Matchsimile: A Flexible Approximate Matching Tool for Searching Proper Names

Gonzalo Navarro and Ricardo Baeza-Yates
Department 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

We present the architecture and algorithms behind Matchsimile, an approximate string matching lookup tool especially designed for extracting person and company names from large texts. Part of a larger information extraction environment, this specific engine receives a large set of proper names to search for, a text to search, and search options; and outputs all the occurrences of the names found in the text. Beyond the similarity search capabilities applied at the intraword level, the tool considers a set of specific person name formation rules at the word level, such as combination, abbreviation, duplicity detections, ordering, word omission and insertion, among others. This engine is used in a successful commercial application (also named Matchsimile), which allows searching for lawyer names in official law publications.

1 Introduction

Most current search technology focuses on exact searching. Although appropriate for classical scenarios, several new applications need a more sophisticated form of searching. Computational biology, image analysis, speech processing and natural language text searching are examples where exact searching is of little use. For different reasons, in these applications valuable information can suffer some kind of corruption or it may not be clear which is the exact pattern we look for. These applications need “approximate searching” capabilities, that is, finding objects which are “similar,” albeit not necessarily equal, to a search pattern.

On the other hand, text processing applications, such as information retrieval and filtering, natural-language understanding and machine translation, need to identify multi-word expressions that refer to proper names of people, companies, places and other entities. The recognition of known names in a target document collection differs thoroughly from the discovery of new names, which relies upon context and word knowledge. Even recognizing a known name in a text presents problems. The first is the possibility of typing, spelling, or other errors at the character level. The second is due to different name formation rules, which permit writing a name in different ways and normally involves abbreviations and differences at the words level. An effective name recognition system has to consider the cultural rules that affect name formation.

In this paper we focus on a real application related to searching legal texts for a large set of relevant person and company names, in Portuguese language. Several complications arise from the name formation rules and from typos, spelling errors, optical character recognition (OCR) errors, etc. Another particularity of our application is that we want to search for thousands of names in parallel, that these queries are static (that is, we know the set of names in advance) and that the text databases are dynamic (more text is added every day). This problem could be seen as a particular information extraction problem.

Matchsimile can find a person name, a company name or a simple geographical address even if the words that form the name present errors at the character level. Suppose the following example for a hypothetical fellow named “Juan Abighail 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 will be correctly triggered by Matchsimile: “Caprioli, Juan A. Slope-nio”. Easy to detect by a human sense of similarity, but not by classical query languages, this occurrence has a big chance to be the pattern name we are looking for.

Received August 6, 2001; revised May 21, 2002; accepted July 3, 2002
© 2003 Wiley Periodicals, Inc.
The reverse scenario is also true, when we do not know for sure what we are looking for or the query is spelled incorrectly. For example, we can search for a person name like “Catano Velozo” (which is wrong for the Brazilian singer and composer named Caetano Veloso) and Matchsimile will trigger occurrences for “Caetano B. Costa Veloso”, and so on.

Thus, Matchsimile allows users to find the information they want, even when they are not sure the way this information should be written or matched, allowing character-level and word-level errors. Unlike the typical “advanced search” features found in popular search engines, Matchsimile uses no special query syntax—no prefixes, suffixes, brackets, braces, or Boolean connectives. On the other hand, Matchsimile lets the user personalize a set of matching options that permit adapting it to different search scenarios, precision/recall tradeoffs, languages and types of errors.

The system 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, most of the relevant variations are captured. Matchsimile uses a combination of known and new techniques for fast multipattern approximate searching. The result is an extremely fast search engine capable of processing thousands of patterns against a large textual database (a few Gb) in a few hours.

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 via Web, html-mail and wap-enabled cellulars (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 duplicates in addresses lists or to identify a client in software that handles customer complaints via e-mail.

This paper is organized as follows: Section 2 presents related work, section 3 name formation rules, section 4 a precise definition of the problem, section 5 the system architecture, sections 6 and 7 the main algorithmic techniques used, section 8 analytical and experimental performance results, and the last section some concluding remarks. A preliminary version of this article appeared in Navarro, Baeza-Yates, and Arcoverde (2001).

2 Related Work

Two related but widely different problems are discovering (previously unknown) names in text and searching for known names in text. In the former case there is no pattern to search for, but one has to discover what is and what is not a name. Several existing works address this problem, e.g., Ravin and Wacholder, 1996; Ravin, Wacholder, and Choi, 1997; Baluja, Mittal, and Sukthankar, 2000; Bikel, Miller, Schwartz, and Weischedel, 1997; Cucerzan and Yarowsky, 1999; Dozier and Haschart, 2000; Strunk, 1991; and Gross, 1990. A related issue is the so-called “authority control”, where there is a set of names and one has to cluster them into groups that represent the same entity. This is particularly interesting in online libraries and catalogs (Weintraub, 1991; Drabenstott & Weller, 1996; Taylor, 1999), as well as several other applications in art history, bibliography, commerce, genealogy, and law enforcement, with the aims of authority control, information retrieval, and duplicate detection.

With respect to our particular focus of searching for known names, some work has been carried out as well. An interesting example is Synoname (Borgman & Seigfried, 1992; Seigfried and Bernstein, 1991), a system that searches for a given person name by using a sequence of twelve algorithms for pattern matching that include both character- and word-matching techniques. The matched pairs of names are considered to be “candidate matches” until confirmed by a human name-authority editor. The need to consider not only differences at the character level but also at the word level is argued for in (French, Powell, and Schulman, 1997).

A recent system relying on a slightly different approximate matching model is Likelt (Yianlos & Kanzelberger, 1997). In this system symbol transpositions are permitted and penalized according to their distances from their original positions.

The main features that distinguish Matchsimile from previous approaches are (a) its focus on Portuguese language and Brazilian culture for name formation, and (b) its ability to efficiently search for thousands of patterns simultaneously, which is a nontrivial algorithmic challenge. As far as we know, no previous work exists on these two subjects.

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., Masters, 1927; Damerau, 1964; Nesbit, 1986; and Kukich, 1992.

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 Navarro (2001) and Navarro and Raffinot (2002) for recent surveys. 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 Navarro, Baeza-Yates, Sutinen, and Tarhio (2001) for a survey.

From the applications point of view, we can mention some systems for specific computational biology applications such as Darwin (Gonnet, 1992), as well as some for natural language such as Glimpse (Manber & Wu, 1994), 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 (Baeza-Yates & Navarro, 2000; Navarro, Moura, Neubert, Ziviani, & Baeza-Yates, 2000). The latter permits approximate matching at the word level.

Nevertheless, our particular problem involves approximate searching for thousands of patterns (fixed in advance) in a dynamic text, which can be seen as a particular information extraction problem. Multiple approximate pattern matching is a rather undeveloped area (Navarro, 2001; Navarro & Raffinot, 2002), so in Matchsimile we have used a combination of known and new techniques, from formal algorithmics to heuristic considerations. For the former, we borrow mostly from trie backtracking techniques (Shang & Merrettal, 1996), while for the latter we rely on name formation rules, which are described next.

### 3 Name Formation Rules

The information nature of proper names, depending on a given culture or source of information, demands information retrieval models and techniques specific to filter and rank its occurrences. Independently of cultural and linguistic factors that determine the proper names formation structure, we have that in all cases they can be presented under diverse forms, formal or informal, that identify the same proper name. An example is the case of the partial or complete abbreviations, words omission and transpositions. Let us examine variants for the Portuguese name “Juan Abigahil Eslopênio de Capriolli”: (a) “Capriolli, Juan A. Eslopênio”, (b) “Juan Capriolli”, (c) “Abigahil Eslopênio de C.”, (d) “Mr. Capriolli”, (e) “JAEC”, and so on. Brazilian person names have two parts (indeed this is common in most cultures). The first, which we will call “given names”, are those chosen for the person by his/her parents (“Juan Abigahil” in our example). The second, which we call “family names”, are those inherited from the family in one way or another (“Eslopênio de Capriolli” in our example).

With respect to the cultural aspect, proper names are susceptible to appear under one or more morphological structures that had consolidated certain standards of presentation, becoming conventions for a given language, different from others. It is the case of the Brazilian culture, where the last family name is almost always inherited from the father and the first family name from the mother. The last family name has a greater cultural value as a result of the legacy of a past male’s world. Then we assign more importance to the last family name. In the Spanish language, on the other hand, the first family name is inherited from the father and hence has a bigger value, but also it is much more common than in Brazil to omit the mother’s family name. In both languages, the first given names are usually more important than the others, but this rule has exceptions, in particular when the first given name is a short and common one, such as “Juan.”

In the English language, it is common to present a proper name by putting the father’s family name before the first given name, followed by a comma, as in our example (a). This occurs much less frequently in Brazil, almost always as the result of copying the English transposing convention.

These are basically cultural aspects that determine the morphological structure of proper names formation. However, there is another determinant factor in the formation of proper names: the nature of the target document collection. In official sources of information, a consensus for the presentation of proper names exists, demanding that at least one given name and one family name must appear. When only one of the two appears, this constitutes an informality that almost always is preceded by a previous occurrence that follows the formal rule. In our example, the occurrence (d) would disobey this rule, as well as (c) and (e), as abbreviating a name is not a formal way to present it.

### 4 The Search Problem

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

#### 4.1 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. The choice of which characters are letters and which are separators is configurable by the user. On the other hand, we have a set of patterns to search for 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. Typical normalizations are mapping to lower case and (sometimes) discarding accents, since these differences tend to obscure similarities. Finally, the user can specify a set of stopwords, i.e., text and pattern words that will not be considered when matching. Typical stopwords are articles, prepositions and other connectives. The reason is that a lot of time is saved (about one out of two text words is a stopword) and that connectives are not very important when matching names.

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.

#### 4.2 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 (Levenshtein, 1965; Lowrance & Wagner, 1975).

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 (Damerau, 1964). It is also known, however (Nesbit, 1986; Kukich, 1992), 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, δ(a, b) is the cost to substitute a by b in the text word (it is assumed that δ(a, a) = 0). For a letter a present in the text word, δ(a, ε) is the cost to delete a from the word. For a letter a, δ(ε, a) is the cost to insert a in the word. Finally, for two different letters a followed by b, adjacent in the text word, τ(a, b) is the cost to transpose them, i.e. to convert ab into ba.

All these values can be defined by the user to fit different needs and error scenarios. If it holds that δ(a, ε) = δ(ε, a), δ(a, b) = δ(b, a) and τ(a, b) = τ(b, a), then the cost to transform the text word into the pattern word is the same as that of doing the reverse process, because inserting in the text costs the same as deleting in the pattern and vice versa. This is normally the choice, but we do not force that. We permit the cost of operations in the text and pattern words to be different, which may be useful. For example, we may expect the patterns to be written more carefully than the text (in our first real application, for example, the patterns are names of persons written by themselves). In this case, if, say, inserting letter a is a common mistake but omitting it is rare, then we can give the insertion in the text a low cost, but a high cost to the deletion.

Note that if the cost to transform text to pattern is the same as to transform pattern into text, then we have a distance. A distance function has to be symmetric (d(x,y) = d(y,x)), but also satisfy d(x, x) = 0 and d(x, z) ≤ d(x,y) + d(y, z). The two latter conditions hold on any function defined as the minimum additive cost of a sequence of steps to convert x into y.

4.3 Phrase Similarity

We now define 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 between interesting words.

For our particular Portuguese language application of person and company 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):

- Company names may specify a “most relevant word” that must be present in every match. Otherwise we risk matching “Banco Meridional do Brasil” with “Banco Sudameris Brasil” or with “Banco do Brasil” because we miss the point that the important word is “Meridional”. We also permit specifying that all the words are relevant, so deletions are not permitted.
- Person names cannot match unless at least one given name and one family name appears. This permits discarding a huge number of false matches taking advantage that a given or family name never appears alone in this application.
- Person names cannot match if they are part of a longer name. This, again, permits us discarding a lot of irrelevant matches. Since the person names are written by their owners, we can expect them to write their first given name and their two family names (in Brazil). Hence, if there are recognizable given names preceding or family names following the candidate match, we decide that it is not a correct match. This choice has some disadvantages, such as the need to maintain a dictionary of given and family names and the possibility of missing some relevant occurrences (e.g., in a stream of given and family names we can miss some elements because in
Portuguese some words can be both given and family names; or because some Portuguese names are also common words and hence these cannot be used to discard matches).

- In our application given names appear always before family names.

## 4.4 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.”

## 5 System 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. The main data handled by the system consists of four files:

- **Names file**: Contains the person and company names to search for, each with a unique identifier. Each name specifies the number of word insertions and deletions allowed. For person names, the given and family names are distinguished, and one can specify whether or not to use dictionaries. For company names, the most important word can be specified.
- **Text files**: Contain the text to scan.
- **Search options**: Specifies global search options such as the cost to edit characters, intraword error level permitted depending on the length, character mapping, minimum word length, stopwords, dictionary files, and so on.
- **Output file**: Produced by the system, is a sequence of occurrences. Each occurrence specifies identifier of the name, text file, row and columns of the match. The software works at three levels.
- **Text tokenizing**: This is a very basic layer that delimits and normalizes text words.
- **Recognizing pattern words**: This level recognizes the text words with enough similarity to pattern words, the similarity being measured at the character level.
- **Recognizing whole patterns**: This level recognizes text phrases (sequences of words) which are similar enough to whole patterns, where we measure the similarity at the word level.

One of our driving principles is that the algorithm has to make just one pass over the text, never requiring to go back or to have a large internal buffer for the text. The reason is that if we achieve this we have better scanning time and lower memory requirements (independently of the text size). In addition this makes the algorithm to be on-line without really needing to store the text.

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. In the next two sections we detail the two most important levels, focusing on the algorithms and data structures used. 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.

6 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.

6.1 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 (Lowrance & Wagner, 1975) fills a matrix $C$ of size $(n + 1) \times (m + 1)$ with the following rule:

$$C_{00} = 0$$

$$C_{ij} = \min \left( C_{i-1,j-1} + \delta(x_i, y_j), \quad C_{i-1,j} + \delta(x_i, \varepsilon), \quad C_{i,j-1} + \delta(\varepsilon, y_j) \right)$$

where we assume that $C$ yields $\approx$ 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 $C_{i,j-1}$). If $x_i = y_j$, then a transposition can be attempted: we convert $x_{i-1}y_j$ into $x_{i-1}y_j$, and then convert in the best possible way $x_{1..i-2}$ into $y_{1..j-2}$ (at cost $C_{i-2,j-2}$).

6.2 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, using the fact that the set of patterns is known in advance.

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 6.1 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, $\varepsilon$. 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. Figure 2 shows an example trie.

Let us assume that our text word is the string $x$ and our pattern word (any of them) is $y$. All those pattern words 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 6.1. 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 6.1, we initialize the first column $C_{i,0} = \sum_{k=1}^{l_i} \delta(x_i, \varepsilon)$, 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 pre-
vious 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 (Shang & Merrettal, 1996; Gonnet, 1992; Baeza-Yates & Gonnet, 1998).

We reduce the traversal cost further by performing several improvements over the basic algorithm.

1. 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.

2. Since we permit transpositions, in order to perform the above pruning we must check also that the column of the parent has all its values larger than the threshold minus the minimum cost of a transposition. Moreover, a cell $C_{i-1,j-1}$ can be candidate for a transposition only if $x_{i-1} = y_j$ (we do not know $y_{i+1}$ yet).

3. It is not necessary to compute the whole matrix in order to know the result. If the threshold is $k$, the cheapest deletion of a text character costs $d_{\text{min}}$ and the cheapest insertion of a text character costs $i_{\text{min}}$, then there can be at most $\max_j = \lceil k/i_{\text{min}} \rceil$ insertions and $\max_d = \lceil k/d_{\text{min}} \rceil$ deletions. Hence we just need to compute a band around the main diagonal, namely $j \in [i - \max_d, i + \max_d]$.

Figure 3 shows how to search for the text word “abord” in our example trie. 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.

7 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.

7.1 Sequential Word Matching

Let us assume that our pattern is a sequence of words $P = \text{p}_1\text{p}_2 \ldots \text{p}_m$. Also assume that we have a specific sequence of text words $T = \text{t}_1\text{t}_2 \ldots \text{t}_n$. Furthermore, for each text word $t_i$ and each pattern word $p_j$, we have precomputed the answer to the question “does $t_i$ approximately 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 $n + 1$ 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 $t_1 \ldots t_i$ 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

$$W'_{j,k} = \begin{cases} W_{0,k-1} & j > 0 \end{cases}$$

whose rationale is as follows. If we consider the empty pattern ($j = 0$), then the question is how many deletions are necessary to match $\varepsilon$ against $t_1 \ldots t_i$ with $k$ insertions. Clearly the answer is zero for $i \leq k$ and $\approx$ 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 $\approx$ when accessed outside bounds, so the $\approx$ shows up when we use this scheme for $i > k$.

Let us now consider a nonempty pattern. In the new text word $t_i$, the number of insertions necessary to match $p_1 \ldots p_j$ against $t_1 \ldots t_i$ with $k$ deletions is the same as that for matching $p_1 \ldots p_{j-1}$ against $t_1 \ldots t_{i-1}$ with $k$ 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 $t_1 \ldots t_{i-1}$, which we can do 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 $j$. Figure 4 illustrates the process.

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 $\Delta$ 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 $\Delta \leftarrow \Delta + s$.

### 7.2 Given Names, Family Names, and Necessary Words

We show now the modifications necessary to accommodate the particular restrictions for matching person and company names.

**Distinguishing necessary words.** In order to account for the "most important word" of companies, we simply avoid the pattern word deletion rule ($W_{j,k} = W'_{j-1,k} + 1$) when $j$ is the index of the word that cannot be deleted. If the specification is that all the words are important, then we just set $D = 0$, as no pattern word can be deleted.

**Matching at least one given name and one family name.** This is more complicated than it seems. Despite that the patterns show a clear separation between given and family

---

**FIG. 4.** The process of including a new text word $t_i$ and keeping the invariants on $W$. 

```plaintext
\[
W = \begin{pmatrix}
0 & k-1 & k & 1 \\
0 & 0 & k-1 & 1
\end{pmatrix}
\]

\[
W' = \begin{pmatrix}
0 & k-1 & k & 1 \\
0 & 0 & k-1 & 1
\end{pmatrix}
\]

\[
\text{word deletions to match } p(1..j) \text{ to } t(1..i-1) \\
\text{word deletions to match } p(1..j) \text{ to } t(1..i)
\]
```
Later, we do not permit a person name occurrence to start at 
We mark together with the occurrence whether the word is 
vious and next word in the given and family names tries, 
(permitting errors at the character level) we check its pre-
a dictionary of given names and another one on family names.
In order to discard 
Discarding non-maximal person names. In order to discard 
matches of person names preceded by a given name or 
followed by a family name, we keep a trie data structure on 
a dictionary of given names and another on family names. 
Each time a word in a person name pattern is recognized 
(permitting errors at the character level) we check its pre-
vious and next word in the given and family names tries, 
respectively (without permitting errors at character level). 
We mark together with the occurrence whether the word is 
preceded by a given name or followed by a family name 
(recall that at this point we cannot know whether the 
matched word is a given name or a family name or both). 
Later, we do not permit a person name occurrence to start at 
a word preceded by a given name or to finish at a word 
followed by a family name.

There are good reasons to permit only exact searching in 
the given and family names tries. First, a name sought with 
errors can match an unexpected number of other words, so 
we have a good chance of incorrectly assuming that the 
word is surrounded by a given or family name and missing 
a perfectly clear match. We prefer that if the name is part of 
a bigger name but this bigger name is incorrectly written, 
then we report it anyway. Second, the efficiency would be 
severely degraded if we performed three approximate trie 
searches per word instead of one.

7.3 Operating with Triggered Occurrences

Finally, we explain how we simulate the algorithm of 
Section 7.1 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 
\((pos_i, mask_i)\) \((pos_r, mask_r)\), 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 \( t_{pos_i} \). The positions are in increasing order in the 
list, \( pos_i < pos_{i+1} \). 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 \( pos_i \) stored has already 
been reported.
2. It holds \( pos_j \) \(-\) \( pos_i \) \+ \( 1 \) \( \leq \) \( m \) \( + \) \( I \) \since no occurrence can appear later). 
3. 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 

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, suppose that we find that a word \( t_j \) that matches \( y = p_j \). We start by adding \((pos_{j+1}, mask_{j+1}) = (i, (j))\) at 
the end of the list of \( P \) and increment \( I \). In fact, it is possible that 
\( t_j \) has already matched some other word of \( P \), in which case 
\( i = pos_j \). 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 \( mask_j \), 
indicating that \( t_j \) also matches \( p_j \).

If the enlargement of the window does not make it 

3. We abandon the algorithm only when it holds 
\((pos_{j+1}, mask_{j+1}) = (i, (j))\) at 
the end of the list of \( P \) and increment \( I \). In fact, it is possible that 
\( t_j \) has already matched some other word of \( P \), in which case 
\( i = pos_j \). 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 \( mask_j \), 
indicating that \( t_j \) also matches \( p_j \).

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 \( pos_j \) \(-\) \( pos_i \) \+ \( 1 \) \( > \) \( m \) \( + \) \( I \), 
then we need to restore the invariants. We have 
now information on a text area that spans in more than \( m 
+ I \) words, which is enough to report at least maximal 
matches starting at \( pos_i \).

Checking for a maximal occurrence is done using the 
sequential word matching algorithm of Section 7.1: we 
initialize the matrix and feed it with the text words of the 
window. The precomputed answers to \( "p_j = t_j?" \) are 
precisely in the bit mask \( mask_j \), so we do not have to really look 
at the text. We abandon the algorithm only when it holds 
\( W_j > D \) for all \( j \) (since no occurrence can appear later). 
This eventually happens because our window is long 

Finally, we explain how we simulate the algorithm of 
Section 7.1 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 
\((pos_i, mask_i)\) \((pos_r, mask_r)\), 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 \( t_{pos_i} \). The positions are in increasing order in the 
list, \( pos_i < pos_{i+1} \). After we have processed text word \( t_i \), the 
following invariants hold on the list of pairs stored for every 
P:
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.

Figure 5 exemplifies the case of a pattern of $m = 4$ words, where we allow $D = 1$ deletion and $I = 2$ insertions. Full edged boxes represent text words that match some pattern word (we indicate which, and for simplicity assume that they match only one pattern word). Dashed edged boxes represent text words that match no pattern word. In fact these words (the majority for a given pattern) are never seen by the algorithm: each pattern is informed only of the presence of the full edged boxes, and infers the existence of the gaps. The example shows how the algorithm waits until it has a window long enough to ensure that a potential maximal occurrence is totally inside the window. Each time the length is exceeded, the first words in the window are removed after checking that they do not initiate a valid occurrence.

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

8 Performance

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

8.1 Average Case Analysis

Let us assume that we have a text of $N$ words, where we have to search for $M$ patterns of $m$ words each, allowing $I$
A total cost of this level is pessimistically bounded by \(O(NM)\), 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 \(O(M)\).

### 8.2 Experimental Results

We have tested our algorithm in a real case drawn from our official law application.

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 has linear time with \(N\), we use a fixed text of 1 Mb size. In this text, we searched for 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 measured user times, as these turn out to be very close to elapsed times.

Figure 6 shows the results. We show a plot with all the figures and also a zoomed version to appreciate the cheaper parts of the cost.

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 × 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 for 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 $5.69N^{1.04}$, with a relative error of 3%.

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

Hence, the total cost in our machine to process $M$ patterns on $N$ Mb of text is $N \times (2.05 + 5.69M^{0.8} + 0.01M)$, 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 for 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 Yianilos & Kanzelberger (1997), 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 0.4N M 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. In the current installation we are considering only letters as part of a word, mapping all them to lower case and without accents, discarding words of length 1 and 2, as well as three very common Portuguese articles, using dictionaries of given and family names of about 10,000 entries each, giving cost 1 to every operation on characters, setting a length-dependent error threshold (0 for lengths 3–4, 1 for lengths 5–6, 2 for lengths 7–8, 3 for lengths 9–12, and 4 for longer words), forcing that at least one word per pattern appears in every occurrence and permitting at most one spurious word inside occurrences. Overall, this constitutes a reasonable setup for our current application, so the figures we have shown belong to a realistic scenario.

From this setup, the part that most likely affects the search time is the tolerance when searching for pattern words. Of course it is of little use to relax the conditions up to a point where we get thousands of irrelevant matches. A careful balance between false positives and negatives has to be obtained. This is not an issue of how much time we pay but of how carefully we use the tools offered by the system to reflect real error probabilities and to be able to process the matches reported.

9 Concluding Remarks

Currently, in practice, Matchsimile has been tuned to have no misses at all, finding all possible names, because the legal impacts of missing a real name are high. This creates more false matches (at least 30% of the output), but this is bearable, as before was done manually.

Future plans with Matchsimile include incorporating new models for word matching where the order between words is not important (useful for company names in some cases, and for person names with some modifications). Another one is to rank the occurrences according to the total number of intra-word errors and inter-word errors, giving more weight to matching the first given names and the latter family names.

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 for every text word sequentially. This avoids repeating the same work for similar pattern prefixes, but similar text words are processed over and over. Using a technique known in computational biology to find all the approximate matches between two tries (Baeza-Yates & Gonnet, 1998), 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.

Acknowledgment

Supported by Millennium Nucleus Center for Web Research Grant P01-029-F, Mideplan, Chile.

References


