slopenio"`. Easy to be detected by the human sense of similarity, but not by normal query languages, this occurrence has a large chance to be the pattern name we were looking for.

The reverse scenario is also true, when we do not know for sure what we are looking for. For example, we can query a personal name like `"Catano Velozo"` (which is wrong for the Brazilian singer and composer named Caetano Veloso) and the *Matchsimile* search engine once more will trigger occurrences for `"Caetano B. Costa Veloso"`, and so on.

Thus, *Matchsimile* allow users of an arbitrary information system to find quickly and easily the information they want, even when they are not sure the way this information should be written or matched, allowing character and word errors.

Matchsimile's revolutionary lookup technology effectively redefines the search paradigm. Under other existing technologies, the terms used in a search query strictly limit the information that the query returns; as a result, even minor errors in the query potentially return unwanted or irrelevant information. *Matchsimile* tolerates a wide spectrum of variations and errors, in an attempt to model a human notion of similarity. Despite that this is done at a simple, low syntactic level, it helps bridge the gap between human tendencies and computer requirements.

To solve the problem of string matching allowing errors, also called approximate string matching, *Matchsimile* uses a balance of theoretical data structures and advanced techniques for fast searching algorithms to model a human similarity judgment under strict time performance constraints. The mathematical properties of this type of model allow *Matchsimile* to compute each match using an extremely efficient algorithm. The result is an extremely fast search engine capable to process thousands of patterns against a large textual database (measured in Gb) in hours.

Unlike the typical "advanced search" features found on popular search engines, *Matchsimile* uses no special query syntax - no special prefixes, suffixes, brackets, braces, or Boolean connectives. Otherwise, *Matchsimile* lets the user personalize a set of length independent personal and company names, called "Inputs", to be searched against any textual database, independent of its length and language. It makes no language-specific assumptions, intelligently handling accented and special characters found in many languages.

This paper is organized as follows: Section 2 presents related work, section 3 the overall design of the system, section 4 the algorithmic techniques used by the search engine, section 5 analytical and experimental performance results, and the last section the conclusions.

2 Related Work

The algorithmic problems faced by *Matchsimile* lie in what is known as "approximate string matching", a well established field in stringology with applications in text retrieval, computational biology, pattern recognition and a dozen of other fields. The main error model used in approximate string matching permits symbol insertions, deletions, substitutions and transpositions. This model has been validated many times in the past, e.g. [10, 4, 14, 6].

The problem of approximate string matching consists of finding all the occurrences of a pattern in a text where a limited number of differences between the pattern and an occurrence is permitted. We distinguish between sequential and indexed solutions. Sequential solutions do not permit to preprocess the text. There has been research on sequential searching since the sixties, see [11] for a recent survey. Indexed solutions permit building a data structure on the text beforehand in order to answer queries later. There has been research in this trend since the nineties, see [12] for a survey.

Nevertheless, our particular problem involves searching thousands of patterns in a text allowing errors. Multiple approximate pattern matching is a rather undeveloped area, so in *Matchsimile* we have used a combination of known and new techniques. We borrow mostly from trie backtracking techniques [16, 5].

From the applications point of view, there are few systems permitting approximate matching on natural language text (there are more systems for specific computational biology applications, e.g. [5]), and none addressing our particular problem. The first such system was *Glimpse* [9], which indexes the text and permits approximate searching by looking sequentially all the vocabulary words. The same idea, with few modifications, has been used in other natural language indexes [3, 13].

A recent system relying on a slightly different approximate matching model is *LikeIt* [17]. In this system symbol transpositions are permitted and penalized according to their distances from their original positions. Based on recent algorithmic developments, *LikeIt* still does not deal with the simultaneous search of thousands of patterns.

3 System Design and Capabilities

Basically, the *Matchsimile*'s kernel accepts an input text file with a set of personal and company names (also called *patterns*) with a set of parameters which to determine the search features to be used. The engine preprocesses the patterns building an index based on suffix trees. Notice that in our application the text database changes more often than the patterns, so it is not worth to preprocess the text (for example, the official journals are printed daily). Then, using this pattern index, the engine sequentially scans the target text files. At the end, the engine outputs one text file containing the occurrences that matches each record for the input files, and optionally can output another text file containing spurious or weaker occurrences, if they were filtered through a exact search mechanism based in dictionaries.

Next we highlight the main features of the parameterization.

The system distinguishes between personal and company names. For personal names, the input file points out which are given names and surnames. In legal texts, we will

find always at least one of them (in the worst case, one is an initial). Let us examine the following example: `"Juan Carlos Bartolomeu Mattos Netto"`. It can be published like this: `"Matos, J. Carla B. Neto"`. There exists higher chances to be the person we are looking for. Note that we have one surname followed by on name, with a stopword among them with less than three characters, which we can discard with a lower cost (its existence is insignificant to the final result). The following occurrence is also honored: `"Juan Neto"`. Less chances to be the person we are looking for but can be.

Inside names we can: 1) allow intruders words insertions. Ex: `"Juan `*`Benedito`*` Neto"`; 2) set the error level applied in each word (as a % of the length); 3) filtering the results using a dictionary of names and surnames that discard spurious occurrences. All these values or actions have a default case.

Allowing intruders words (that ones that are not among those given for the original input) is meaningful if the original words are the first word of a name set and the last word of a surname set respectively. This could represent people derived from the same family. The variable error level per word is useful when you have short and long names that you want to treat differently. The filter checks for "personal names rules formation". These rules can differ from language to language, depending on their cultures and morphological/semantical constructions and speaking habits. The disadvantage of this filter is that there must exist one dictionary for names and one for surname for each language processed, so it is an optional feature.

Company names can be one or more words. Inside them we can flag which is the most important word (that typically will always appear). For example, in the following input record: `"Eletropaulo Metropolitana"`, the first word is the important one, and will appear even with errors. We also allow to have duplicate input keywords for common abbreviations which are not due to errors. For example: `"Eletrop Metrop"`.

All these parameters and actions can be predefined through a configuration file. In particular, defines which set of characters can compose one valid word. Numbers can be discarded because there is no sense to have them inside personal names. Each word is a sequence of a valid characters subset, surrounded by space character at both sides. Nevertheless, a minimal length can be specified (the default is 3). Shorter sequences are not considered words (so they cannot be intruder words).

Mapping of charactars can also be specified. This can simulate the "case insensitive" behavior, for example, and discard the accents arose from our alphabet. The "mapping" is a good technique to enhance the algorithm performance, allowing the code to work with a valid subset of characters determined by the "word characters" session.

The default cost of each error is 1. However, this can also be changed, specifying different costs for inserting, deletion, replacing, or transposing letters. In this case the maximum allowed cost to trigger a match must be specified.

4   Algorithmic Principles

We describe in this section the algorithms and data structures behind *Matchsimile*. Some of these are already known in the scientific literature, while others have been specifically developed for our needs. This last category includes a phrase matching algorithm and our overall architecture.

4.1 The Search Problem

We first define the search problem precisely, motivating the decisions taken.

Defining the text and patterns. We consider the text as a sequence of words. A *word* is a string formed by *letters* and delimited by *separators*, which can be defined by the user. On the other hand, we have a set of *patterns* to search in the text. Each pattern is formed by a sequence of *pattern words*. Patterns and text words obey the same formation rules. The user can also specify a mapping of characters, which is used to normalize every text and pattern word, as well as a set of *stopwords*, i.e. text and pattern words that will not be considered when matching.

Now that we have defined precisely what is the text and what is the set of patterns, we define the matching criterion. There are two levels of matching. A first level deals with single words and their possible typing or spelling errors. A second level deals with phrases (sequences of words) and their possible differences in arrangement.

Intraword similarity. Our first task is to determine when a text and a pattern word are *similar enough*. By "similar enough" we mean that the *cost* to transform the text word into the pattern word is smaller than a user defined threshold. The user can specify this threshold in several ways, and it can be different for every pattern word.

There are many forms to define "cost", but a popular one is the minimum number of insertions, deletions, substitutions and transposition of adjacent characters that are necessary to convert the text word into the pattern word. This is a variant over the original Levenshtein distance [7, 8].

The effectiveness of this cost measure is well known. For instance, about 80% of the typical typing errors are corrected allowing just one insertion, deletion, substitution or transposition [4]. It is also known, however [14, 6], that making every such operation to cost 1 (i.e. just counting the number of those operations) is simplistic, as much better results are achieved by permitting common errors to cost less. For example, we can give a lower cost to the transposition of two letters that are close in the keyboard or to omissions due to common spelling errors. So we choose a cost model where all these operations are permitted but we let the user change the cost of the insertion or deletion of every character, and the cost of substituting or transposing every character with every other. This permits us parameterizing the tool to different scenarios and languages.

The cost model is defined by means of two functions, δ and τ, which represent the costs to perform the diverse alterations on the text word (we could have chosen to think on altering pattern word instead). For two different letters a and b, $\delta(a, b)$ is the cost to substitute a by b in the text word (it is assumed that $\delta(a, a) = 0$). For a letter a present in the text word, $\delta(a, \varepsilon)$ is the cost to delete a from the word. For a letter a, $\delta(\varepsilon, a)$ is the cost to insert a in the word. Finally, for two different letters a followed by b, adjacent in the text word, $\tau(a, b)$ is the cost to transpose them, i.e. to convert ab into ba.

Phrase similarity. We define now when two phrases match. The first is a sequence of text words and the second is a whole pattern. From now on, we say that a text and a pattern words *match* whenever they are similar enough according to the user defined threshold, and we disregard their internal differences.

For sequences of words, we use a model where we can delete pattern words and insert text words in the pattern (or which is the same, delete text words). Permitting substitution of words seems unreasonable given that we already detect words that are close to each other and assume that they match. We found the transpositions to be of little use at this level, although for future work we are considering models where the order of the words is irrelevant.

The similarity criterion for phrases includes two thresholds. We permit deleting at most D words from the pattern, and inserting at most I spurious (text) words in the pattern. The user has several ways to specify these thresholds, in general or for specific patterns in the set. This turned out to be more adequate than setting a single threshold, say for $I + D$, because we can control more precisely the minimum amount of pattern words that must be present in order to consider that a match has occurred, as well as how many spurious words can be reasonably accepted in between interesting words.

For our particular Portuguese language application of personal and companies names searching, however, we need a finer control. This has lead to some extensions of the above matching criterion (which can be switched on or off for every pattern).

Reporting the results. The goal is to report *maximal* sequences of text words that match some pattern by outputting its exact text position (as well as the identification of the pattern matched and some information on how close is the occurrence to the correctly written pattern, used for ranking the results). The word "maximal" means that we cannot enlarge the sequence reported and still make it match.

Reporting maximal occurrences is in general a good choice because it calls the attention of the user over a longer sequence of text words that match the pattern, giving a better grasp of the relevance of the match. For example, if we permit one insertion and one deletion, then `"Maria Rosa Ferreira de Oliveira"` matches against `"Maria Ferreira de Oliveira"`, yet it also matches with the prefix `"Maria Ferreira"`.

4.2   General Architecture

Now it should be clear that our problem is to detect patterns in the text even when the words are spelled differently and arranged differently. Hence the software works at three levels: (1) Text tokenizing, a very basic layer that delimits and normalizes text words; (2) Recognizing pattern words, which recognizes the text words with enough similarity to pattern words, the similarity being measured at the character level; and (3) Recognizing whole patterns, which recognizes text phrases (sequences of words) which are similar enough to whole patterns, where we measure the similarity at the word level.

The first level implements a reading routine that delivers the text words one by one. It delimits the words, maps the characters, removes stopwords and delivers normalized words to the next level. The set of patterns is normalized according to the same rules.

The second level processes each word received against the set of all the patterns in one shot. A suitable data structure is used to arrange all the set of patterns in order to permit simultaneously comparing the text word against the whole set of patterns. As a result, this level triggers for each text word a set of occurrences (permitting errors) of the word inside the patterns, pointing out every pattern involved and specifying which pattern word has matched.

The third level is in charge of matching the whole pattern. However, it is invoked only when a text word relevant to some pattern has been recognized. This level keeps for every pattern P information about the *last* text window where the pattern could match. Since we report maximal occurrences, we need to have surpassed the area of interest before analyzing the window and reporting possible occurrences.

Hence, we run the phrase matching algorithm only over text windows that have some chance of being similar enough to a pattern. Each text word is analyzed in turn, and the patterns holding similar words get their windows updated. Those that may trigger a match are analyzed at that moment. At the end of each text document processed we increment our virtual word count by a number large enough to avoid any confusion with previous text. When we finish processing all the text collection we must check all the patterns for remaining matches not yet reported because we did not know they were maximal (note that we know that a match is maximal only when we find that the next occurrence in the text is far ahead).

The architecture is shown in Figure 1. We detail now the two most important levels.
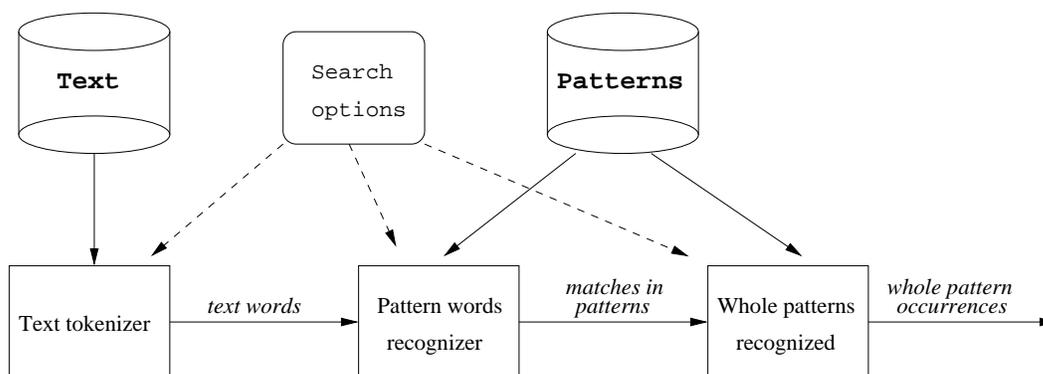
Figure 1: The architecture of the algorithm.

4.3   Recognizing Pattern Words

The first level is responsible for detecting all the text words that are similar enough to some pattern word. We first explain how to compute the similarity between a text and a pattern word, and then how to do the same against a large set of pattern words.

4.3.1   Similarity between Two Words

Let us assume that we have a text word $x_{1...n}$ and a pattern word $y_{1...m}$ and want to compute the cost to convert x into y. A well known dynamic programming algorithm [8] fills a matrix C of size $(n+1) \times (m+1)$ with the following rule:

$$
\begin{aligned}
C_{0,0} &= 0 \\
C_{i,j} &= \min (\quad C_{i-1,j-1} + \delta(x_i, y_j), C_{i-1,j} + \delta(x_i, \varepsilon), C_{i,j-1} + \delta(\varepsilon, y_j), \\
&\qquad\qquad \text{if } x_{i-1}x_i = y_j y_{j-1} \text{ then } C_{i-2,j-2} + \tau(x_{i-1}, x_i) \text{ else } \infty)
\end{aligned}
$$

where we assume that C yields ∞ when accessed at negative indices.

We fill the matrix column by column (left to right), and fill each column top to bottom. This guarantees that previous cells are already computed when we fill $C_{i,j}$. The distance between x and y is in the final cell, $C_{n,m}$.

The rationale of this formula is as follows. $C_{i,j}$ represents the distance between $x_{1...i}$ and $y_{1...j}$. Hence $C_{0,0} = 0$ because the two empty strings are equal. To fill a general cell $C_{i,j}$, we assume inductively that all the distances between shorter strings have already been computed, and try to convert $x_{1..i}$ into $y_{1..j}$.

Consider the last characters x_i and y_j. Let us follow the four allowed operations. First, we can substitute x_i by y_j (paying $\delta(x_i, y_j)$) and convert in the best possible way $x_{1..i-1}$ into $y_{1..j-1}$ (at cost $C_{i-1,j-1}$). Second, we can delete x_i (at cost $\delta(x_i, \varepsilon)$) and convert in the best way $x_{1..i-1}$ into $y_{1..j}$ (at cost $C_{i-1,j}$). Third, we can insert y_j at the end of $x_{1..i}$ (at cost $\delta(\varepsilon, y_j)$) and convert in the best way $x_{1..i}$ into $y_{1..j-1}$ (paying $C_{i,j-1}$). Finally, if $x_{i-1}x_i = y_j y_{j-1}$ then a transposition can be attempted: we convert $x_{i-1}x_i$ into $x_i x_{i-1} = y_{j-1}y_j$ (paying $\tau(x_{i-1}, x_i)$ for this) and convert in the best possible way $x_{1..i-2}$ into $y_{1..j-2}$, at cost $C_{i-2,j-2}$.

4.3.2   Comparing against Multiple Words

Now, our problem is that we have a large set of pattern words (thousands of them) and want to find every approximate match between a given text word and a pattern word. Comparing the patterns one by one is a naive solution, but we present a better one.

We address this problem as follows. We build a *trie* data structure on the set of pattern words, which permits us simulating the cost computation algorithm of Section 4.3.1 so as to compare each individual text word to all the pattern words at the same time. A trie built on a set of words is a tree with labeled edges where every node corresponds to a unique prefix of one or more words. The root corresponds to the empty string, ε. If a node corresponds to string z and it has a child by an edge labeled a, then the child node corresponds to the string za. The leaves of the trie correspond to complete words.

Let us assume that our text word is the string x and our pattern word (any of them) is y. All those pattern words y are stored together in the trie. Since each node of the trie represents a prefix of the set of patterns (in our example, the first node of the third line represents "ab", which is a prefix of two of the words of the trie), the plan is to go down the trie by all the possible branches, and fill for every node a new column of the dynamic programming matrix of Section 4.3.1. The idea is that the column computed for a node that represents the string z corresponds to the C matrix between our text string x and the pattern prefix z.

According to the formula to fill C of Section 4.3.1, we initialize the first column $C_{i,0} = \sum_{k=1}^{i} \delta(x_i, \varepsilon)$, which corresponds to the root of the trie, i.e. the empty string (which is a prefix of every pattern). Now, we descend recursively by every branch of the trie. When we descend by a branch labeled by the letter a, we fill a new column which corresponds to adding letter a to the current pattern prefix z. Hence, children nodes generate their column using that of their parent and grandparent nodes (recall that transpositions make the current column dependent on the two previous ones). Note that since a node may have several children, different columns can follow from a given one.

When we arrive to the leaves of the trie, we have computed the cost matrix C between the text word x and some pattern word y, so we check whether the last cell of the final

column is smaller than the threshold. If this is the case, then the corresponding pattern word matches the text word.

So the trie is used as a device to avoid repeating the computation of the cost against the same prefixes of many patterns. This algorithm is not new but an adaptation of existing techniques [16, 5]. We reduce the traversal cost further by performing several improvements over the basic algorithm. For lack of space we just mention the most important: it is possible to determine, prior to reaching the leaves, that the current branch cannot produce any relevant match: if all the values of the current column are larger than the threshold, then a match cannot occur since we can only increase the cost or at best keep it the same.

Figure 2 shows how to search the text word "abord" in an example trie holding the words "abacus", "aboard", "board" and "border". We assume that all the operations cost 1 and that our threshold is 2. In this case the pattern words "aboard" and "board" match, but "abacus" and "border" do not. If we computed the 4 matrices separately, we would have filled 27 columns, while the trie permitted us to compute only 19, mostly due to shared prefixes (the reduction is much larger when there are many patterns and hence many prefixes shared). In the example we do not need to traverse all the path of "abacus", since at the point of "abacu" it is already clear that a match is not possible.
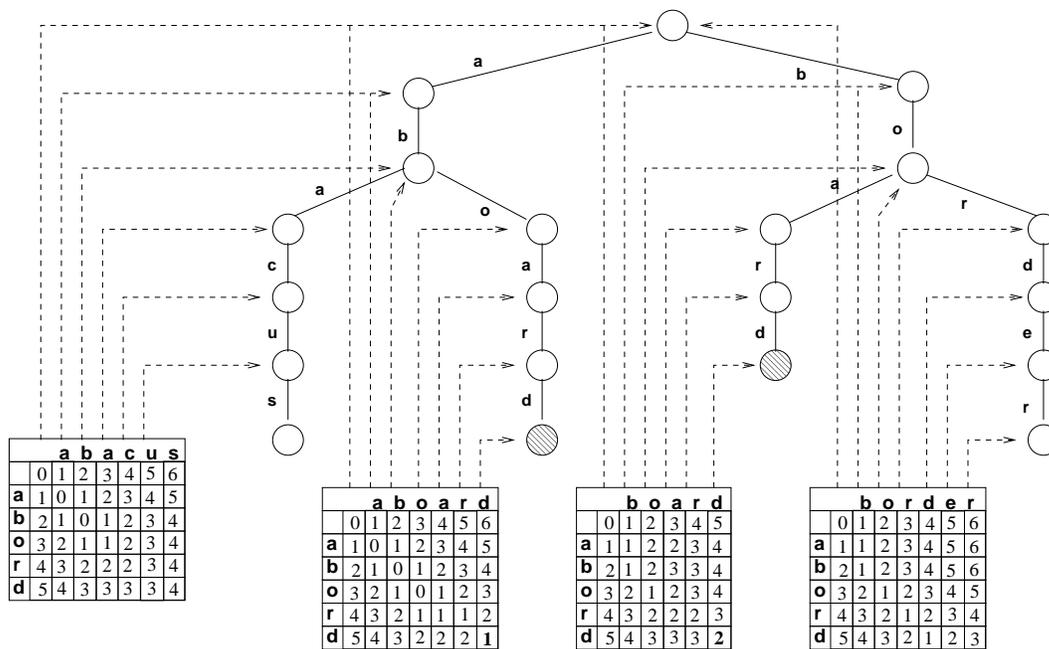
Figure 2: Searching "abord" with threshold 2 in our example trie.

4.4 Recognizing Whole Patterns

We first explain how to determine, given two sequences of words, whether they match or not under the (I, D) restriction. Later we show how to apply this algorithm only using the information of words (approximately) matched.

4.4.1 Sequential Word Matching

Let us assume that our pattern is a sequence of words $P = p_1 p_2 \ldots p_m$. Also assume that we have a specific sequence of text words $T = t_1 t_2 \ldots t_n$. Furthermore, for each text word t_i and each pattern word p_j we have precomputed the answer to the question "does t_i match p_j?". The following algorithm, which is new as far as we know, permits evaluating the similarity between P and T.

We consider the words t_i one by one, and for each new word we (re)fill a matrix W of $m + 1$ rows and $I + 1$ columns. After we have processed $t_1 \ldots t_i$, it holds that $W_{j,k}$ is the minimum number of deletions necessary to match $p_1 \ldots p_j$ against $t_1 \ldots t_i$ permitting at most k insertions. Hence, P and T match if and only if at the end it holds $W_{m,I} \leq D$.

Before processing the first text word we initialize W with the formula $W_{j,k} = j$, which means that in order to match $p_1 \ldots p_j$ against ε with at most k insertions, we need the deletion of the j pattern words (indeed the insertions are not used). When we have a new text word t_i, we update W (which refers to $t_1 \ldots t_{i-1}$) to W' using the formula

$$
\begin{aligned}
W'_{0,k} &= W_{0,k-1} \\
W'_{j,k} &= \text{if } p_j = t_i \text{ then } W_{j-1,k} \text{ else } \min(W'_{j-1,k} + 1, W_{j,k-1}), \qquad j > 0
\end{aligned}
$$

whose rationale is as follows. If we consider the empty pattern ($j = 0$), then the question is how many deletions are necessary to match ε against $t_1 \ldots t_i$ with k insertions. Clearly the answer is zero for $i \leq k$ and ∞ otherwise. Alternatively, this can be expressed as: zero if $i = 0$ (which matches our initialization $W_{j,k} = j$), otherwise the same value as for $i - 1$ with $k - 1$ insertions (which is precisely $W_{0,k-1}$). We assume that W delivers ∞ when accessed outside bounds, so the ∞ shows up when we use this scheme for $i > k$..

Let us now consider a nonempty pattern. If the new text word matches p_j, then the number of deletions necessary to match $p_1 \ldots p_j$ against $t_1 \ldots t_i$ permitting up to k insertions is the same as that for matching $p_1 \ldots p_{j-1}$ against $t_1 \ldots t_{i-1}$ permitting up to k insertions. Otherwise we must do something with those p_j and t_i that refuse to match. A first choice is to get rid of the last p_j (paying a deletion) and match in the best possible way $p_1 \ldots p_{j-1}$ against $t_1 \ldots t_i$, which can be done with $W'_{j-1,k}$ deletions (we keep k because we have not used insertions). Note that we use W' instead of W because we refer to i, not $i-1$. The second choice is to get rid of the last t_i by inserting it at the end of $p_1 \ldots p_j$, and then convert in the best possible way $p_1 \ldots p_j$ into $t_1 \ldots t_{i-1}$, using $W_{j,k-1}$ deletions (it is $k - 1$ because we have used one insertion).

It is easy to keep W and W' in the same matrix, as long as we fill it for decreasing values of k and inside each k for increasing values of j.

Something that is interesting for what comes next is that, if we know that the next s text words do not match against any pattern word, then we can directly skip them in one shot. The reason is that the only way to deal with these words is inserting them into the pattern, so for each of them we will have to shift all the $W_{j,k}$ values to the right. Faster than that is to shift virtually, i.e. keep a Δ value initialized in zero and accessing $W_{j,k-\Delta}$ every time we need the value of $W_{j,k}$. Hence, we can process the sequence of s text words by assigning $\Delta \leftarrow \Delta + s$.

For lack of space we omit the modifications necessary to accomodate the particular restrictions for matching personal and company names.

4.4.2 Operating with Triggered Occurrences

Finally, we explain how we simulate the algorithm of Section 4.4.1 when, for a given pattern, we are only notified of relevant words that appear as the text is scanned.

We keep for every pattern P a list of up to $m + I$ pairs $(pos_1, mask_1) \ldots (pos_\ell, mask_\ell)$, where pos_r is the index of a text word that has matched a word in P and $mask_r$ is a bit mask (of m bits) indicating which pattern words have been matched by t_{pos_r}. The positions are in increasing order in the list, $pos_r < pos_{r+1}$. After we have processed text word t_i, the following invariants hold on the list of pairs stored for every P: (1) Every occurrence ending before pos_1 stored has already been reported. (2) It holds $pos_\ell - pos_1 + 1 \le m + I$. Since $m + I$ is the maximum possible length of an occurrence of P, this means that all the window could be part of a single occurrence, and hence we still do not have enough information to determine a maximal match starting at pos_1.

The word matching algorithm processes each text word in turn. Some data are stored at the trie leaves so that each time a word y is found in the trie, we can identify the patterns the word y belongs to and its index(es) in those patterns (i.e. those (P, j) such that $y = p_j$). For each of these patterns involved, we have to carry out some actions.

First, say that we find that a word t_i matches $y = p_j$. We start by adding $(pos_{\ell+1}, mask_{\ell+1}) = (i, \{j\})$ at the end of the list of P (and increment ℓ of course). In fact it is possible that t_i has already matched some other word of P, in which case $i = pos_\ell$. In this case we do not add a new entry to the list but simply add j to the set represented by the bit mask $mask_\ell$, indicating that t_j also matches p_j.

If the enlargement of the window does not make it exceed the size $m + I$, nothing else needs to be done. However, if after the insertion we have that $pos_\ell - pos_1 + 1 > m + I$, then we need to restore the invariants. We have now information on a text area that spans more than $m + I$ words, which is enough to report at least maximal matches starting at pos_1.

The idea is then to remove pairs from the beginning of the list until it covers an area not larger than $m + I$. However, prior to deleting each pair, we must make sure that a maximal match cannot start at it. So, while $pos_\ell - pos_1 + 1 > m + I$, we check for a maximal occurrence starting at text position pos_1. If it is not found, we remove the first entry $(pos_1, mask_1)$ and make the list start at pos_2. If, on the other hand, we find a maximal match spanning the text area $[pos_1, pos_e]$, we report it and make the list start at $pos_e + 1$. This last assertion means that our reporting is greedy, i.e. no overlapping sequences are reported.

Checking for a maximal occurrence is done using the sequential word matching algorithm of Section 4.4.1: we initialize the matrix and feed it with the text words of the window. The precomputed answers to "$p_j = t_i$?" are precisely in the bit mask $mask_i$ (so we do not have to really look at the text). We abandon the algorithm only when it holds $W_{j,I} > D$ for all j (since no occurrence can appear later). This eventually happens because our window is long enough. When this finally occurs, we check which was the last window position where we found a match, i.e. the last position pos_e where the matrix satisfied $W_{m,I} \le D$. If this ever happened, then that e is the end of a maximal occurrence, otherwise there are no occurrences starting at pos_1. Occurrences are reported at their exact text positions thanks to information kept together with every pair.

Note that between consecutive entries $(pos_r, mask_r)$ and $(pos_{r+1}, mask_{r+1})$ we have

$pos_{r+1} - pos_r - 1$ text words that match no pattern word. Here is where we use our ability to process all the gap in one shot.

We can avoid the sequential matching in some cases. First, if the length of the list of pairs is $\ell < m - D$, then we will need more than D deletions to match it. Second, if the accumulated gap length $pos_\ell - pos_1 - (\ell - 1) > m + I$ then we will need more than I insertions. We keep track of those values so as to verify as little as possible.

5 Performance

We consider now the performance of our system, both in theory and in practice.

Analysis. Let us assume that we have a text of N words, where we have to search for M patterns of m words each, permitting I insertions and D deletions when matching phrases. Assume that words have w letters on average. We estimate the cost of our algorithm as follows.

The trie of the Mm words has $O(Mm)$ nodes on average [15]. Each time we traverse it with backtracking permitting a maximum error threshold we touch $O((Mm)^\alpha)$ nodes, for some $0 < \alpha < 1$ that depends on the threshold and the costs of the operations (under a simple model of constant probability of traversing an edge [1]). Since we fill a column of the matrix at each node, we have a total cost of $O(Nw(Mm)^\alpha)$ average time for recognizing pattern words. The space necessary for the traversal is that to store the trie, $O(Mm)$, plus that of the backtracking. This last one is proportional to the height of the trie because we need to store only the columns of the current path during the backtracking, which gives $O(w \log(Mm))$ [15].

Let us now consider recognizing whole patterns. Unlike words, each complete pattern is in general different from the rest, so we can consider that their probability of occurring (approximately) in the text is additive. Hence, if we have M patterns we expect that they will trigger $O(NM)$ verifications (albeit multiplied by a very small constant). Assuming that every time a word from a pattern appears we add a node to the list of that pattern, and that we are unable to avoid verifications, we have that every node that enters the list needs to exit it, and in order to exit the list a verification is necessary. The verification needs to fill the W matrix, of size $O(Im)$, a number of times which is at most $m + I$. Hence the total cost of this level is pessimistically bounded by $O(MNIm(m + I))$. The space required is that of one list per pattern, $O(M(m + I))$.

Hence the total cost of the algorithm is $O(N(w(Mm)^\alpha + MmI(m + I)))$ and the space is $O(Mm + w \log(Mm) + M(m + I))$. If we are interested in the behavior of the algorithm when the text size N or the number of patterns M grow, and want to assume that the other quantities D, I, m and w remain more or less constant, then we can make the simpler statement that the algorithm is $O(NM^\alpha)$ time to traverse the trie and $O(NM)$ to process the sequences of words. Despite that this is formally $O(NM)$, in practice the time spent at the trie dominates, as processing the sequences of words is multiplied by a much smaller constant. Hence in practice the algorithm behaves more like $O(NM^\alpha)$ for $0 < \alpha < 1$. The space required is $O(M)$.

Experimental results. We have tested our algorithm in a real case (see a brief description in the Conclusions).

We took our measures in a development machine, a Sun UltraSparc-1 of 167 MHz and 64 Mb of RAM, running Solaris 2.5.1. Since there is no doubt that the algorithm is linear time with N, we have fixed a text of 1 Mb size. In this text, we searched the first $M = 5,000$ names of our test data, the first $M = 10,000$ names, and so on until $M = 65,000$. Also we have noticed that the process is strongly CPU bound, so we measure user times, as these turn out to be very close to elapsed times.

Figure 3 shows the results. We show a plot with all the figures and also a zoomed version to appreciate the cheaper parts of the cost.
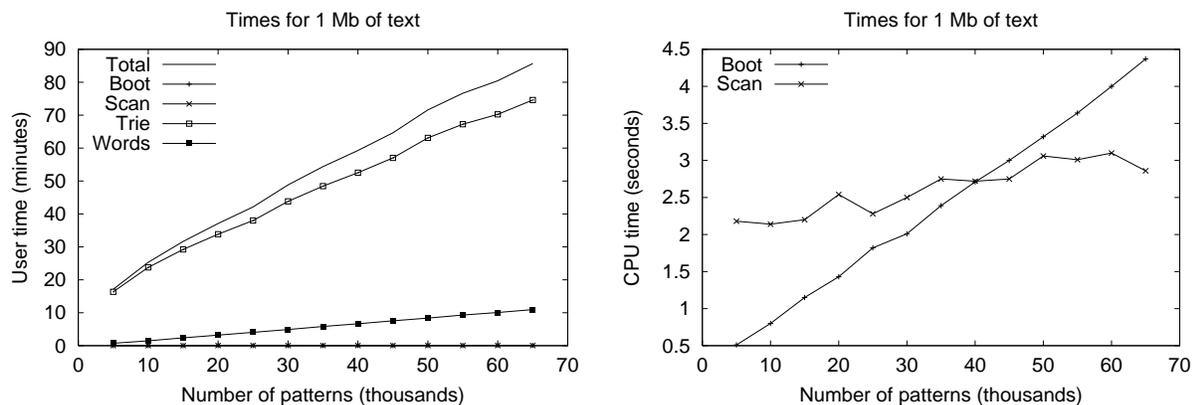
Figure 3: Search time for 1 Mb of text as the number of patterns M grows. On the left we show all the times and on the right a detail of the cheaper costs.

The *Boot* time is that of loading the M patterns from disk and setting up the trie and other data structures to start the search. As it can be seen, this cost is negligible compared to the rest. Predictably, it grows linearly with M and it is independent of N. A least squares estimation yields $0.17 + 6.4 \times 10^{-5}M$ seconds (1.5% of relative error).

The *Scan* time is that of tokenizing the text: reading , separating the words, normalizing, removing stopwords, keeping positional information to permit reporting the exact positions of the occurrences, searching the dictionaries, etc. This is basically dependent on N, although there is a slight dependence on M probably due to less locality of reference as M grows. A least squares estimation yields $2.05N + 1.6 \times 10^{-5}MN$ (4% of relative error), where N is measured in megabytes (not in words).

The *Trie* time is that of searching every relevant word in the trie of pattern words, with backtracking. This is by far the heaviest part of the process, and it is clearly sublinear. A least squares estimation yields $5.69NM^{0.6}$, with a relative error of 3%.

Finally, the *Words* time is that of verifying potentially relevant sequences of words. We argumented that this process was linear on M, so we now check the hypothesis cNM^b, obtaining $0.005NM^{1.04}$, which shows that it is effectively linear. Under a model of the form cNM we obtained $0.01NM$, with 3.7% of error.

Hence, the total cost in our machine to process M patterns on N Mb of text is $N \times (2.05 + 5.69M^{0.6} + 0.01M)$, discarding negligible contributions.

The constants in the result depend on the machine we used and are only illustrative of the relative importance of the main parts of the algorithm and of their growth rate in

terms of N and M. The production machine in *Matchsimile* is right now an Intel 700 MHz machine with 128 Mb of RAM, with a common IDE hard disk. In this machine we search all the 65,000 patterns in 60 Mb of text every day, in an elapsed time of 3 hours, much faster than in our development machine.

Let us compare this performance against that of LikeIt. As reported in [17], that tool is able of scanning the text for one pattern at a rate of 2.5 Mb/sec on an Intel 200 MHz processor. Lacking multipattern search capabilities, the search for M patterns in N megabytes of text would take about $0.4NM$ seconds. Extrapolating to our machine of 700 MHz, scanning 60 Mb for 65,000 names would require 5 days.

The result also depends on the search parameters. The values reported represent a realistic scenario, since they correspond to the current real world application where *Matchsimile* is being used.

6 Conclusions

The first well succeeded commercial application of this software is also called *Matchsimile*. Despite that more tuning of the parameters is still needed, the combination of performance and precision/recall has proven very good in practice for this application, which has been responsible for the development of the software and has pushed the improvement of the code performance and capabilities to adapt it to new circumstances.

The objective of this application is to retrieve lawyers names from official law journals in Brazil and gather that information for each company, personalized through daily reports that can be retrieved by www, html-mail and wap-enabled celulars (`www.matchsimile.com`). Currently, three official publications are scanned daily: DOSP (Diario Oficial de Sao Paulo), DOMG (Diario Oficial de Minas Gerais) and DOPE (Diario Oficial de Pernambuco). Nevertheless, this tool can be useful for other applications, like eliminating duplacates in addresses lists or to identify a client in software that handles customer complaints via e-mail.

Future plans with *Matchsimile* include sorting the output by decreasing similarity with the input and incorporating new models for word matching where the order between words is not important (useful for company names in some cases, and for personal names with some modifications). On the side of the efficiency, we plan to improve it using a more sophisticated technique: right now we build a trie of patterns and search every text word sequentially. This avoids repeating the same work for similar pattern prefixes, but similar text words are processed over an over. Using a technique known in computational biology to find all the approximate matches between two tries [2], we plan to build a trie with the text words and match it against the trie of pattern words. Since the whole text will not fit in main memory, the text will be divided in chunks of appropriate size and each chunk will be processed as a whole trie against the patterns.

References

[1] R. Baeza-Yates and G. Gonnet. Fast text searching for regular expressions or automaton search ing on a trie. *Journal of the ACM*, 43(6):915–936, 1996.

[2] R. Baeza-Yates and G. Gonnet. A fast algorithm for all-against-all sequence matching. In *Proc. String Processing and Information Retrieval (SPIRE'98)*, pages 16–23. IEEE CS Press, 1998.

[3] R. Baeza-Yates and G. Navarro. Block-addressing indices for approximate text retrieval. *Journal of the American Society for Information Science (JASIS)*, 51(1):69–82, January 2000.

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

[5] G. Gonnet. A tutorial introduction to Computational Biochemistry using Darwin. Technical report, Informatik E.T.H., Zurich, Switzerland, 1992.

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

[7] V. Levenshtein. Binary codes capable of correcting spurious insertions and deletions of ones. *Problems of Information Transmission*, 1:8–17, 1965.

[8] R. Lowrance and R. Wagner. An extension of the string-to-string correction problem. *Journal of the ACM*, 22:177–183, 1975.

[9] U. Manber and S. Wu. GLIMPSE: A tool to search through entire file systems. In *Proc. USENIX Technical Conference*, pages 23–32. USENIX Association, Berkeley, CA, USA, Winter 1994.

[10] H. Masters. A study of spelling errors. *University of Iowa Studies in Education*, 4(4), 1927.

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

[12] G. Navarro, R. Baeza-Yates, E. Sutinen, and J. Tarhio. Indexing methods for approximate string matching. *IEEE Data Engineering Bulletin*, 24(4):19–27, 2001. Special issue on Managing Text Natively and in DBMSs. Invited paper.

[13] G. Navarro, E. Moura, M. Neubert, N. Ziviani, and R. Baeza-Yates. Adding compression to block addressing inverted indexes. *Kluwer Information Retrieval Journal*, 3(1):49–77, 2000.

[14] J. Nesbit. The accuracy of approximate string matching algorithms. *Journal of Computer-Based Instruction*, 13(3):80–83, 1986.

[15] R. Sedgewick and P. Flajolet. *Analysis of Algorithms*. Addison-Wesley, 1996.

[16] H. Shang and T. Merrettal. Tries for approximate string matching. *IEEE Transactions on Knowledge and Data Engineering*, 8(4), August 1996.

[17] P. Yianilos and K. Kanzelberger. The LIKEIT intelligent string comparison facility. Technical report, NEC Research Institute, 1997.