AGH University of Science and Technology

Faculty of Electrical Engineering, Automatics,
Computer Science and Electronics

Master Thesis

Approximate string matching algorithms in art media archives

Student:
Fernando Casanovas Martín
10th February 2009

Attendants:
Professor Mikołaj Leszczuk

AGH University of Science and Technology
Faculty of Electrical Engineering, Automatics, Computer Science and Electronics
al. Mickiewicza 30,
30-059 Kraków

Casanovas Martín, Fernando:
Approximate string matching algorithms in art media archives
Thesis, AGH University of Science and Technology, 2009.
"I hereby declare that the work presented in this thesis is solely my work and original, except where indicated by references to other authors."

ABSTRACT

The implementation of a distributed electronic art platform with a considerable amount of data available involves some technical challenges. Due to the lack of a common European platform for media art, the European Commission has decided to finance the implementation of GAMA (Gateway to Archives of Media Art), with the aim of establish a common platform for media art archives. One of the universities involved in the project is the AGH University of Science and Technology from Krakow, with responsibilities concerning to the architecture and IT solutions. The AGH team involved in the project is integrated by professors and students of the University, including the author of this paper. In the following chapters, an approach to the architecture of GAMA is presented, as well as the actual problem of integration of services and the solutions used to solve it in GAMA. Also the concept of harmonization and approximate string matching is presented, along with its applications and most used implementation solutions. For the GAMA project a system has been implemented to deal with the problem, this thesis describes the methodology used and the details of the implementation.

TABLE OF CONTENTS

1. INTRODUCTION

This chapter presents an introduction of what are the objectives of the GAMA project, and why there is a need for a "harmonization" process of the databases that are going to be used in it. This process uses approximate string matching concepts and algorithms. The State-of-the-Art section treats the actual situation of approximate string matching technologies and theories, and it also introduces the main technologies used in GAMA and basic concepts of Service-Oriented Architectures. The overall work methodology for developing the system that will allow the harmonization process to be done, and the novelty of it, is treated at the end of the chapter, as well as an explanation of how it is organised the rest of the thesis.

1.1 PROBLEM STATEMENT

GAMA (Gateway to Archives of Media Art) is a project financed by the European Commission, within the framework of the eContent*plus* programme (project number: ECP510029). The partners involved in the project are:

- Technologie Zentrum Informatik, Universität Bremen, (DE)
- Akademie der Bildenden Künste Wien, (AT)
- AGH University of Science and Technology, (PL)
- Argos – interdisciplinary centre for art and audio-visual media, (BE)
- Atos Origin s.a.e., (ES)
- C3 Center for Culture & Communication Foundation, (HU)
- CIANT International Centre for Art and New Technologies, (CZ)
- Stiftelsen Filform, (SE)
- Heure Exquise, (FR)
- Staatliche Hochschule für Gestaltung (DE)
- Hochschule für Künste Bremen, (DE)
- Zuercher Hochschule der Kuenste, (CH)
- Hogeschool vorr den Kunsten Utrecht, (NL)
- IN2 search interfaces development Ltd, (DE)
- Les INSTANTS VIDEO Numériques et poétiques, (FR)
- Ludwig Boltzmann Institut Medien Kunst Forschung, (AT)
- Netherlands Institut voor Mediakunst Montevideo/Time based Arts, (NL)
- SCCA Center for Contemporary Arts – Ljubljana, Videodokument, (SI)

- Universitat de Barcelona – Laboratori de Mitjans Interactius, (ES)

The aim of GAMA[1] is to establish a central platform to enable multilingual, facilitated and user-oriented access to a significant number of media art archives and their digitalized contents and, with time, become the main European online interface and portal for any person interested in media art.

Nowadays there is not comparable platform in existence in Europe networking digital archives, only Websites with links to other archives, but not common search interface or interoperable systems use. Nevertheless, there are other current projects and initiatives with similar aim to the GAMA project's. One example of a previous effort for providing unified access to electronic resources of, in this case, the main European National Libraries, is the EDLproject[2]. It was funded by the European Commission under the eContentplus Programme and integrates the bibliographic catalogues and digital collections of the National Libraries of all the EU countries. It is still in process of enlargement and a new prototype was launched in November 2008, with the name of Europeana[3], which gives access to approximately 2 million digital objects, including film material, photos, paintings, sounds, maps, manuscripts, newspapers and archival papers. The digital content comes from the already available in Europe's museums, libraries, archives, and audio-visual collections. These new platforms for accessing digital content are not only expected to attract new users but also more content providers, and act as incentive for further provision and creation of digitized data.

Being a search interface for such heterogeneous and diverse contents doesn't mean that some aspects regarding the precision of the results given for a determined query shouldn't be treated strictly. One of the problems expected concerns the different spellings and variations for a same name that appear in the different databases. This is due to language differences, typing mistakes, abbreviations, etc. The expected results for a user query should be all the ones regarding to the search made, no matter how it is stored in the different databases. If a user makes a query for a person's name, for example, the expected results are all the ones regarding to that person, no matter if the name is stored in the databases with misspellings errors, in different languages, with abbreviations, etc.

In order to solve this problem, a system that allows to "harmonize" the data stored in the databases needs to be implemented. The system would be based on approximate string matching algorithms and theories, and its aim would be to create a relation between names of persons and their spelling variations (considering possible spelling errors), and names of persons and art groups. The process made through and by this system has been called "harmonization".

1.2 STATE-OF-THE-ART

This section presents an introduction to approximate string matching algorithms and technologies, and a brief explanation of how these concepts are going to be used in the harmonization process. The main technologies used in GAMA and an introduction to Service-Oriented Architectures are also presented at the end of the section.

1.2.1 INTRODUCTION TO APPROXIMATE STRING MATCHING

Approximate string matching is a string matching that allows errors. The objective is to perform a string matching of a pattern in a text where one or both sources have suffered some kind of misspelling.

In its most general form, the problem consists in finding a text pattern inside another text, allowing a number of "errors" in the matches. For this purpose, the first thing needed is to create an error model, which defines how different two strings are. This idea of how different two strings are is called "distance" between strings, and the goal is to make that distance small when one of the strings is likely to be an erroneous variant of the other under the error model in use.

This introduction to approximate string matching will focus on online searching (as it is the case in GAMA). That is, when the text is not pre-processed.

MAIN APPLICATION AREAS

The largest application areas for approximate string matching are computational biology, signal processing and text retrieval.

In computational biology, DNA and protein sequences can be seen as long texts over specific alphabets. Searching specific sequences over those texts is a crucial operation for problems such as assembling the DNA chain from the pieces obtained by experiments, looking for given patterns in DNA chains, or determining how different two genetic sequences are. For these kinds of applications exact searching wasn't an appropriate technique, since the patterns searched hardly ever matched the text exactly: experimental measures present different kinds of errors, and even the correct chains may have small differences. Also, it was needed to find similar DNA chains, and establish how different two sequences were, in order to reconstruct the tree of evolution[4]. All these problems required a concept of "similarity", and an algorithm to compute it. This was the start of "search allowing errors". The "distance" between two sequences was defined as the minimum number of operations to transform one into the other. The operations were given a "cost", and the aim was then to

minimize the total cost. Nowadays there are other more complex problems, such as structure matching or searching for unknown patterns.

The second major area of application for approximate string matching is signal processing and, more specifically, speech recognition. There, the problem is to determine, given an audio signal, a textual message which is being transmitted. Even a simplified problem becomes complex in this environment, for example, discerning a word from a small set of alternatives: the signal may be compressed in time, parts of the speech may not be pronounced, etc. A perfect match is practically impossible. Another problem is error correction. To ensure a correct transmission over a physical channel, it is necessary to be able to recover the correct message after a possible modification (error) introduced during the transmission. The probability if this kind of errors is present is obtained from the signal processing theory and used to assign a cost to them. In case of errors, it is possible not to know even what to search for, and just search for a text that is correct and closest to the received message. This area has generated the most important measure for similarity used in approximate searching: the Levenshtein distance[6] (also called edit distance)[5]. The rapidly evolution of multimedia database demands the ability to search by content in image, audio and video data, which are also potential applications for approximate searching.

The last major application for approximate string matching is text retrieval: the problem of correcting misspelled words in written text is rather old, and approximate string matching has always been one of the most popular tools to deal with it. It is also wide extended to use it to deal with problems of information retrieval. That is, finding the relevant information in a large text. However, classical string matching is not normally enough, because the text collections are becoming larger (e.g. the Web text has surpassed 6 terabytes)[6], more heterogeneous (for example, different languages), and with more errors. A word that is entered incorrectly in a database may be impossible to control and retrieve. Also, the pattern itself may have errors. This would be the case, for example, of a cross-lingual scenario where a foreign name is incorrectly spelled, or in old texts that use outdated versions of the language.

Text collections digitalized via optical character recognition contain a significant percentage of errors. The same problem appears with typing and spelling errors. Approximate string matching applications allow in these texts to make queries that survive syntactic or spelling mistakes: the user supplies a phrase to search, and the system search the text positions where the phrase appears with a limited number of *word* insertions, deletions and substitutions.

The harmonization system implemented for GAMA project can be included in this last application area. However, the algorithm used has been specially designed to the characteristics of the text stored in the databases and the result expected (the process of approximate string matching is done between strings of very short length, and most of them have the structure of persons' names), it doesn't consist in a classic text retrieval algorithm.

CONCEPTS AND DEFINITIONS

This section introduces basic concepts of approximate string matching algorithms, some of them used for the implementation of the harmonization system.

Approximate string matching:

As introduced before, the problem of approximate string matching is defined as that of finding the text positions that match a pattern up to k errors. For the following definitions, x, y, z, v, w will be used to represent arbitrary strings, and $a, b, c \ldots$ to represent letters. Writing a sequence of strings and/or letters will represent their concatenation. For any string s, its length will be denoted as $|s|$, and the i_{th} character of s will be denoted as s_i (for an integer $i \in \{ 1..|s| \}$.

Considering:

- Σ a finite alphabet of size $|\Sigma| = \sigma$.
- $T \in \Sigma$ a text of length $n = |T|$.
- $P \in \Sigma$ a pattern of length $m = |P|$.
- $k \in \Re$ the maximum errors allowed.
- $d : \sum \times \sum \to \Re$ a function that defines the distance between 2 strings.

The problem is: given T, P, K and $d(.)$, return the set of all the text positions j such that there exists i such that $d(P, T_{i..j}) \leq k$.

Distance functions:

The distance $d(x, y)$ between two strings x and y is the minimal cost of a sequence of operations to transform x into y. The cost of a sequence of operations is the sum of the costs of the individual operations. The operations are defined as a set of rules of the form $d(x, y) = t$, where x and y are two strings and t is a nonnegative real number. Once the operation has converted a substring x into y, no further operations can be done on x.

For each operation of the form $d(x, y)$ exists the respective operation $d(y, x)$ at the same cost (the distance is symmetric). The distance $d(x, y) \geq 0$ for two different strings, $d(x, x) = 0$, and $d(x, y) \leq d(x, z) + d(z, y)$. Then, if the distance is symmetric, the space of strings forms a metric space.

In most applications, the set of operations is restricted to:

- Insertion: inserting a letter in the string, e.g.
  Joan ⟶ Jo**h**an
- Deletion: deleting a letter in the string, e.g.
  Jo**h**an ⟶ Joan
- Substitution or replacement, e.g.
  Joh**a**n ⟶ Joh**e**n
- Transposition: swap adjacent letters, e.g.
  Jo**ha**n ⟶ Jo**ah**n

The most commonly used distance functions are the following[6]:

- Levenshtein distance: The Levenshtein algorithm allows insertions, deletions and substitutions. In the simplified definition, all operations costs 1, but is a common practice to assign a cost of 2 to the substitutions. The result is, then, the minimal number of insertions, deletions and substitutions to make two strings equal (Fig. 1). The distance is symmetric.

		J	o	h	e	n
	0	1	2	3	4	5
J	1	0	1	2	3	4
O	2	1	0	1	2	3
H	3	2	1	…	…	2
A	4	3	…	…	…	…
n	5	4	…	…	…	…

Fig. 1: Calculation matrix of the Levenshtein distance. Described in detail in table 1.

- Hamming distance: Allows only substitutions, which cost 1 (Fig.2). So the Hamming distance between two strings of the same length is the number of positions for which the corresponding symbols are different. If the Hamming or edit distance are used, then the problem only makes sense for $0 < k - m$, since if m operations are made, the pattern can be matched at any text positions using m substitutions. The distance is symmetric.

JOHAN ⟶ JOHEN ⟹ $edist_{Ham} = 1$

11001101 ⟶ 1001111 ⟹ $edist_{Ham} = 1$

JUHAN ⟶ JOHEN ⟹ $edist_{Ham} = 2$

Fig. 2: Examples Hamming distance

- Episode distance: Allows only insertions, which cost 1. It is also called "episode matching" since it models the case where a sequence of events is sought, where all of them must occur within a short period (Fig. 3). This distance is not symmetric and it may not be possible to convert x into y. Then, $d(x, y)$ is either $|y| - |x|$ or ∞.

JOHAN ⟶ J OHANN ⟹ $edist_{Epi} = 1$

Fig. 3: Example Episode distance

- Longest common subsequence (LCS): Allows only insertions and deletions, which cost 1. It calculates the length of the longest pairing of characters that can be established between both strings, so that the order of the letters is respected. The distance is the number of unpaired characters, and it is symmetric (Fig. 4).

AGCGA ⟶ CAGATAGAG ⟹ $LCS = AGGA$

Fig. 4: LCS between 2 strings

In all the cases, except the episode distance, the changes can be made over x or y. Insertions on x is the same as deletion on y and vice versa, and substitutions can be made in any of the two strings to match the other.

Note that transpositions are of special interest in case of typing errors. However, not many algorithms deal with them. They are just simulated as an insertion plus a deletion, and therefore assigned a higher cost than a single operation. The algorithms above can also be extended to assign different costs of operations, including the case of not allowing some operations.

There exist many other algorithms with important improvements in their theoretical complexity, but they are very slow in practice and they are only used in very specific scenarios, which do not appear in most of applications. The parameters for the problem where can be considered practical to use the algorithms explained in this section are[6]:

- The pattern length can be as short as 5 letters and long as a few hundred.
- The number of errors allowed k satisfies that k/m is a relatively low value. Reasonable values for matching range from $1/m$ to $1/2$.
- The alphabet size can be as low as 4 letters and high as 256 letters
- The text length can be as short as a few thousand letters and as long as megabytes or gigabytes.

The Levenshtein algorithm:

The algorithm finds the cheapest way to transform one string into another, and it is the base for many approximate string matching algorithms. Transformations are made making one of these operations: insertion, deletion, substitution (extended versions include also transposition). Each operation is assigned a cost, and the result is the total cost of transform one string into another.

This an example of how the algorithm works, and illustrates how it looks for all of the different ways for operations to transform one string into another. As example strings, *Johan* and *Johen* will be used. The algorithm starts with the upper left-hand corner of a two-dimensional array indexed in rows by the letters of the source word, and in columns by the letters of the target word. It fills out the rest of the array while finding all the distances between each initial prefix of the source on one hand and each initial prefix of the target on the other. Each $[i, j]$ cell represents the minimal distance between the first i letters of the source word and the first j letters of the target word.

It is only possible to fill in the value of a cell only in case the values of all its neighbours upward and to the left have been filled in (Fig.5):

		J	o	h	e	n
	0	1	2	3	4	5
J	1					
o	2					
h	3					
a	4					
n	5					

Fig. 5: Levenshtein distance calculation matrix

- Top row is 0, 1, 2,… (cost of insertions).
- Left column is 1, 2,… (cost of deletions).
- Then, starting at upper-left, the process for filling a cell (see Tab.1).

Diagonal cell	Above cell
Left cell	$\min(above + delete, diag + replace, left + insert)$

Tab. 1: Cell filling in Levenshtein distance calculation matrix

When cells up and to the left are filled, the adjoining cell can be filled. For filling it, the process is just to calculate all three values that may appear there and insert the minimum one. The values that may appear are: the result of deleting the letter indexing that column added to the value directly above, the result of inserting the letter in the indexing row added to the value to the left, and the result of replacing the column letter by the row letter added to the value diagonally upward and to the left.

This way, the entire array can be completed. At the end, each lower right corner of the table contains the Levenshtein distance, which is the least costly set of operations that map one string into the other (Fig. 6).

		J	o	h	e	n
	0	1	2	3	4	5
J	1	0	1	2	3	4
o	2	1	0	1	2	3
h	3	2	1	0	1	2
a	4	3	2	1	2	3
n	5	4	3	2	3	2

Fig.6: Calculation matrix of the Levenshtein distance

The algorithm can be refined, for example varying the costs of the operations. One interesting aspect is the possibility to change the cost of transformations by making them sensitive to the phonetic similarity of elements replacing one another: for example, making the replacing of “d” by “t” less costly than replacing “d” by “u”.

Phonetic algorithms:

Phonetic algorithms index names by sound, as pronounced, basically in English, although there are developments for many other languages. The goal is to encode with equal representation names with the same pronunciation, so that minor differences in spelling don’t affect the matching.

For the harmonization process, in order to improve the results of applying an algorithm based in the edit distance, a phonetic algorithm has been used. There were three possible candidates: Soundex, Metaphone, and Double-Metaphone[7]. For the three of them there are free available implementations in PHP.

Soundex is the most widely known of these algorithms, and is the basis for many modern phonetic algorithms. It is designed primarily for use with English names. The algorithm converts each name to a four-character code, which can be used to identify equivalent names, and it is structured as follows:

1. Retain the first letter of the name, and drop all occurrences of a, e, h, i, o, u, w, y in other positions.
2. Assign the following numbers to the remaining letters after the first:
   b, f, p, v ⟶ 1
   l ⟶ 4
   c, g, j, k, q, s, x, z ⟶ 2
   m, n ⟶ 5
   d, t ⟶ 3
   r ⟶ 6
3. If two or more letters with the same code were adjacent in the original name (before step 1), omit all but the first.
4. If there are less than 3 digits, add trailing zeros to convert to the form ‘letter, digit, digit, digit’. If there are more than 3 digits, drop the rightmost ones.

Metaphone was developed by Lawrence Philips[8] for matching words that sound alike and it is based on the rules of English pronunciation. Given a string, it generates a code of variable length that represents its pronunciation. The algorithm ignores vowels after the first letter and reduces the remaining alphabet to sixteen consonant sounds, although vowels are retained when they are the first letter. Duplicate letters are not added to code. Zero is used to represent the “th” sound, since it resembles to the Greek “theta” when it has a line through it, and “X” is used for the “sh” sound. The sixteen symbols used for encoding consonants sounds are: B X S K J T F H L M N P R Ø W Y.

The rules for the encoding are the following (see Tab. 2):

LETTER	CODE	COMMENTS
B	B	unless at the end of a word after “m” as in “dumb”
C	X	X (sh) if “-cia-” or “-ch-”
	S	S is “-ci-”, “-ce-”, or “-cy-”
		Silent if “-sci-”, “-sce-”, or “-scy-”
	K	Otherwise, including “-sch-”
D	J	If in “-dge-”, “-dgy-”, “-dgi”
	T	Otherwise
F	F	
G		Silent if in “-gh-” and not at end or before a vowel
		In “-gn” or “-gned”
		In “-dge-”, etc., as in above rule
	J	If before “i”, or “e”, or “y”, if not double “gg”
	K	Otherwise
H		Silent if after vowel and no vowel follows
	H	Otherwise
J	J	
K		Silent if after “c”
	K	Otherwise
L	L	
M	M	
N	N	
P	F	If before “h”
	P	Otherwise
Q	K	
R	R	
S	X	(sh) if before “h” or in “-sio-” or “-sia-”
	S	Otherwise
T	X	(sh) if “-tia-” or “-tio-”
		Silent if in “tch-”
	T	Otherwise
V	F	
		Silent if not followed by a vowel
	W	If followed by a vowel
X	KS	
Y		Silent if not followed by a vowel
	Y	If followed by a vowel
Z	S	

Tab.2: Metaphone encoding rules

Exceptions:

- Initial “kn-“, “gn-“, “pm-“, “ae-“, “wr-“ drop the first letter.
- Initial “x-“ is changed to s.
- Initial “wh-“ is changed to w.

These encoding rules were designed to make the maximum phonetic matches without entering in contradictory cases. The algorithm could be much more complex, but the number of exceptions will grow exponentially. For example, the “sh” sound could be assigned in a word like “casual” (“s” between two vowels), but then it would create wrong encodings for the sound in words like “persuade. In comparison to the Soundex code (in which Metaphone is based), the Metaphone technique is able to distinguish names such as Bonner and Baymore (BNR and BMR), Smith and Saneed (SMØ and SNT), or Van Hoesen and Vincenzo (VNHSN and VNSNS), which the original method gave the same code to.

The Double Metaphone phonetic algorithm was written by Lawrence Philips[8] and is the second generation of the Metaphone algorithm. It is called “double” because two codes can be produced from a single string (primary and secondary code). This is useful for some ambiguous cases as well as for multiple variants of surnames with common ancestry. For example, “Smith” and “Schmidt”, provide the same primary key if encoded with Double Metaphone, but different secondary keys. It uses a much more complex ruleset for coding than the original algorithm, as it tries to account for a large number of irregularities (and not only in English language).

1.2.2 SERVICE-ORIENTED ARCHITECTURES AND DISTRIBUTED ARCHITECTURES

There have been different architecture proposals for GAMA. As observed in following chapters, GAMA platform will present characteristics of service-oriented architectures and distributed architectures.

Service-oriented architectures are a kind of software architecture that is designed to establish a dynamically organized environment, based on networked services that are interoperable and composable. In a SOA, services are separated from their implementation, using the concept of an interface. This interface controls how the interaction between the parties will be taken place. SOAs offer a number of advantages for composing federations of services among loosely connected and disparate organizations, while allowing each one to conserve its autonomy in terms of how it builds and designs services as well as their ownership[9].

SOAs are characterized by the following properties:

- **Diversely Owned** – SOAs may be composed of services which are owned and operated by outside organizations. Diverse ownership implies that the published service interface will be treated as a black-box from the standpoint of the programmers since they cannot penetrate the interface and modify code and behaviour behind it.

- **Locationally Transparent** – SOAs are constructed in such a way that the overall system is unaware of the location of various services.

- **Interoperable** – Standards guarantee that different organizations can use each other's services.

- **Composable** – In SOAs, applications are created by composing well tested, pre-existing services from various providers.

- **Self-healing** – The capacity to rediscover and bind to working services when services fail is critical, in a system where applications are designed by composing dynamically discovered components that are owned by different organizations.

Distributed architectures are based on a notion of complex endpoints and transport carrying simple data. Early distributed system designers were creating a programming style that used hardware as a model for software development. Hardware is characterized by complex components, chips that hook together in simple ways. The data that is transported between chips is mere bits and even when aggregated into busses rarely looks anymore complicated than an array of bits. This model influenced the designers of early distributed systems to create complex, rigid interfaces that carried simple, serializable data.

For choosing an appropriate architecture for GAMA platform one of the most relevant aspects taken in account has been the problem of integration of services.

1.2.2.1 THE PROBLEM OF INTEGRATION: HISTORY AND ACTUAL SOLUTIONS

One of the technical challenges in GAMA is the integration of different services in an environment where there are different elements and applications that need to communicate, and also different protocols involved. This interoperability between different systems and programming languages needed in GAMA can prove more problematic than it seems. GAMA is a case of study of problems that nowadays affects many enterprises and organisations: enterprises need to significantly improve their flow of data and information, and users expect everything to be connected and be able to access to different services remotely. As an example, let's think of a user that connects to a flying company Web page. The user will expect to be able to check the availability of flights, buy tickets, change reservations, etc.

However, this integration of services cannot be done unless the technology used by the organisation is also integrated. Historically this have been solved using systems that operate in isolation from other systems, and using solutions like point-to-point connections between applications, which have many disadvantages: exponential growing of connections when new modules are added (let's consider, for example, how inefficient this could be in a situation

when the number of individual systems that have to be integrated – this is of the order of hundreds), no reusable solutions, highly dependent on applications, etc. Another crucial problem with integration is that one system can have been developed on a technology platform that is incompatible with the technology platforms used by other systems.

The most popular solutions that try to solve the problem of integration are[10]:

- CORBA (Common Object Request Broker Architecture): provided an open standard for distributed systems to communicate; however CORBA projects require a high development effort and standardization proved problematic.

- ERP (Enterprise Resource Planning): involve replacing systems with a suite of interconnected modules from a single vendor. Usually no single ERP can address all the requirements of a system and sometimes it increases, rather than removing it, the need for integration.

- Web services: collection of protocols and standards used for exchange data between applications. The organizations OASIS and W3C are the responsible committees for the architecture and regulating of Web services. Enable systems to communicate over the Internet or an intranet, but they present some barriers: the standards are relatively new and continue to evolve rapidly, they are not usually suitable for high-volume transaction processing, and though it may be useful to develop new systems using Web services, existing systems may need to be redesigned to conform to a Web Services model.

- EAI (Enterprise Application Integration): EAI is defined as the use of software and systems architecture principles to integrate a group of applications. EAI tools are marketed by companies like Microsoft, IBM, TIBCO, Seebeyond, etc., and they evolved from the message-oriented middleware tools that became popular as a meaning to provide high-volume, reliable communications between systems. In general, they have three components: an integration broker that serves as a hub for intersystem communication and performs some functions like multi-format translation, transaction management, monitoring and auditing. A set of adapters that enables different systems to interface with the integration broker, and an underlying communication infrastructure, such a reliable high-speed network, which enables systems to communicate with each other using a variety of different protocols.

- SOA (Service-Oriented Architecture): In SOA, systems present their functionality as a set of services (usually as a set of Web services), without mattering which technology platform the systems use or in which development language they are implemented, as services are presented in a way that other systems can understand. In a SOA the integration is

not a problem as far as the systems can provide their functionality through a set of defined services and conform to the SOA model, so it presents the same main barrier as the Web services: it's a good solution for implementing new platforms, but it can be problematic to adopt a SOA model in an existing system.

1.2.2.2 ESB: THE IMPLEMENTATION BACKBONE FOR A SOA

An Enterprise Service Bus (ESB) is a standards-based integration platform that combines messaging, Web services, data transformation, and routing, to reliably connect and coordinate the interaction of a significant number of diverse applications.

It is based in EAI and SOA patterns, and solves most of the problems that may appear when implementing a SOA. In a SOA, built using an ESB, the bus is the piece of software that lies between the applications and enables the communication among them, so all the communications take place via the ESB. For this purpose, it uses en enterprise message model, that defines a set of messages that the ESB is capable of transmit and receive, routing them to the appropriate applications. Usually the applications are not built according to this message model, so the ESB has to make the needed transformations into a format that it can work with.

The key characteristics of an ESB that, together, solve the requirements of a SOA, are:

- Distributed messaging: this guarantees the delivery of the messages even in case of anomalies in the network.

- Transparency of the locations: when a client service invokes the service provider, it only has to know that the service exists; the client doesn't need to know where the service is being executed. That provides a certain level of virtualization of the services, so if a service provider fails or changes its location, it is not necessary to notify the change to all the individual service clients.

- Multiprotocol support

- Quality of Service (QoS): in enterprise's applications, the QoS refers, mainly, to the reliability. An ESB can provide a service of high reliability, and using methods that satisfy the standards (for example, being compatible with the WS-ReliableMessaging specifications).

- Transformation: the main task of an ESB is to direct the messages from one service to the next one. However, there are cases when the data format of one service doesn't satisfy the requirements of the next service.

For this reason, the ESB must be capable of transform the data from one format to the other.

- Content-based routing: this is how the transparency of locations is achieved. When a service call enters the ESB, it routes the answer to the proper service provider, without the needing for the client service to know its location.

1.2.3 MAIN TECHNOLOGIES USED IN GAMA

Different programming languages and technologies have been used for the implementation of GAMA. Most of the code has been written, mainly, in PHP, Java and Perl. This section presents a brief introduction to these 3 technologies.

PHP: HYPERTEXT PREPROCESSOR

PHP (Hypertext Pre-processor) is a computer scripting language, originally designed for producing dynamic Web pages. It can be used in standalone graphical applications or from a command line interface, but is mainly used in server-side scripting. It is specially suited for Web development and can be embedded into HTML. In most cases it runs on a Web server, where PHP code is taken as its input and Web pages are created as output. PHP acts primarily as a filter, taking input from a stream or a file containing text and/or PHP instructions, and outputs another stream of data (most commonly the output will be HTML).

PHP can be deployed on most Web servers and operating systems, and can be used with many relational database management systems. It is available free of charge, and the PHP Group (http://php.net/) provides the complete source code for users to build, extend and customize their own use.

JAVA

Java is a programming language initially developed by Sun Microsystems, and most of the Java technologies are available as free software under the GNU General Public License. The language derives from C and C++, but it has a simpler object model and fewer low-level facilities. Java applications are typically compiled to byte-code that can run on any Java virtual machine, regardless of computer architecture, and it uses object-oriented programming methodology.

One of the most important characteristics of Java is the platform independence: programs written in Java must run similarly on any supported operating-system platform. It should be possible to write a program once, compile it once, and run it anywhere. This is performed by most Java compilers by compiling the Java language code halfway (to Java byte-code, simplified

machine instructions specific to Java platform). The code then is run on a virtual machine, a program written in native code on host hardware that interprets and executes generic java byte-code.

PERL

Perl is an interpreted programming language optimized for scanning text files, extracting information from those text files, and printing reports based on it. It's also useful for many system management tasks. The language is meant to be practical (easy to use, complete efficient) rather than beautiful (elegant, tiny, minimal), and its one of the most popular dynamic language to write Web applications.

Its major features include support for multiple programming paradigms (procedural, object-oriented, and functional styles), reference counting memory management, built-in support for text processing, and a large collection of third-party modules.

The structure of Perl derives mainly from C. Perl is procedural in nature, with expressions, variables, assignment statements, brace-delimited code blocks, subroutines and control structures. However, Object Oriented Programming can be employed in Perl as well.

1.3 METHODOLOGY

The first thing to be done was the design of an intuitive interface that would allow the editors of the databases to use the system as easy as possible. Once the interface was ready a test database was created, with the contents of one of the future databases that will use the system for the harmonization process.

Second step was implementing an algorithm, adapted to the goal aimed by the system, which made the harmonization process. For this purpose, the overall methodology was the following:

- Test the algorithm with the database specially created for testing the system.
- Study the results and improve the algorithm.
- Consult editors and other users that are not implementing the system but are involved in the GAMA project and will have to work in the harmonization process.
- Study their comments and proposals and improve the algorithm.
- Go to the testing step again.

The testing was made two ways: randomly, and looking for specific results. This was done due to the goal aimed by the system. Its use will be to search for names that refer to the same person or art work, so every testing phase some specific names with some misspellings or desired characteristics were tested to study how the new algorithm worked (that included names with typical typing mistakes, e.g. transposition of letters, abbreviations, etc.). Also, determined expected situations were simulated.

Consulting users that are not directly implementing the system but will have to use it allowed not only to improve the algorithm, but also the interface, as they can provide valuable comments about the usability of it.

Firstly, only one algorithm was implemented. It was based on approximate string matching algorithms and simply made a fuzzy search of the query through the database. Once the results of the first testing were obtained, the adaptation of the system for the specific characteristics of the process could start. To improve the results another algorithm based in phonetic features was implemented, to work in combination with the original one. The methodology for improving the algorithms was the same for both of them.

1.4 NOVELTY

The search algorithm considers not only the usual measures (e.g. Levenshtein distance) used in most text retrieval applications, it is specially adapted to the functionality expected for the harmonization process.

That is, it considers some grammar and spelling aspects that a common fuzzy search does not. This is specially important in the case of queries for human names, where the length of the strings is limited, and also each letter has a different value if the goal is to match with another human name. For example, the order of words and some punctuations symbols cannot be treated as in an ordinary approximate string comparison (e.g. “David Smith” and “Smith, David”, are strings considerably different but it is highly probable that they refer to the same person). The algorithm has been also improved to make an intelligent processing of abbreviations and titles (e.g. “David Smith” will match with “D. Smith” and “Dr. David S.”).

The result once the databases have been harmonizated will be an interface that once a user provides a query, will give results that refer to the query no matter about the language or spelling of the data in the different databases, but without the user having to choose between the results the ones that really refer to the search made (this is the case, for example, of Google). All the results provided will refer to the person or artwork that the user is searching for.

1.5 ORGANISATION OF THE REST OF THE THESIS

In chapter 2 the harmonization process is described in detail. Section 2.1 describes the methodology details, concerning the organization of the work, the algorithms and the interface implemented. Section 2.2 treats the implementation details of the interface and both approximate string matching algorithms developed for the harmonization process.

Chapter 3 presents the results and the first conclusions of testing the algorithms. Section 3.1 contains the results of the test with a subsample data set of short strings and section 3.2 the results with a subsample data set with long strings (more than 10 letters). The conclusions achieved after the implementation and testing of the system are presented in chapter 4, concerning not only the algorithms implemented but also all approximate string matching algorithms and applications. To end the thesis, chapter 5 presents the acknowledgements.

2. HARMONIZATION

Due to the different databases (with information in different languages) and significant amount of data that the GAMA platform has to deal with, it has become a necessity to somehow "harmonize" the data stored. That means, that a query for an artist name, for example, should give the results regarding this artist no matter the different spellings of the same name in the different databases. This process has been called "harmonization", and this chapter will present the solution implemented by the author of this paper to deal with it.

2.1 METHODOLOGY DETAILS

The solution implemented for solving the problem has been continuously improved until the implementation of the final algorithm has been achieved. This evolution is treated in this section, explaining the problems encountered during the development and the decisions taken to solve them.

2.1.2 INTRODUCTION

Taking a look at the problem, it's obvious that searches based on exact string matching would leave out of the results many of the correct ones. The first approach, then, is to base the queries on an approximate string matching. This, however, would give wrong results mixed with the correct ones. Some undesired results would have the same percentage of error, comparing with the pattern searched, as correct ones (in the case of the correct ones, the error would be due to misspellings, differences in languages, etc.).The goal of the platform is to provide directly the correct results, without the user having to choose between all the ones provided by an approximate string matching, the results that really correspond to the search made.

A problem that looked simple at the beginning, becomes more complex at this point. An implementation of an algorithm based on approximate string matching theories that assures the correctly of the results is not possible, because it is based on allowing a certain percentage of error in the matches, and so, it is based in the probability of error within the strings searched. This allows to develop an algorithm that provides correct results with higher probability than undesired results, but not 100% correct results.

As explained in previous chapters, the error allowed in the matching can be modified according to the goal aimed. A calculation of an optimal error allowed for having the highest number of correct results based on probability theories could be made, but this would still not be the functionality expected for the platform. Even if it would ensure the highest number of correct matches, there would be still wrong results mixed with the good ones, and correct ones that would not appear. This would mean, for example, that in some cases one user searching for a specific person name, artist group, or other, would not be able to find it even if it was stored in the databases.

2.1.3 HARMONIZATION OF DATABASES

The harmonization process will work with the data before any action of the user, and assign the same ID for names that refer to the same person. This would involve that the user queries will be able to be based on exact string matching, as once is found a result that matches the query, the process of finding matches that can refer to the same query no matter spelling problems, will be already done.

For this purpose, a system has been implemented that allows the partners to make this process as simple as possible. It involves to choose manually the names that refer to the same person, but applying before a filter based in approximate matching algorithms, that has been specially implemented to provide a small number of results (compared to the thousands that are stored in the databases), but ensuring that no correct matches will be left out by the filtering.

The algorithm is mainly based in the Levenshtein distance, but applying some modifications as Levenshtein algorithm is more oriented to make searches of a pattern in biggest texts and tend to leave out correct results if the error allowed is small (this is not problematic in most common uses of this algorithm, but in GAMA it is not an alternative as it would make “invisible” for some user queries correct matches), and increases too much the number of possible matches if the error allowed increases (then the process of harmonization would require too much effort as the names have to be chosen manually). The implementation used for the harmonization process is one written in PHP, that is available in the free and open source officially documented libraries for the programming language.

Levenshtein was chosen because the concept of “edit distance” was extremely useful for the approximate string matching algorithm that was needed for the harmonization process. For finding inside a database different spelled names of persons or art works that refer to the same but are differently typed, edit distance is a reliable measure, and Levenshtein allowed not only to calculate this distance between strings but also to easily make modifications in the algorithm in order to adapt the solution to the problem.

Besides the use of the Levenshtein algorithm it has been also used a phonetic algorithm (Metaphone) that encodes with the same representation names with same pronunciation, even if the spelling is different. Due to the characteristics of the harmonization process, this permits to reduce the error allowed in Levenshtein and so the number of possible matches given by the interface.

As a phonetic algorithm, Metaphone was chosen among 3 candidates: Soundex, Metaphone, and Double-Metaphone (the three most extended used phonetic algorithms). The one chosen was Metaphone. The accuracy of Metaphone is better than Soundex, but worse than Double-Metaphone[7]. However, the speed of the algorithms also decreases with the improvement in accuracy: on the computer used for the testing, for calculating the Double-Metaphone code representation of a given string 1000 times and set a variable to that value, it took 4 seconds. Given the same exact code, the built in Metaphone function took 0.03 seconds, and Soundex took 0.015 seconds.

Double-Metaphone was discarded, as for a user query of two words, over a database that contains 2000 names of persons or art works, it will represent an addition of more than 10 seconds in the loading time of the interface that shows the results. The first approach was to use Soundex, but after implementing an algorithm based in it, the results proved not good enough for the quality expected.

Then Metaphone was tried. The speed was good enough but, as in Soundex, the accuracy was still low (although higher than with Soundex). The solution implemented was to use it in combination with Levenshtein (faster than the phonetic algorithms): Metaphone generates codes of variable length for each string that represent its pronunciation. A way that proved effectively was taking Levenshtein distances between two Metaphone codes, and then taking this a percentage of the length of the original Metaphone code. This way, a percentage error can be defined (e.g. 20%) and accept only matches that are closer than that.

2.1.4 THE INTERFACE

The main page of the interface consists of a table with names where the user has to select with which one he is going to work. The table shows by default all the names in the database, but 3 filters can be applied to reduce the number of names showed: show all edited, all none edited, or show the names making a filter by keywords/letters (Fig. 7).

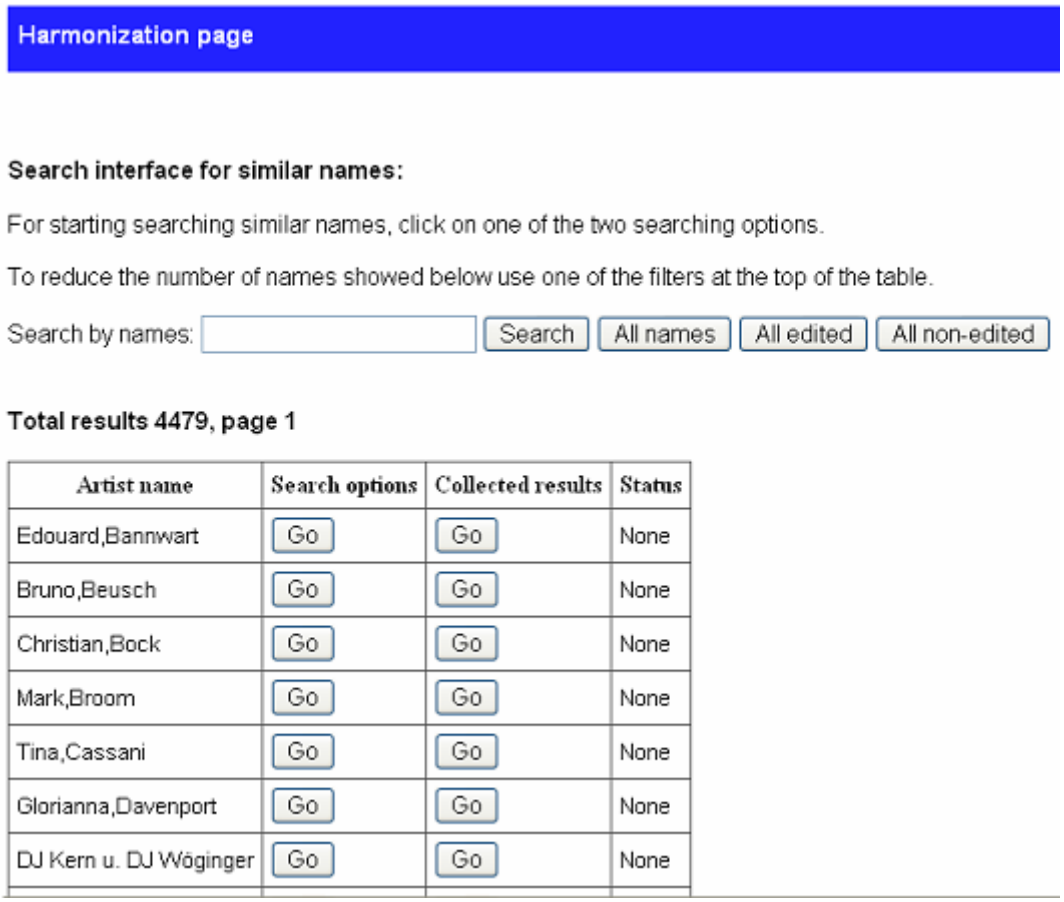

Fig. 7: Main page screen shot

Once the name desired is found between all the possible ones, the next step is to choose between 2 working options: find possible names in the database that are related to the same person, or work with the already collected results for that name.

If the user selects the option of searching the database for possibly related names, he will be redirect to another page where two searching options are available:

- Plain search: will make a filtering based in the Levenshtein algorithm, combined with some other improvements to provide an approximate string matching.

- Search with features: same as the plain search, but taking in consideration phonetic features of the pattern and the target text.

Clicking one of the search options, all the possible matches will appear, ordered by the similarity estimation calculated by the algorithm selected. Then the names that are related to the search will have to be selected manually, choosing the proper type of relation (main form, correct spelling variation, incorrect spelling variation, related person/group, pseudonym, different

person/group). The process will finish when the user clicks the “Confirm changes” button, which will store all the changes made in the database (Fig. 8).

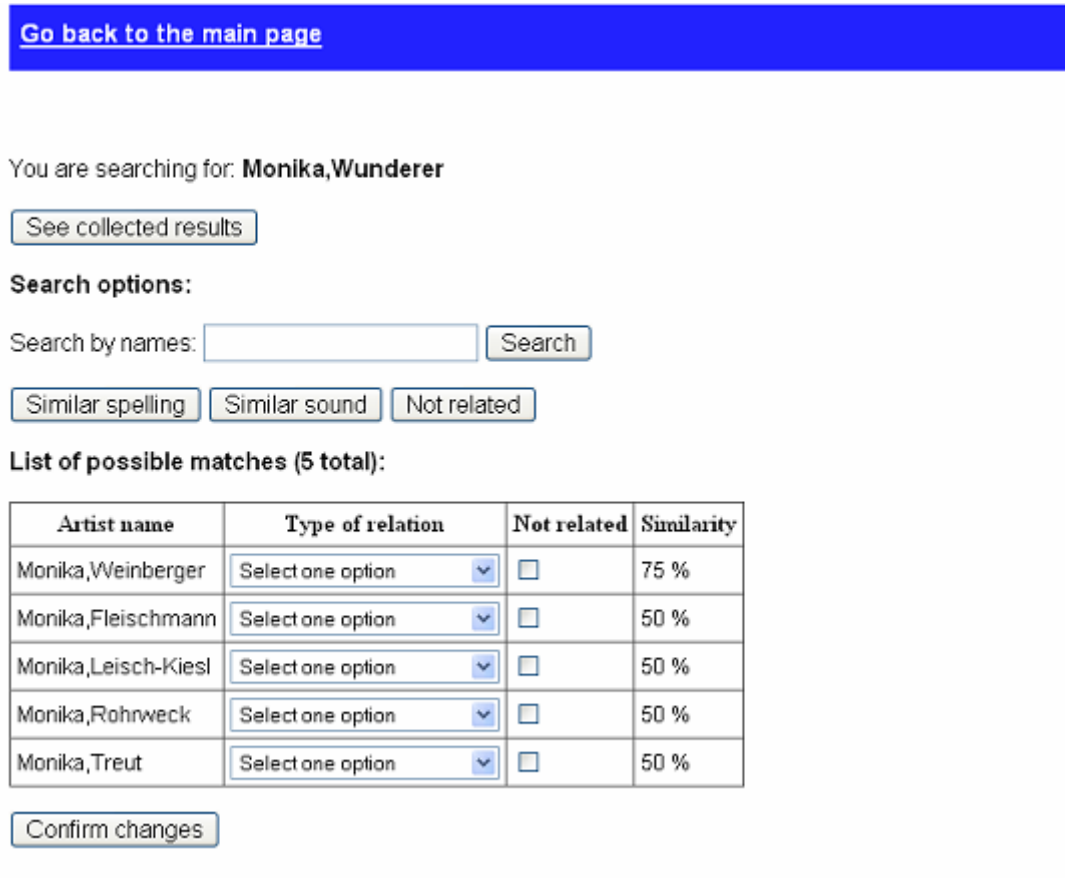

Fig. 8: Results page screenshot

The second working option is modifying already matched results. This will not allow to make new relations to the search, but to change the existing ones. Also it will be possible to provide extra information about the progress of the harmonization process, like the status of the work with that database entry (ok, unsure, revised by expert), or the possibility of adding comments in a text field (Fig. 9).

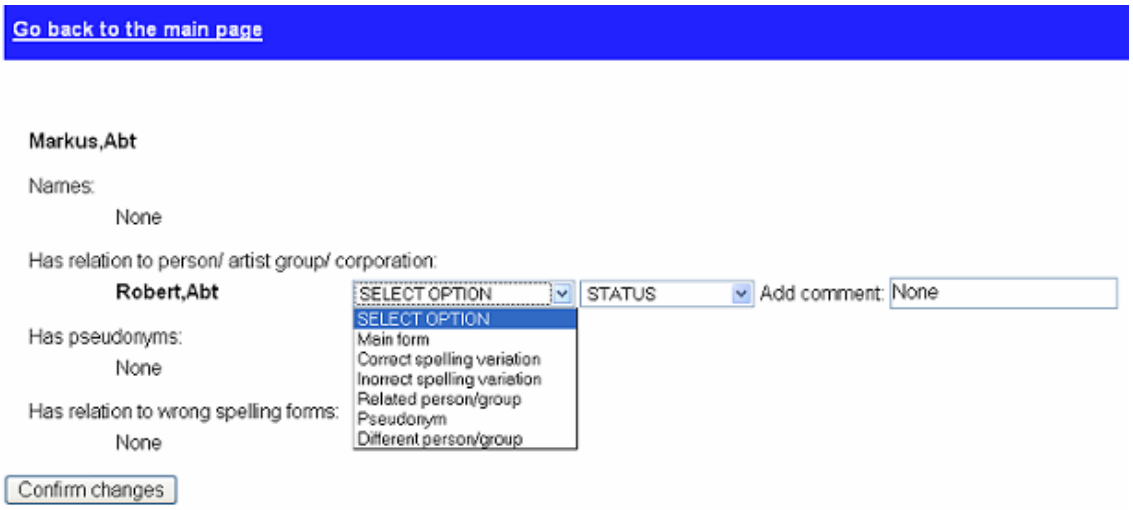

Fig. 9: Collected results page screenshot

2.1.5 PLAIN SEARCH ALGORITHM

The first approach for the plain search was using the Levenshtein algorithm to directly compare the pattern searched with all the other names in the database, assigning to all the operations a cost of 1. Then, normalize each result dividing the total number of letters minus the total cost of the transformation, by the number of letters in the shortest text (this provides an estimation of the similarity between the two strings). The result of the Levenshtein algorithm could not be applied directly as it consists of an integer value and some other measure must be required to make the filtering, as a cost of 5 for making the transformation can be considered a possible match in, for example, strings of 15 letters, but it is obviously a bad result for strings of 6 letters. The following abbreviations are going to be used in this section:

$S =$ Similarity estimation
$L_1 =$ Length of string 1
$L_2 =$ Length of string 2
$W_1 =$ Number of words string 1
$W_2 =$ Number of words string 2
$W_{1_i} =$ Number of letters word i from string 1
$W_{2_j} =$ Number of letters word j from string 2
$C =$ Cost of transformation (Levenshtein distance)

This first solution (Fig. 10) indeed provided a list of possible matches. The error allowed at the beginning of the development was 50% (must be remembered that for the purpose of harmonization, no correct matches could be left out by the filtering). However, even allowing such a considerable error, some correct matches were filtered.

$$S = \frac{(L_1 - C)}{\min(L_1, L_2)} \times 100$$

Fig. 10: First similarity estimation in plain search

As explained before, Levenshtein is more oriented for searches of a pattern in biggest texts (i.e. DNA sequences). After some testing, the reason for some correct matches to be left out by the filtering proved to be that the edit distance calculated by the algorithm gave high cost values to transform strings that consisted in the same words, but differently ordered. This was an undesirable behaviour for the algorithm, because a considerable quantity of the strings within the databases in GAMA project consist in person's names, where the order of "name" and "surnames" may change between different entries that, however, refer to the same person. The algorithm indeed worked for approximate string matching between sequences like "David Smith" vs. "Dvid Smmit", but it gave undesirable results for comparisons like "David Smith" vs. "Smith David" (Fig. 11).

		S	m	i	t	h		D	a	v	i	d
	0	1	2	3	4	5	6	7	8	9	10	11
D	1	2	3	4	5	6	7	6	7	8	9	10
a	2	3	4	5	6	7	8	7	6	7	8	9
v	3	4	5	6	7	8	9	8	7	6	7	8
i	4	5	6	5	6	7	8	9	8	7	6	7
d	5	6	7	6	7	8	9	10	9	8	7	6
	6	7	8	7	8	9	8	9	10	9	8	7
S	7	6	7	8	9	10	9	10	11	10	9	8
m	8	7	6	7	8	9	10	11	12	11	10	9
i	9	8	7	6	7	8	9	10	11	12	11	10
t	10	9	8	7	6	7	8	9	10	11	12	11
h	11	10	9	8	7	6	7	8	9	10	11	12

Fig. 11: Levenshtein algorithm with unwanted result

This comparison should be considered a possible match by the interface, but the Levenshtein algorithm gives as a result a cost of 12 to transform “David Smith” in “Smith David”. Strings formed by the same words, but differently ordered, result in a high Levenshtein distance when compared.

As the order of names and surnames could change in the entries of a database, but still refer to the same person, then some changes needed to be applied. The simplest and more effective solution was just to use a script to separate the words inside the string and then calculate the costs of the transformations comparing each word of the source pattern with each word of the target text. An individual percentage of similarity estimation would be given to each comparison word-by-word, and the result similarity estimation would be calculated using the sum of all percentages. To improve the accuracy of the algorithm, only the comparisons that passed a minimum value of 50% of similarity estimation were taken in account for the final similarity estimation value. These words inside the target string were considered possible matches, so the final similarity estimation was calculated dividing the sum of the similarity estimations of the possible matches by the number of words of the shortest string. The result was then an estimation of the similarity between both strings, but without penalising the orders of the words inside the strings (Fig. 12).

$$S_i = \frac{(W_{1i} - C)}{\min(W_{1i}, W_{2j})} \times 100$$

$$S = \sum \frac{S_i}{\min(W_1, W_2)}$$

Fig. 12: Second similarity estimation in plain search

Testing the algorithms confirmed that this solution was enough to make the filter consider possible matches strings where the words were in a different order and maybe misspelled. It was also considered that no correct matches were filtered keeping the same minimum value that was used in the previous implementation, which penalised the orders of words within the strings and left out correct matches. However, it added a new problem: in the databases some of the entries consist of large names or a combination of names of different artists. Then, a comparison between strings such as “David Julien Smith” and “Julia Smith Greg David” pass the filtering.

This couldn’t be avoided, as in the GAMA partner’s databases some strings contain a list of names, and the algorithm should be able to consider a possible match a comparison between one of the individual names with the string that contains that name mixed with other ones.

Also, as the results given are ordered by this estimation of similarity, and because there are still many comparisons where both strings have large number of words (that gives undesirable results if the order of words is not

penalised), another improvement was applied. First the process of separating by words and make a search of possible word matches between strings, will be only applied to the comparisons that doesn't pass a first filtering based only in a directly Levenshtein comparison. The strings that pass the filter will be given a similarity estimation based on the total number of letters minus cost of transformation divided by the **shortest** string length (that gives priority to the matches that are only misspelled but keep the same order of words). The ones that doesn't pass the first filter will be separated by words, each word of the source string will be compared with each word of the target string, and the number of possible word matches will be incremented if the comparison doesn't exceed the error allowed in the Levenshtein algorithm. Then the estimation of similarity will be calculated as the sum of the similarity estimations of each possible match divided by the **highest** number of words. To end, the strings that have been applied this second filtering will have to pass a minimum value of similarity estimation to be presented as possible matches (Fig. 13).

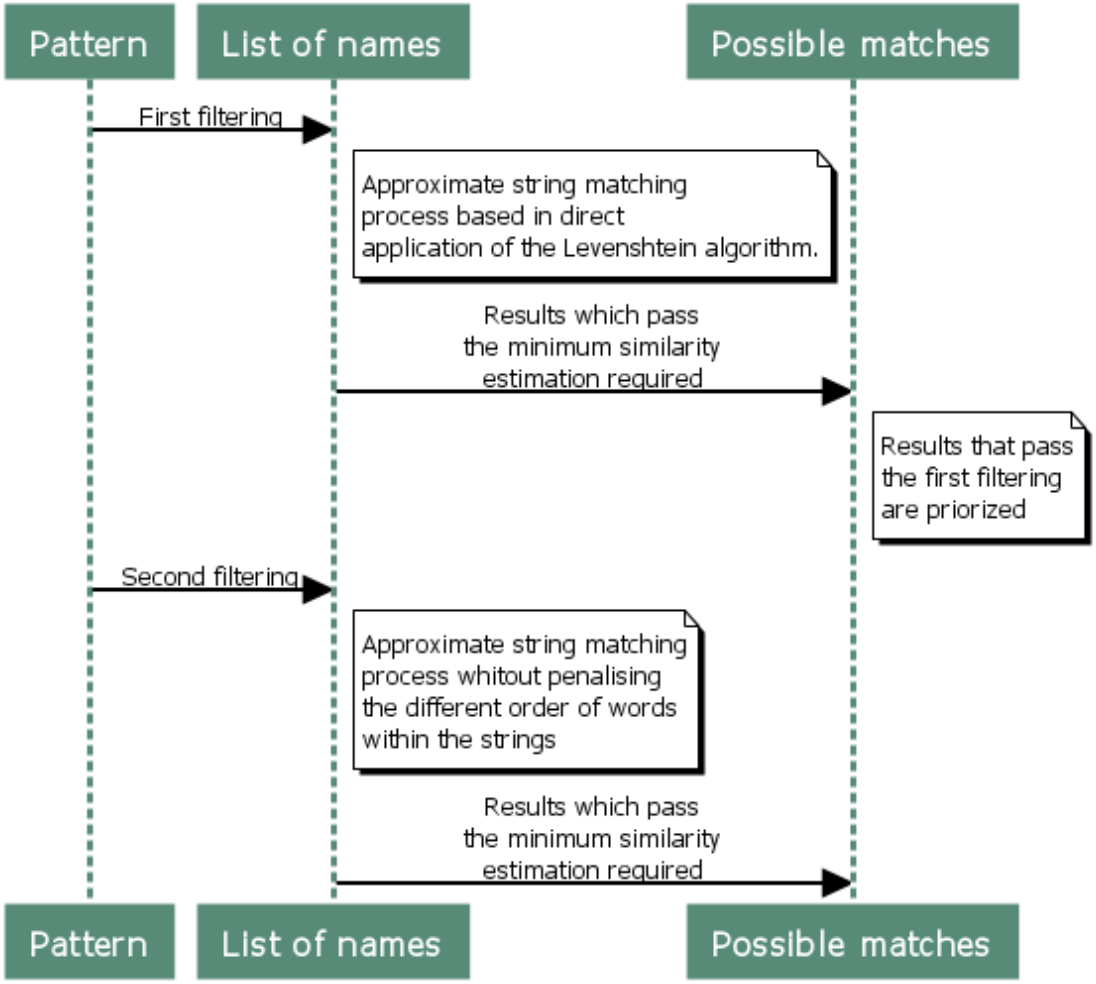

Fig. 13: Sequence diagram for plain search

Note that this use of two different filtering processes introduces also two different measures for the similarity estimation: the values obtained with each

filtering applied to the same pair of strings can be completely different if the order of the words changes.

This is not a problem because the numeric value of the similarity estimation is not as important as to provide a system that really makes an approximate string matching process that satisfies the requirements needed. That is, reduce to the maximum the results provided, but without leave out any correct match. Introducing two different filtering processes and similarity estimation measures improve considerably both aspects.

For completing the filtering process, 3 other improvements were applied:

- In the word by word comparison process, before applying the Levenshtein algorithm a first simple comparison will be made: if one of the strings corresponds to the other string first letter, followed by a ".", and vice versa, it will be directly considered a possible word match. That is, "David" and "D." will be considered a possible word match without passing any other process.
- The surname of an artist provides more information about the person than his name, so before making the comparison between 2 strings, name and surname will be separated within each string (entries in the database which contain a "," will be considered a name which contains 2 parts, name and surname, and the "," will be considered the separator). Note that this is a condition assumed in available databases, and that this behaviour of the algorithm can be easily modified if required. Then the comparison algorithm will stay the same, but will be taken place between the "name" strings and the "surname" strings, not between both complete strings. The total similarity estimation will be able to be modified assigning priority to the surnames (this can be done simply multiplying the result of the "names" strings comparisons by factor, and the results of the "surnames" comparisons by a higher factor, considering that the sum cannot surpass 100%).
- The strings will be cleaned of any other symbol than letters of the alphabet. That is, comas, dots, extra blank spaces, etc. will be removed.

2.1.6 SEARCH WITH FEATURES ALGORITHM

The algorithm is similar than the one used for the second filtering during the plain search, but it introduces a new concept for estimating the similarity between two strings: the phonetic resemblance.

In combination with Levenshtein algorithm, a phonetic algorithm is used: Metaphone. The different codes produced for different strings by Metaphone vary in the parts where they are phonetically different, so a minor difference in the pronunciation is encoded as a minor difference in the Metaphone code. This characteristic allows to use the concept of edit distance between the codes produced by the phonetic algorithm for making an approximate string matching based in their pronunciation.

The first processing of the strings compared consists in separating them by words, with the purpose of making the comparisons word by word instead of comparing directly the full strings. Then, each word is encoded according to its pronunciation and stored in an array.

As in the second filtering of the plain search, each word (in this case, each phonetic code that represents the word) of the source string is compared with each word (again, the words stored in the array are now the phonetic codes that represents that word) of the target string. The comparison is based in the Levenshtein algorithm.

Given two different Metaphone codes the edit distance between them is calculated, and if its value is low enough then they are considered as a match. Then a similarity estimation is calculated for each pair that passes the limit value, and the total similarity estimation is calculated dividing the sum of the similarity estimations between possible word matches by total number of words of the shortest string (in terms of number of words) (Fig. 14).

Strings that satisfy a minimum value of similarity estimation are then considered as correct results.

$S =$ Similarity estimation
$W_1 =$ Number of words string 1
$W_2 =$ Number of words string 2
$M_{1i} =$ Metaphone code of word i from string 1
$M_{2j} =$ Metaphone code of word j from string 2
$C =$ Cost of transformation(Levenshtein distance)

$$S_i = \frac{(M_{1i} - C)}{\min(M_{2i}, W_{2j})} \times 100$$

$$S = \sum \frac{S_i}{\min(W_1, W_2)}$$

Fig. 14: Similarity estimation in search with features

The values of similarity estimation obtained with this algorithm differ from the ones obtained with the plain search one. For example, a comparison between “Smith” and “Smmit” will result in a similarity estimation of 100%, as the edit distance is calculated between phonetic encodings and in this case both strings are encoded equally by the Metaphone algorithm. This can be useful as cases like this one (names with different spelling but same or similar phonetic) are expected during the harmonization process, and they are possible candidates to refer to the same person.

The phonetic encoding and comparison was done directly word by word, instead of doing a first priorized filtering that penalised the different orders of words, due to unexpected results that could appear encoding full strings. Metaphone algorithm treats differently the first letter of a given string, and the phonetic algorithm considers the adjacent letters to make the transformation. If not separated by words, the information of the first letter will be lost, and the given by the adjacent letters will be in some cases wrong (as the adjacent letters that are part of different words in a string doesn't affect the same way the pronunciation as the letters within the same word). It indeed would provide good results, but not as good as by making the comparisons word by word. Therefore, there was no sense in making a prioritized filtered penalising the order of the words because many wrong results would pass the filter, and once is done the process of separating by words the results of this filtering also contain the ones that would appear with the direct comparison one (Fig. 15).

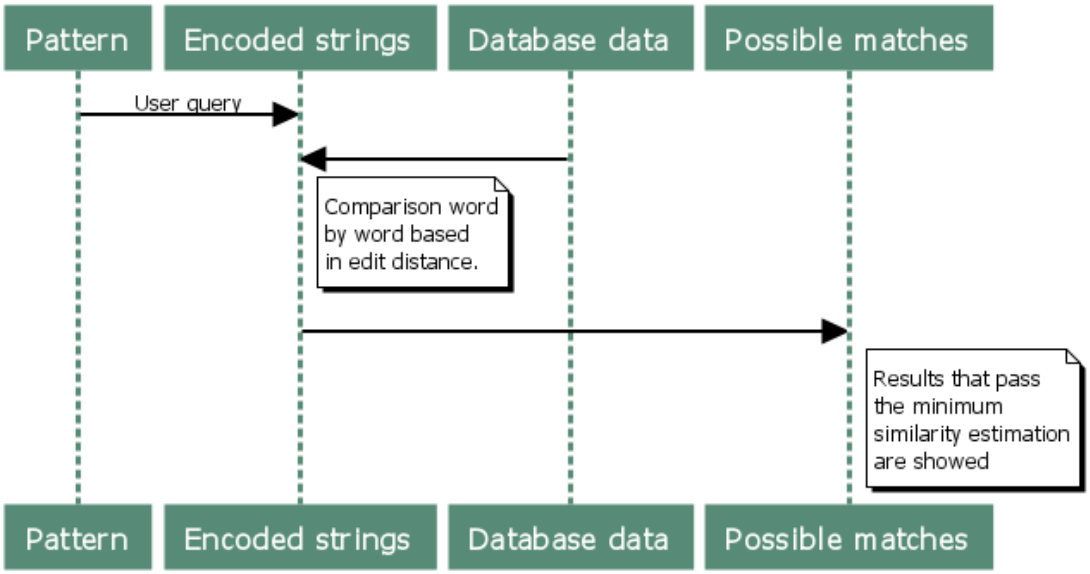

Fig. 15: Sequence diagram for search with features

2.2 IMPLEMENTATION

The language used for the implementation was PHP. As the interface is going to be used online by the editors (the different databases are located in different points around Europe), it provides a powerful tool to build the HTML pages which they will use to make the harmonization process, and allows the exchange of info with the databases using MySQL queries. No other languages (JAVA, Perl, etc.) were needed, all the functionalities expected could be developed in that language without increasing the effort of the development. Also, free implementations of the Levenshtein and Metaphone algorithms were available for the PHP distribution used.

The files involved in the harmonization process are (Fig. 16):

- `harmonizationindex.php`: the main page. Generates the first visual interface where the names that are available in the database are listed. Communicates with the database, and shows in a table a list of names based in one of these four characteristics: all names (default), all edited, all non edited, names by keywords/letters. Sends the name selected to `results.php` or `collected.php`.

- `results.php`: gets the name selected in `harmonizationindex.php`, and makes the process of approximate string matching explained before to provide the list of possible matches. To make the process, calls the function `similar2`, which code is implemented in `similar.php`, or `similar_sound`, implemented in `similar_sound.php`. The user then has to select the ones that refer to the same person to match them, assigning the type of relation desired.

- `collected.php`: gets the name selected in `harmonizationindex.php` or `results.php` and displays all the related entries in the database to that name. It provides an interface where the user can make changes to the data related to the database entry previously selected.

- `similar.php`: has the full code for the approximate string matching process. The function itself is called `similar2`, and it uses the string provided, plus the data gathered from the database, and the function `separate_words`. It also uses the PHP implementation of the Levenshtein algorithm, which is a function available in the free distribution of PHP.

- `similar_sound`: has the full code for the approximate string matching process based in phonetic characteristics. It uses the string provided, plus the data gathered from the database, and the function `separate_words`. It also uses the PHP implementation of the Levenshtein algorithm and metaphone, which are functions available in the free distribution of PHP.

- `separate_words.php`: contains the code of the function that separates the words of a string and returns an array, with a single word of the string in each position of the array.

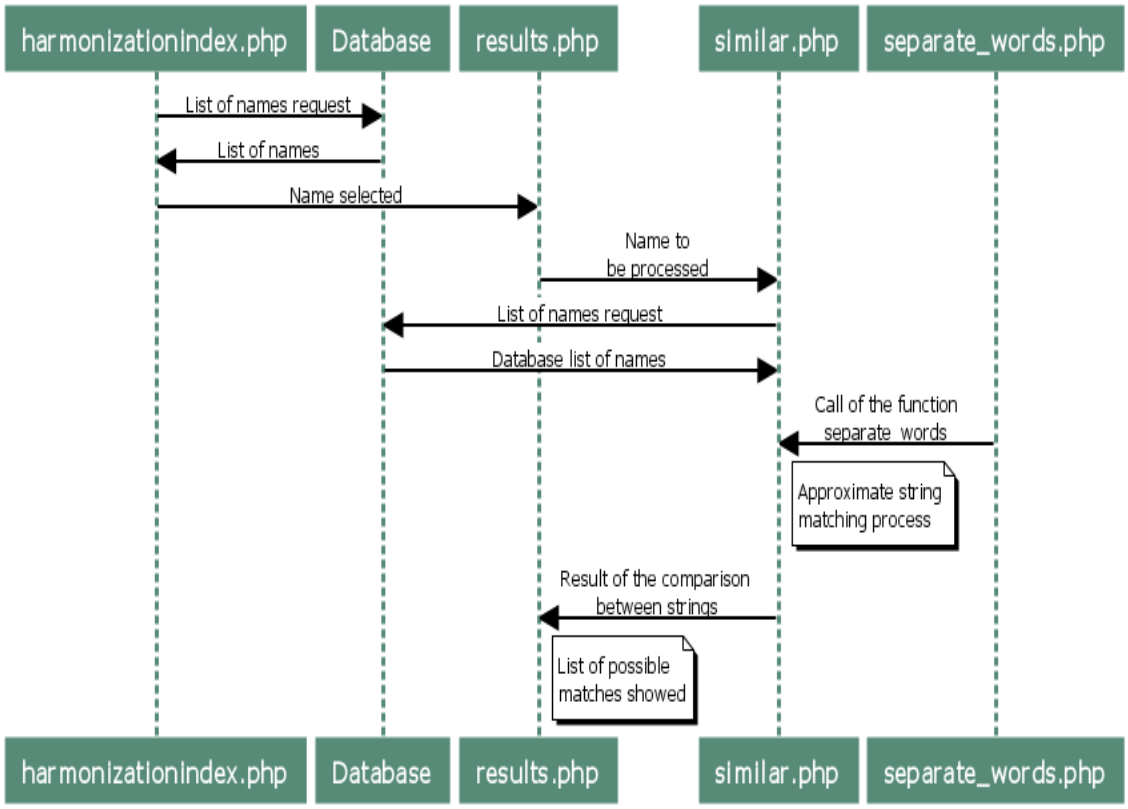

Fig. 16: Sequence diagram

The data stored in the database contains different fields of information, used for the purpose of harmonization:

- Name: The main information of one entry of the database. Contains the name that has to be harmonizated.

- Relation: Numeric value which contains the ID of another entry of the database with which it is related, in case a relation with another entry has already been established.

- Type: Stores information about the kind of relation (pseudonym, incorrect spelling variation, related person/group, etc.) established with another entry of the database, which id is stored in the field “relation”.

- ID: Numeric value, different for each entry of the database, used for identifying the entries.

- Main: Numeric value. If it is equal to 1 the entry of the database is considered a main form.

- Searchable: Numeric value which value determines if the entry is shown when making general searchs and using the filters.

- Status: Stores information about the status of the harmonization process concerning that name (“ok”, “unsure”,” revised by expert”).

- Comments: Field where the editors can store comments about that entry.

2.2.1 HARMONIZATIONINDEX.PHP

The file mainly consists on the code for reading the entries from the database, and displaying them considering the filters applied by the user and the pagination process.

VARIABLES AND FUNCTIONS DOCUMENTATION

The variables and functions used for the implementation are:

db_query($ sql)
Connects to the database and informs in case of error

$keywords = $_GET[“keywords”]
Contains the keywords entered by the user for searching in the database

$number_matches = mysql_num_rows($same_relation)
Total number of already done matches with the name in the same row of the table

$option = $_GET[“option”]
Contains the info about the search filter selected by the user

$pag = $_GET[“pag”]
Contains the number of the actual page displayed

$reg1 = ($pag-1)*$tampag
Start point of the results array that has to be showed

$row = mysql_fetch_array($query)
Contains each result row

$same_relation = mysql_query(“SELECT * FROM names WHERE relation LIKE ‘$row[id]’”)
Query for selecting the names that are already matched with the one displayed in the same row of the table

$tampag = 10
Maximum number of results showed per page

$total = mysql_num_rows($query)

Total number of rows that match with the user's query. It is used for pagination

2.2.2 RESULTS.PHP

It consists on the code for calling the functions that make the process of calculating the similarity estimation between strings and organise the results according to that value, with pagination if necessary. Contains different PHP formularies used to make changes in the database regarding the kind of relation that the editor wants to establish.

VARIABLES AND FUNCTIONS DOCUMENTATION

The variables and functions used for the implementation are:

db_query ($ sql)

Connects to the database and informs in case of error

similar2 ($ *str1*, $ *str2*)

Compares two strings with an algorithm based in the edit distance between both of them.

Parameters:

$str1: first string
$str2: second string

Returns:

Percentage of similarity

similar_sound ($ *str1*, $ *str2*)

Compares two strings with an algorithm based in phonetic codes and Levenshtein algorithm.

Parameters:

$str1: first string
$str2: second string

Returns:

Percentage of similarity

$checked = $_GET["checked"]

If equal to 0, no checking of identical names already done

global $id_changes

Id to which the new matches have to be related to

$id_original = $_GET[“id”]
Contains the id of the query made by the user

global $id_targets
Id’s of the names that have to be unmatched with the search

global $id_targets2
Id’s of the names that have to be matched to the search made by the user

$method = $_GET[method]
Search method selected by the user: plain or with features

global $next
Next page to be displayed

$number = mysql_num_rows($same)
Total number of identical names

$number_matches = mysql_num_rows($same_relation)
Total number of names already matched with the user search

global $pag
Actual page displayed

$porcentage = intval($porcentage)
Similarity estimation between the search made and one name in the database

global $previous
Previous page to be displayed

global $relative_position
Position within the results array that corresponds to the first result of the actual page

global $results
Array with 4 fields:name, id, relation, percentage of similarity with the search made

$same = mysql_query(“SELECT * FROM names WHERE name LIKE ‘$search’”)
Query for selecting the names that are identical to the search made by the user

global $same_relation
Query for selecting the names already matched with the user search

$search = $_GET[“search”]
Stores the query made by the user

global $tampag
Number of results showed per page

global $total_changes
Total number of unmatches that have to be done in the database

global $total_changes2
Total number of new matches that have to be done in the database

global $total_results
Length of the $results array

2.2.3 SIMILAR.PHP

It contains the code for the function used in a plain search for calculating the similarity estimation between two strings.

2.2.4 SIMILAR_SOUND.PHP

It contains the code for the function used in a search with features for calculating the similarity estimation between two strings.

2.2.5 SEPARATE_WORDS.PHP

It contains the code for the function used in `simlar2` and `similar_sound` for, given a string, create an array with one word of the string for each position of the array.

3. TEST RESULTS

This chapter presents the results of testing the algorithms with data from one of the GAMA partner's database. The aim of the testing process was to study the efficiency of the searching algorithms, in order to improve them until the results were the expected. As both algorithms (based only in edit distance, and based in phonetic codes plus edit distance) provided an estimation for the similarity between strings, testing was also used to define a minimum value for this measure for consider a string a possible match.

The test was executed using real data existing within GAMA partner's databases (Netherlands Institut voor Mediakunst Montevideo, Ars Electronica), plus a variety of additional data samples specially created with determined characteristics, that are relevant for studying the functionality of the algorithms. That is, given a query that is going to be tested, some variations of it were added. Variations contained the query with some misspellings. Misspellings consisted in the adding of some letters, from small numbers (regarding to the number of letters of the query), were the searching algorithms should consider them as a possible match to the query, to biggest numbers. They also consisted in deletion and transformation of letters.

Additions, transformations, and deletions of letters in the additional data samples were done at random and considering grammatical and phonetic rules. For example, duplication of letters or deletion of letters which sound didn't alter the pronunciation of the query were always done to test how the algorithms worked with that kind of modifications.

To determine whether a certain string matching algorithm provided better results for small or large strings, two subsample data sets were produced. One consisted in strings of 10 letters or less, and the other of queries of more than 10 letters.

3.1 SMALL STRINGS RESULTS

For testing the approximate string matching algorithms for small strings, the parameters were the following:

- Strings of 10 letters or less.
- 15 strings were tested over a database of 4650 strings.
- 10 strings were considered correct matches for each tested string.
- Different minimum similarity estimation values were tested: 50%, 60%, 65%, 70%, 75%.
- The results showed in the graphics consist in:
  - Precision: number of correct matches retrieved, divided by the total amount of results retrieved (that is, the ones that pass the minimum value of similarity estimation).

$$Precision = \frac{\#\,correct\ matches\ obtained}{\#\ matches\ obtained}$$

  - Recall: number of correct matches retrieved, divided by the total amount of correct results existing in the database.

$$Recall = \frac{\#\,correct\ matches\ obtained}{\#\ correct\ matches}$$

3.1.1 SIMILARITY ESTIMATION 50%

Both algorithms provide as possible matches all the correct results. The number of wrong results mixed with the correct ones is high, so the minimum similarity estimation value can be increased (Fig. 17).

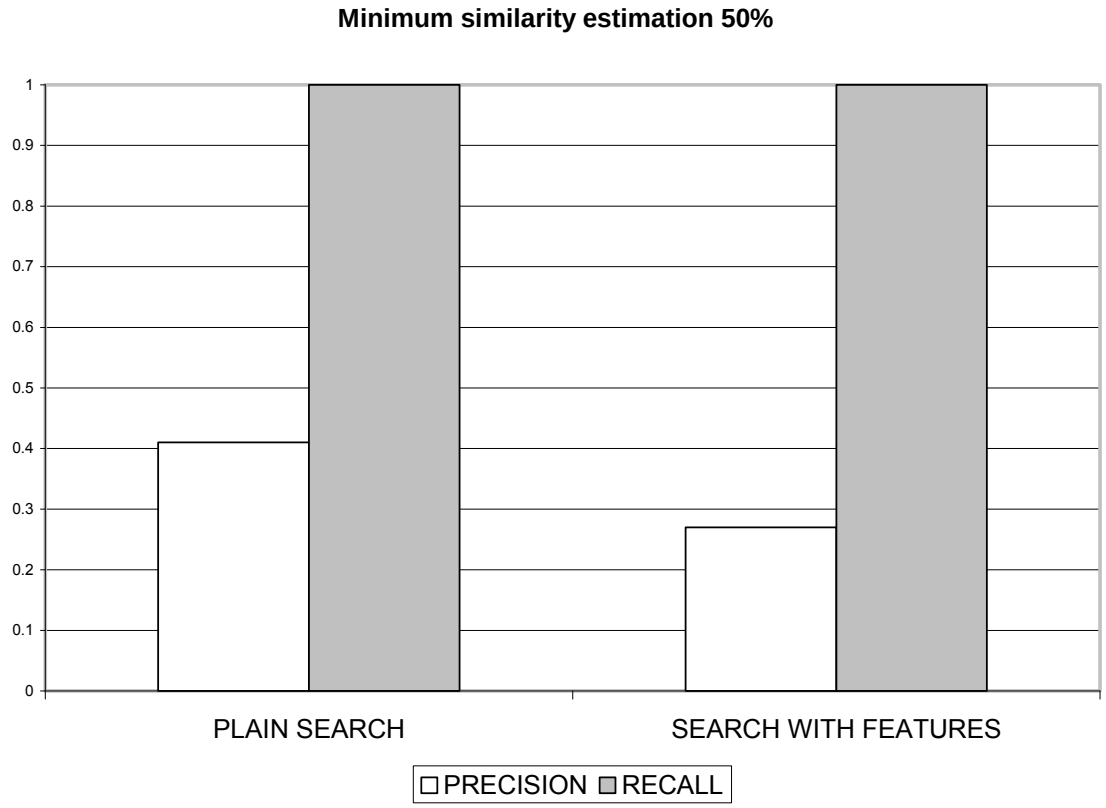

Fig. 17: Test results with minimum similarity estimation 50%

The algorithm based in phonetic features provides a high number of results, it is less accurate than the based in edit distance. This is reasonable, considering that it doesn't consider most of the vowel sounds, and that some differences between consonants (if the difference doesn't affect the pronunciation) doesn't affect the value of similarity estimation obtained.

3.1.2 SIMILARITY ESTIMATION 60%

The improvement of the relation between wrong results and correct ones is considerable with just increasing the minimum value of the similarity estimation to 60% (Fig. 18).

Both algorithms leave out of the results correct ones. For the purpose of the harmonization process this is not viable, but 60% is still a good minimum value if the process is done using both algorithms: correct results filtered by the plain search are not the same as the ones filtered by the search with features. Combining the results of the two algorithms all the correct ones are obtained.

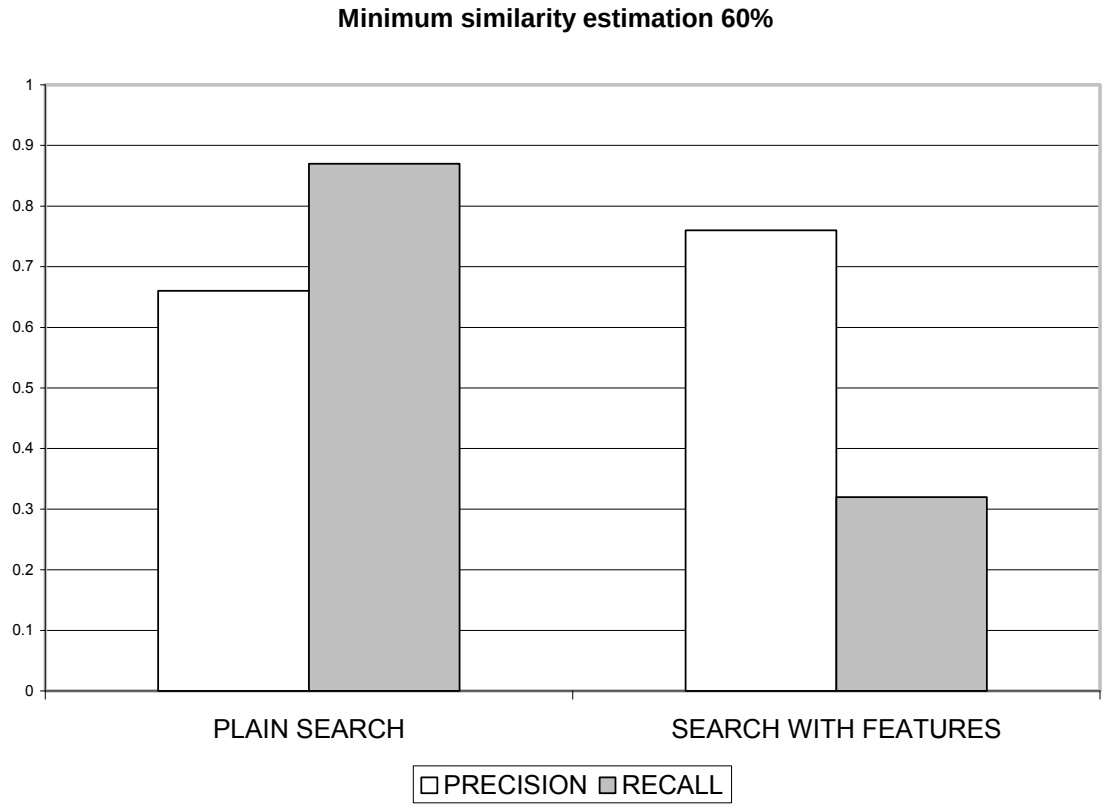

Fig. 18: Test results with minimum similarity estimation 60%

This is due to the differences between the algorithms: In the subsample data set used for the testing, the misspellings applied to the original string were not only applied at random. Some consisted in duplicating of letters (as it is considered a typical typing mistake), or changes that didn't affect the pronunciation. Levenshtein algorithm is sensible to duplication of letters or substitutions, without considering if they affect or not the pronunciation. For example, considering "Kyoko Abbe" the original string, and "Kioco Ave" the misspelled one, the edit distance assigning a cost of 1 to all operations (deletions, additions, substitutions) is 4. In relation with the length of the original string, this would result in a similarity estimation lower than 60%. However, this case should be considered a possible match. The algorithm based in phonetic features doesn't filter it, as the changes y-i, k-c, b-v, and the duplication of letters don't change the phonetic encoding provided by Metaphone.

3.1.3 SIMILARITY ESTIMATION 65%

There are no changes in the results obtained with the algorithm based in phonetic features. In the one based in edit distance, the number of wrong results continue to descend and the percentage of correct ones to increase (Fig. 19).

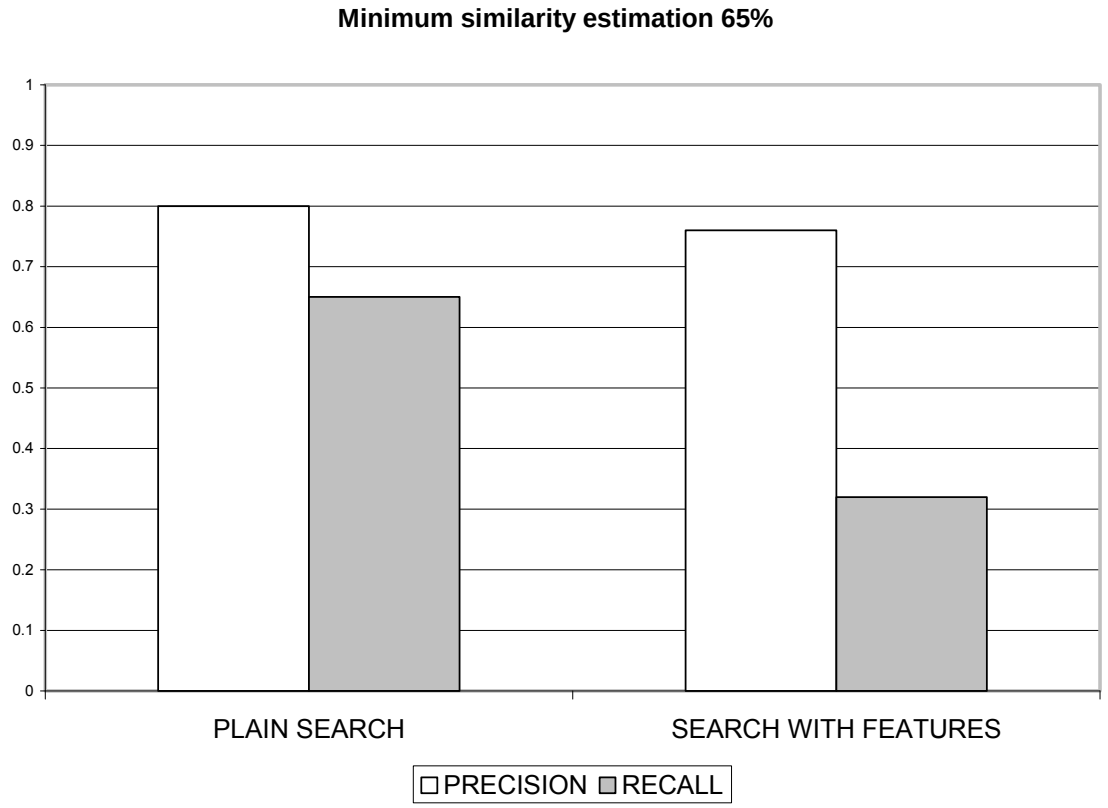

Fig. 19: Test results with minimum similarity estimation 65%

A small percentage of correct results (5%) is filtered even combining both algorithms. Note that the criteria for considering a misspelled version of the original string as a correct result that should be considered a possible match is subjective. For the testing the worst scenario has been considered, strings with a number of misspellings that may not appear in the partners database are considered correct results. Probably the process of harmonization would not leave correct results out with a limit value of 60% or even higher, and the number of results provided decreases considerable increasing it (this facilitates the task of the editor). Once the system is working with the data from all databases a decision will have to be made, equilibrated between accuracy and load of work.

3.1.4 SIMILARITY ESTIMATION 70%

The results obtained by the phonetic algorithm are the same (Fig. 20). The reason is that the difference obtained between the 50% and 60% limit consisted in results which misspellings were basically transpositions and substitutions, typing errors that are more appropriate to treat with algorithms based in the edit distance. However, the limit of 50% is so low that these strings passed the filter based in phonetic features. After the 60% limit these values don't pass the filter, and the ones that are provided as possible matches contain high values of similarity estimation, because of the behaviour of the Metaphone encoding: misspellings that don't affect the pronunciation doesn't affect the

phonetic code generated, so strings that sound the same but are different spelled have the same codification. This is translated in a high similarity estimation, no matter the number of misspellings if the phonetic is not affected, and in a low similarity estimation if it is. Then, the results of the search with features search are divided between two groups: strings with same phonetic, that have a similarity estimation of more than 75%, and strings with contain misspellings of other kind, that have a similarity estimation lower than 60%. The changing of the limit will then not affect the results obtained with this algorithm if it is higher than 60%.

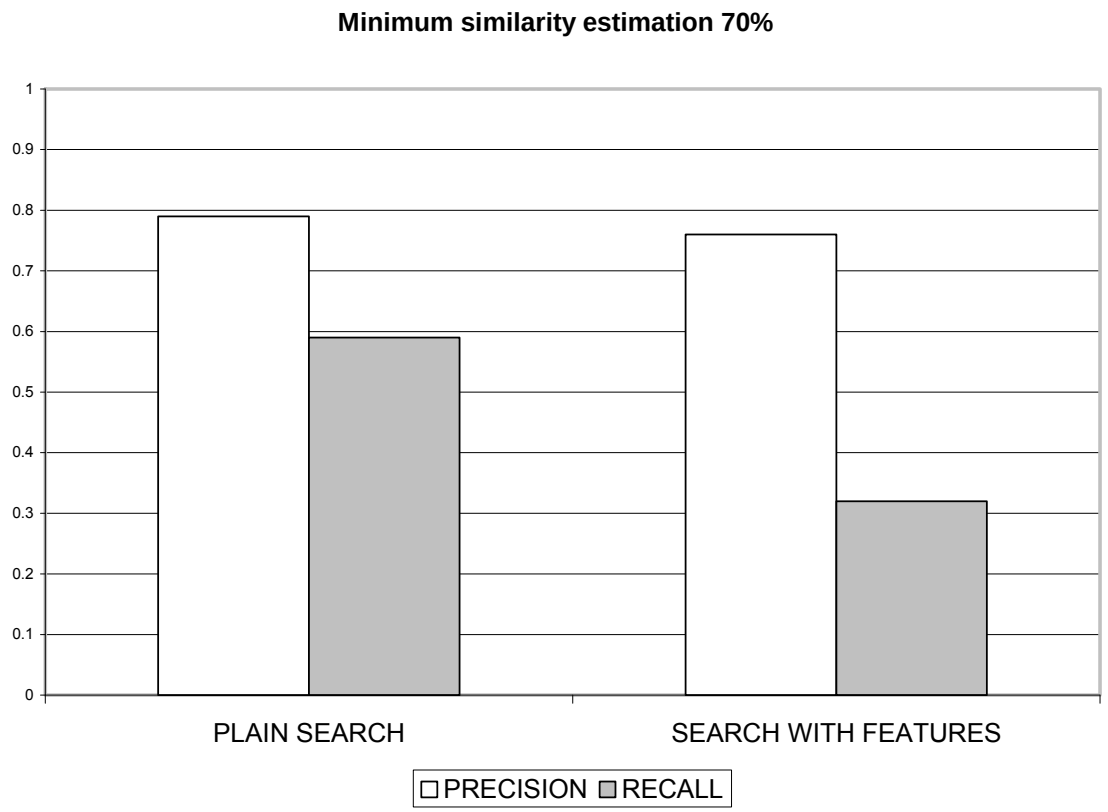

Fig. 20: Test results with minimum similarity estimation 70%

With the plain search algorithm the number of correct results continues to decrease, but also start to decrease their percentage in relation with the total number of results provided. The conclusion is that to a big percentage of the correct results is given a similarity estimation between 65% and 70%. Again note that the criteria for consider a misspelled string a possible match of the original one is subjective, so it doesn't mean that 70% is a bad limit for starting considering possible matches.

3.1.5 SIMILARITY ESTIMATION 75% and 80%

There are no big differences in the number of total results provided but the total correct number of results decreases slightly (Fig. 21 and Fig. 22).

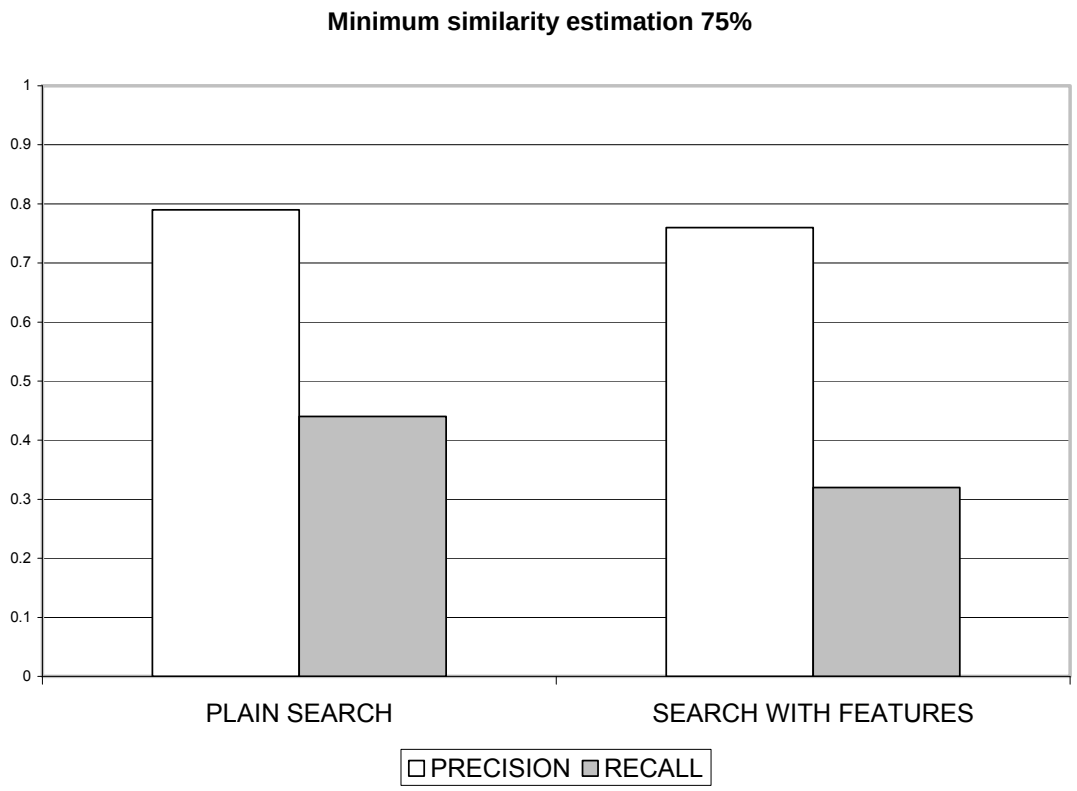

Fig. 21: Test results with minimum similarity estimation 75%

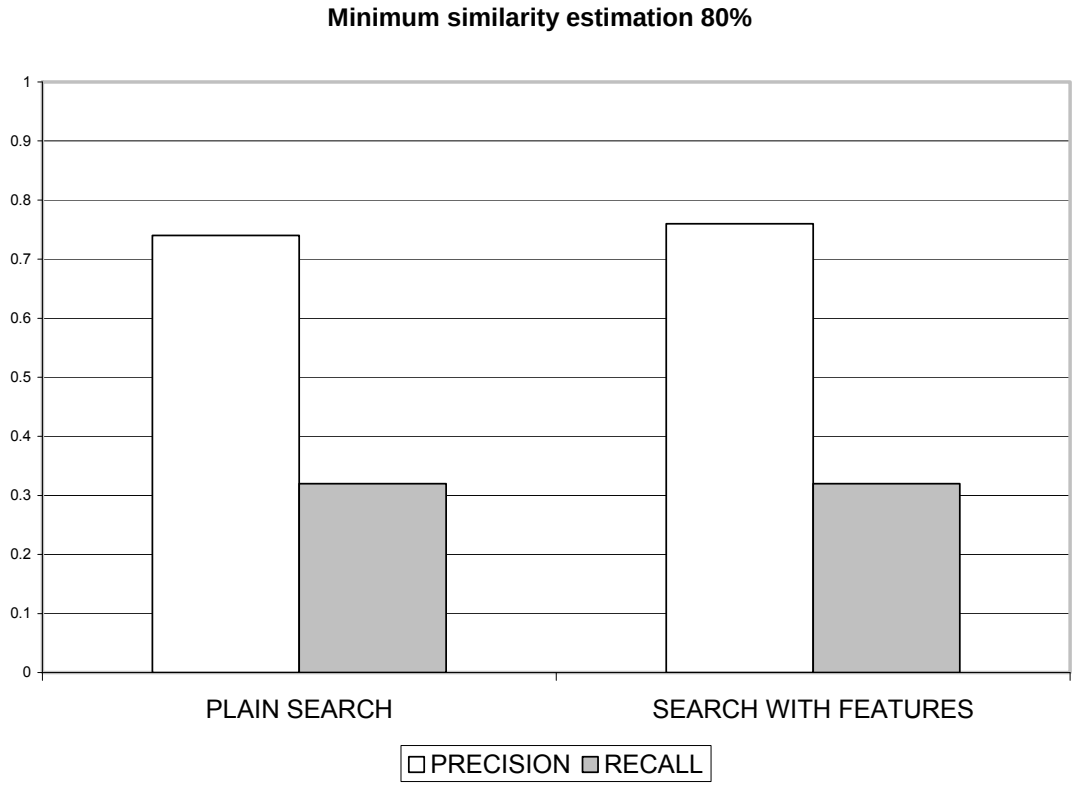

Fig. 22: Test results with minimum similarity estimation 80%

As there are no improvements and some correct results are filtered, an optimal limit value for the similarity estimation should not surpass 70%. The conclusions achieved with the testing process with this subsample data set are treated at the end of the chapter, along with the ones obtained with the long strings subsample data set.

3.2 LONG STRINGS RESULTS

For testing the approximate string matching algorithms for long strings, the parameters were the following:

- Strings of more than 10 letters.
- 15 strings were tested over a database of 4650 strings.
- 10 strings were considered correct matches for each tested string.
- Different minimum similarity estimation values were tested: 50%, 60%, 65%, 70%, 75%.
- The results showed in the graphics consist in:

  - Precision: number of correct matches retrieved, divided by the total amount of results retrieved (that is, the ones that pass the minimum value of similarity estimation).
  - Recall: number of correct matches retrieved, divided by the total amount of correct results existing in the database.

3.2.1 SIMILARITY ESTIMATION 50%

Both algorithms provide all the correct results as possible matches (Fig. 23). However, the accuracy of the plain search is higher, and the number of wrong results is almost the same as the number of correct ones.

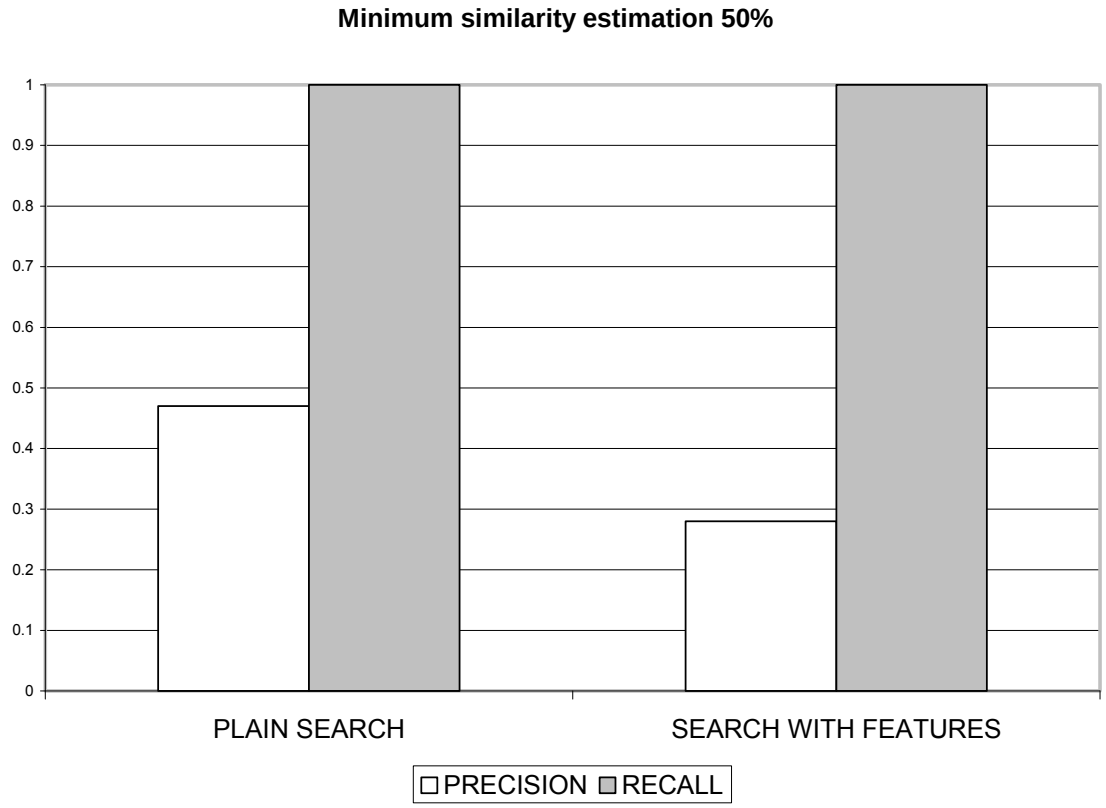

Fig. 23: Test results with a minimum similarity estimation 50%

The search with features algorithm provides a high number of wrong results. With a limit of 50% for the similarity estimation value, considering the characteristics of the phonetic codes generated by Metaphone, is highly probable to obtain wrong results. The phonetic algorithm assigns high levels of similarity estimation to correct matches (more than 70%) which misspellings doesn't affect the pronunciation of the string, and gives low values (less than 60%) for correct matches which misspellings consist in transpositions or deletions, and for wrong matches.

For both algorithms, the limit value for similarity estimation can be increased to improve the results obtained.

The results are better than the ones obtained with the subsample data set of short strings, but the improvement is very little.

3.2.2 SIMILARITY ESTIMATION 60%

The improvement of the results obtained with the algorithm based in edit distance is notable (Fig. 24). The number of wrong results mixed with the correct ones is reduced, but without filtering any of the good ones. With this limit value for similarity estimation some correct results were filtered in the subsample data set of short strings.

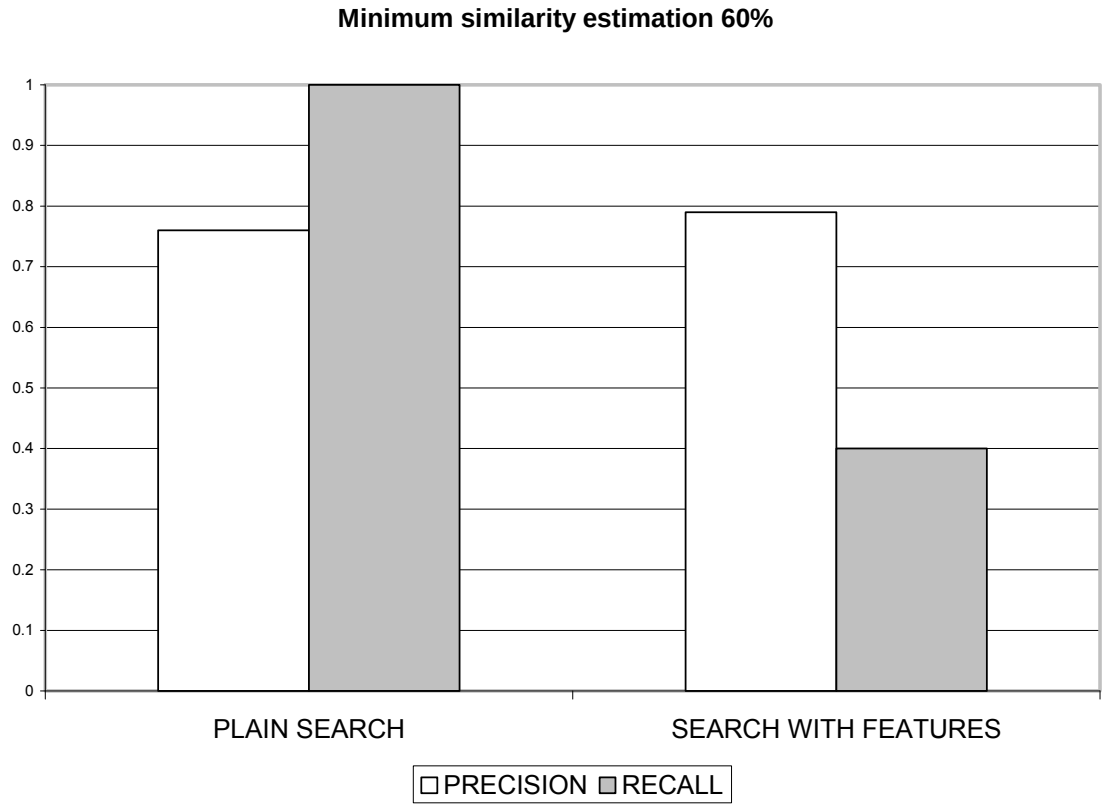

Fig. 24: Test results with minimum similarity estimation 60%

The reason is that the longer one string is, the less probable is that exists in the database a string of similar length with a high similarity estimation value (with each alphabet letter added to the string, the probability of resemblance is reduced, as it is more probable to add a letter that differs from the letter in the same position within the original string, than the same one). Then the number of wrong results is reduced, and also the filtering of good results is reduced: the longer one string is, the less proportion of misspellings it will have. Typing and spelling mistakes follow a pattern of reduced number of occurrences per word. The probability of a typing error is proportional to the number of letters typed, but when it happens it can occur in a word of 3 letters, or in a word of 10. That affects highly in the similarity estimation of short strings, but not as much in long strings.

The phonetic algorithm also improves the relation between correct matches and wrong ones, but it leaves out more than half of the correct results that should be considered possible matches. As in the case of the subsample data set of short strings, misspellings that consist in transpositions or deletions of consonants result in a low value of similarity estimation that doesn't pass even a limit of 60%. This can be seen as a disadvantage of the algorithm, but it is this same characteristic the one that allows it to give high values of similarity estimation when the phonetic is not altered. This is also the case when the algorithm based in edit distance works worse.

3.2.3 SIMILARITY ESTIMATION 65%

The algorithm based in edit distance continues to improve the relation between correct results and wrong results as the limit value for similarity estimation increases (Fig. 25). However, some correct results start to be filtered (6%). The phonetic algorithm provides the same results as in previous case.

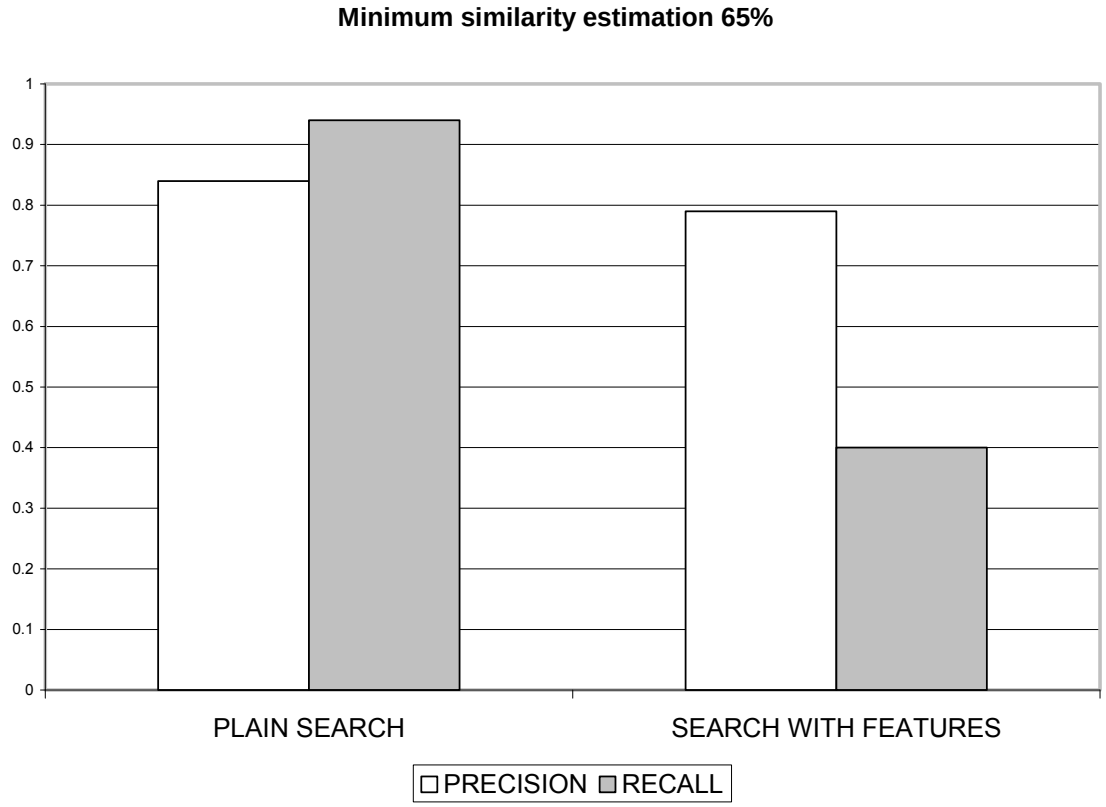

Fig. 25: Test results with minimum similarity estimation 65%

As happened with the subsample data set of short strings, correct results filtered by both algorithms are complementary, so the minimum value for similarity estimation can be increased to reduce the number of total results provided, and combine the results obtained with them.

3.2.4 SIMILARITY ESTIMATION 70%

The algorithm based in edit distance continues to improve the relation between correct results and wrong results as the limit value for similarity estimation increases (Fig. 26). The number of filtered correct results also increases, but as in the previous case, it is still a high value (83%), and the complementation between the filtered by each of the algorithms allows to continue increasing the limit value for similarity estimation without leaving out any possible match.

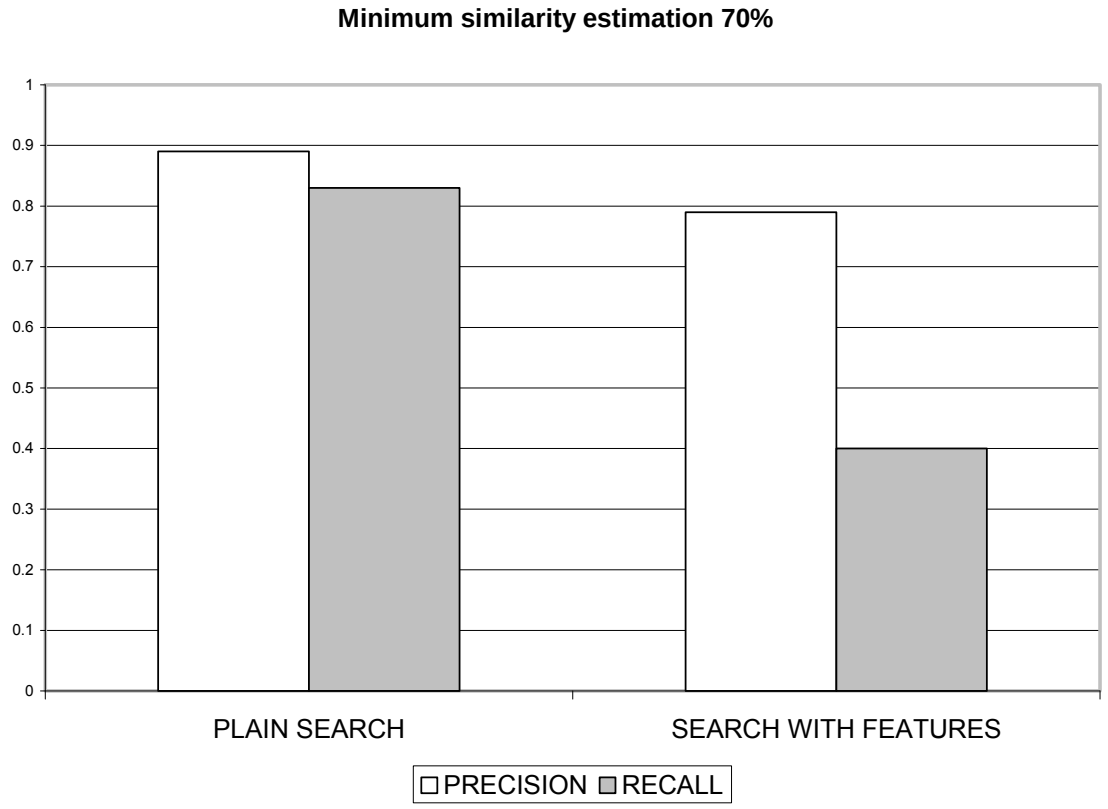

Fig. 26: Test results with minimum similarity estimation 70%

With the subsample data set of short string the relation between correct and wrong results started to descend at this point. Here, it continue increasing. That means that the proportion of possible matches with a similarity estimation value higher that 70% is bigger when working with the subsample data set of long strings.

3.2.5 SIMILARITY ESTIMATION 75% AND 80%

There are no improvements in the results provided by both algorithms (Fig. 27 and Fig. 28). The phonetic one provides the same results as in the previous case, and the one based in edit distance not only continue providing less quantity of correct matches, but also the relation between correct and wrong ones start to decrease for a limit of 75% for the similarity estimation, and continues decreasing with for limit of 85%.

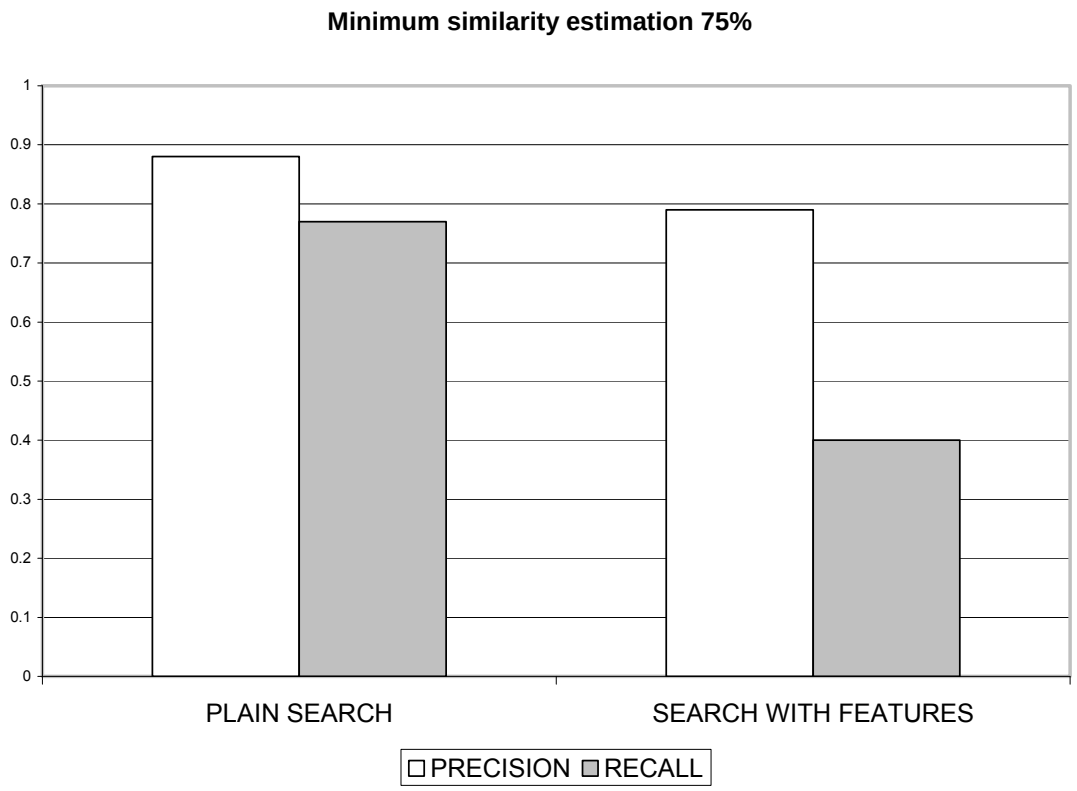

Fig. 27: Test results with minimum similarity estimation 75%

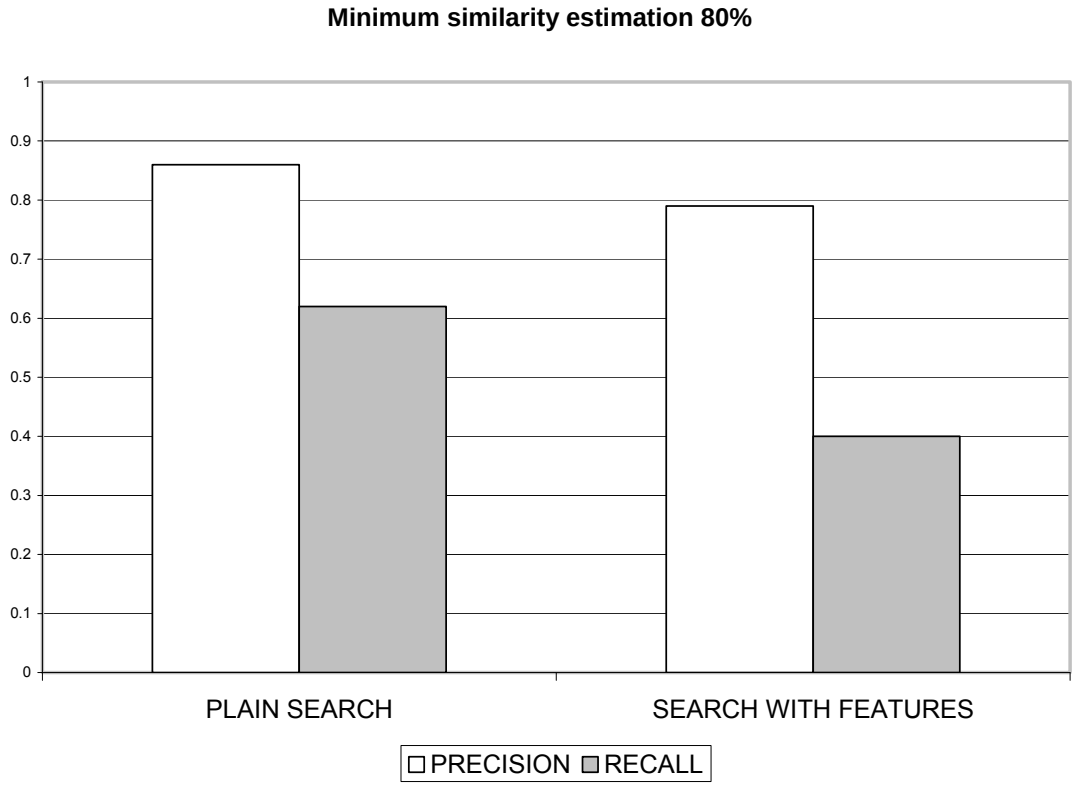

Fig. 28: Test results with minimum similarity estimation 80%

Filtered correct results continue to be complimentary between both algorithms, but as the relation between correct results and wrong ones decreases there is no sense in increasing to this point the limit of similarity estimation (it will not facilitate the task of the editor if this relation is not improved).

3.3 TEST CONCLUSIONS

The relation between correct matches and wrong ones is the most important parameter in order to simplify the task of the editors (Fig. 29 and Fig. 30). The evolution of this relation was expected to increase as the value for the minimum similarity estimation increased, and the testing indeed confirmed this hypothesis. However, there is a limit for this value where this relation starts to decrease. That means the accuracy is so high that the algorithms start to consider simple misspellings in correct results as wrong matches.

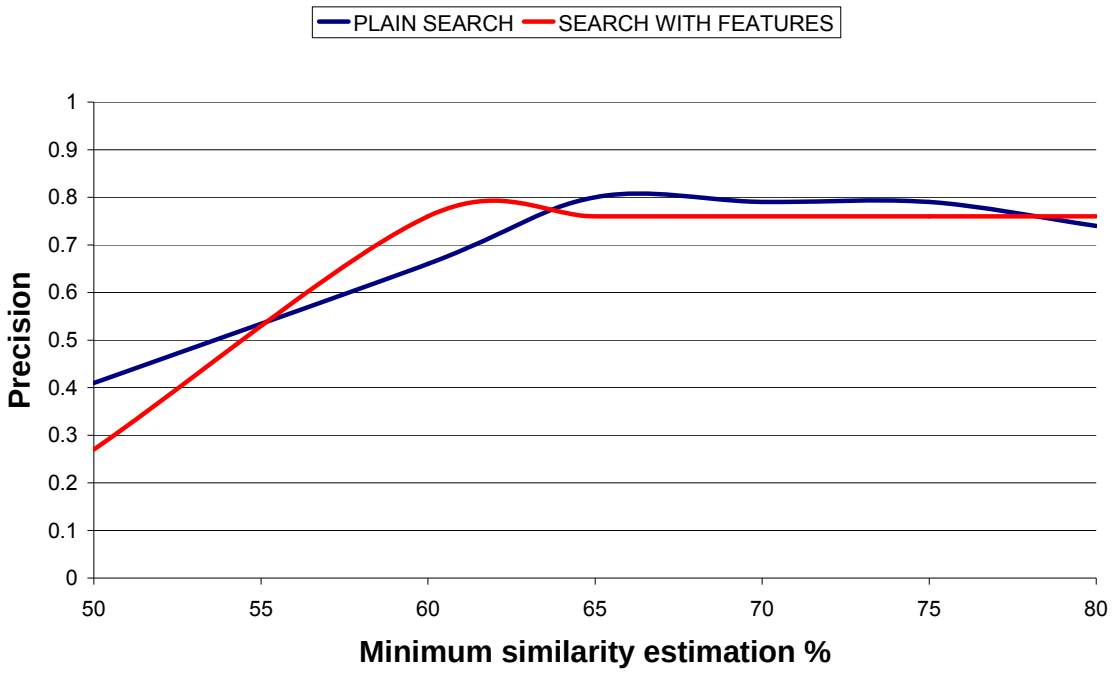

Fig. 29: Precision for short strings sample data set

The number of wrong results considered a possible match by the algorithms decreases as the value for the minimum similarity estimation increases, but there is a limit value from where the improvement in number of wrong results filtered, doesn't compensate the increase of number of correct matches considered wrong results by the algorithms (due to a high minimum similarity estimation). The cause is that for high values of the minimum similarity estimation, most of the wrong matches have already been filtered as wrong results, so an increase of this limit value will not significantly affect the total number of wrong matches still considered correct results by the algorithms. However, the number of correct matches considered wrong results by the algorithms (due to misspellings within the strings that reduce the minimum similarity estimation under the limit value) starts to increase exponentially.

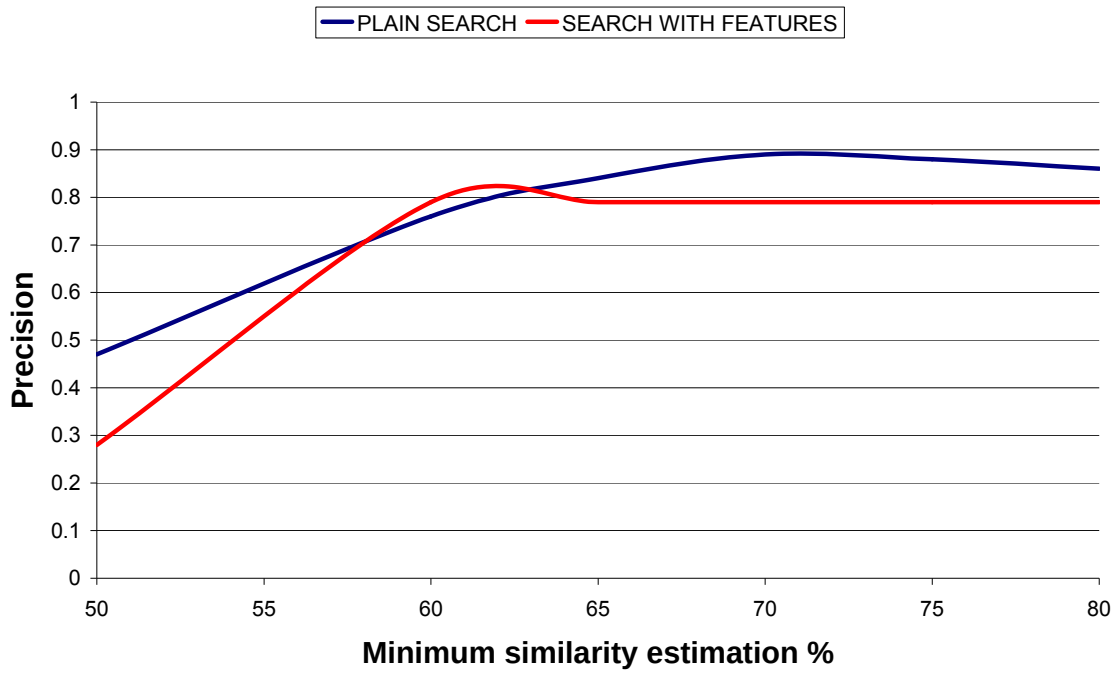

Fig. 30: Precision for long strings sample data set

Also the evolution of this relation is different for both algorithms. As explained previously the phonetic algorithm gives very low values of similarity estimation for misspellings that alter the phonetic of the word and very high values for the ones that do not. This results in a relation between correct matches and wrong ones almost constant from a relatively low value (less than 60%) until too high values, where the total number of correct results provided has decreased to a point where there is a high risk of leaving out of the search correct results, even combining the ones obtained by both algorithms. The evolution of the relation between correct and wrong results, and between correct results obtained and total results present in the data set (Fig. 31 and Fig. 32) is smoother for the algorithm based on edit distance than for the one based on a phonetic algorithm.

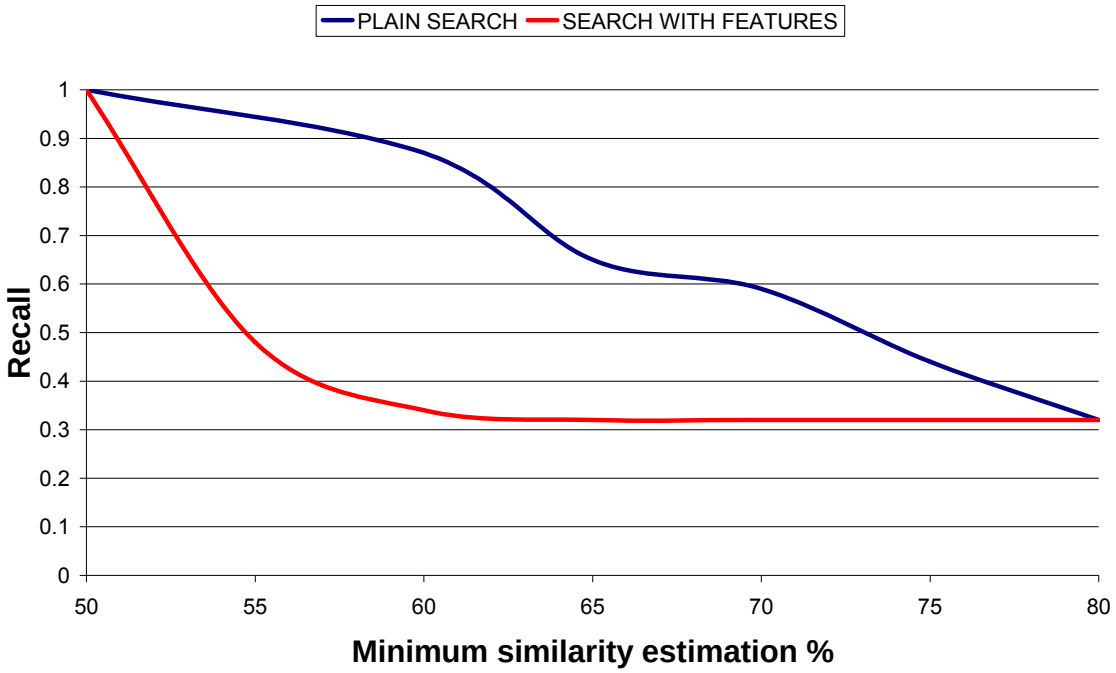

Fig. 31: Recall for short strings data set

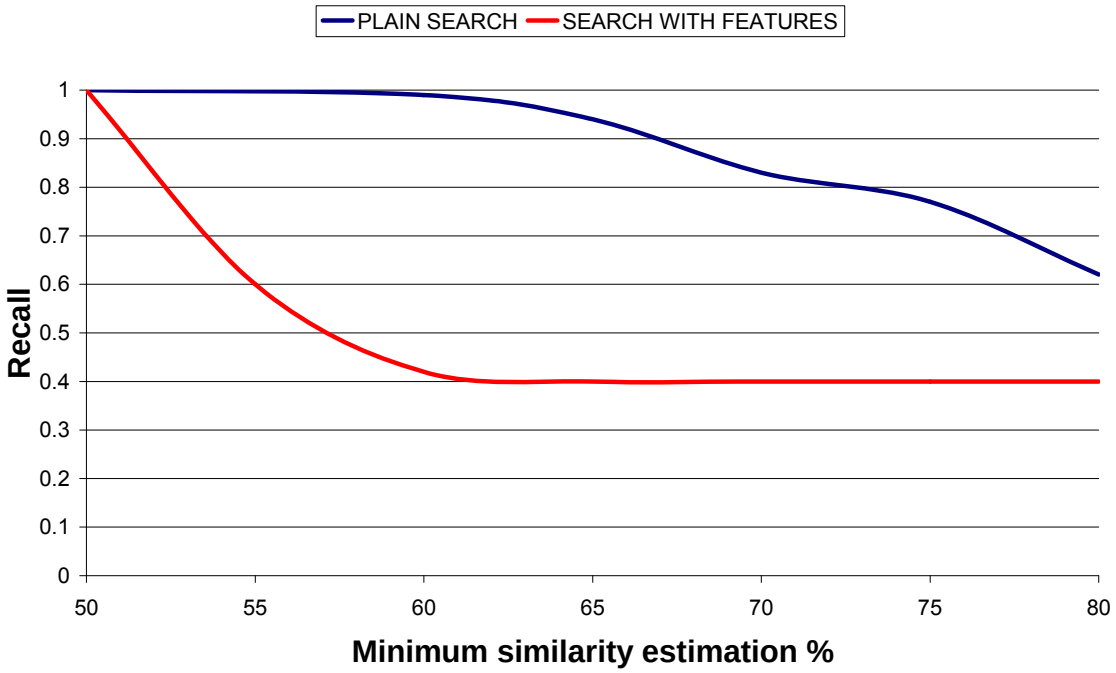

Fig. 32: Recall for long string data set

Considering the results obtained during the tests, the limit value for the minimum similarity estimation should be established at 67.5%. This value doesn't filter all the wrong matches, but simplifies the task of the editors (the harmonization process involves a manual task at the end of the process) reducing their number, without leaving out any of the correct matches if the results from both algorithms are combined.

4. CONCLUSIONS

Considering the overall test results of the approximate string matching algorithms, the comparison has confirmed many prior expectations and revealed a few surprising characteristics of the methods.

For example, it might have been anticipated that the more complex an algorithm is, the more accurate the results obtained. However, this seems not to be the case. The most complex algorithm was the one used for the search with features, which combined a phonetic algorithm (Metaphone) with the Levenshtein algorithm. This algorithm required more execution time than the one based only on edit distance, but was less accurate and tended to determine more names which were unrelated in reality, and to leave out of the results correct ones. It did, however, assign a higher value of similarity estimation to the strings that were related.

Another remarkable characteristic is that the results obtained by both algorithms are complementary. That is, the higher limit value of similarity estimation required for considering a string a possible match, the less total results obtained and more probability of leaving out correct matches. However, the correct matches filtered by both algorithms were different, as they process the strings with different criteria. This is useful, as a high value of limit similarity estimation can be defined, which allows to reduce the total number of possible matches obtained, but assuring not to leave any correct one out if the results from both algorithms are combined. A reduction of the total number of results displayed is useful in terms of the database editor task, who has to make the selection of the correct matches manually.

The length of the strings also affected the results obtained. When working with the algorithm based only on edit distance, the results obtained for the subsample data set of strings with a length higher than 10 letters the accuracy of the algorithm increased considerably. When working with the algorithm based in phonetic features, the results were similar for both subsample data sets.

In terms of execution time, both algorithms were fast enough working with a database with 4650 strings. Certainly any decrease in execution time for an algorithm will be beneficial, providing this does not result in less accuracy as a consequence. In this respect, it is possible to pre-code the strings in the database with Metaphone, whereby each name is first coded using the

algorithm, then the string matching is determined by comparing the codes for each string.

Another possible area of improvement consists of the modification and adaptation of the Levenshtein and the Metaphone algorithms to the results desired. In the Levenshtein algorithm, for example, the costs of the transformation operations can be defined according to the characteristics of the spellings errors present in the database strings (e.g. transpositions can be assigned a lower cost than substitutions, as it is a common typing mistake). Metaphone can be adapted to consider not only the English phonetic rules, and its complexity can be extended considerably concerning the treatment of vowel sounds and first letters of strings. However, concentrating on improvements that handle specific cases will result in a more complicated algorithm which consequently takes longer to execute, and can also lead to contradictory cases.

Furthermore, it could be possible to build a “known matching strings” database, and use it in combination with an approximate string matching algorithm. Unfortunately this is likely to require a very large “known matching strings” database and would be less portable than a simple matching algorithm, adding this problem to the building of the database itself.

The choice of an approximate string matching algorithm would seem to depend largely on the required application and desired results. It is apparent that there is no “best” algorithm ever. For the purpose of the harmonization of databases, the best solution would be to determine a relatively low value for the limit similarity estimation (70%) and combine the results from both algorithms.

Due to the diversity of applications for string-matching techniques, choosing a particular algorithm will depend largely on the nature of the data it is to be applied to. Also to be considered whether the algorithm is in effect employed to provide a comprehensive equivalence class for a particular string, which may include additional incorrect matches, or an accurate list of equivalent strings that may be incomplete.

Many reports on current string-matching techniques quote specific cases of string-pairs that a particular algorithm fails to determine correctly. Although this may highlight certain classes of strings that the algorithm does not perform well on, it must be considered that overall the algorithm may have a high accuracy. The problem seems to arise from the degree of generalisation employed by those algorithms that consider the phonetic structure of the strings being compared.

Since string-matching algorithms are employed basically to avoid lengthy and laborious manual analysis of data, provided they can achieve an acceptable level of accuracy, the time saved by their application is an obvious advantage. Nevertheless, it cannot be expected that such an automated approach to string-matching will achieve 100% accurate results. Even when performing the task manually it is likely that the human-error will cause a certain degree of inaccuracy.

5. ACKNOWLEDGMENTS

The author thanks all the team involved in the GAMA project for all the effort done building the platform and improving every aspect as much as possible. Special thanks to the AGH team, with Andrzej Głowacz, Michał Grega, Lucjan Janowski, Mikołaj Leszczuk and Piotr Romaniak, for helping me with the work and speak English when I was the only non-Polish speaker.

It has been a long time in Krakow and it also would not have been possible without my flatmates: Colline, Jeanne, Eva, Bianca, Camille, Mira, Jeni, Bushka and the 4 unnamed little bushkas. It has really been a great year, and I have made many new friends that I hope will last for all my life: Nacho, Ángel, Pablo, Javi, Raúl, Christian, and many others.

Special thanks to my family, for supporting me when I decided to make the Erasmus, and to Mikołaj Leszczuk, my Master Thesis supervisor, for helping me with the work, orientating me to progress in the good direction.

To end, thanks to all Krakow people because now this city is my second home.

6. REFERENCES

LIST OF WORKS CITED:

[1] Juergen Enge, Pawel Fornalski, Andrzej Głowacz, Michał Grega, Lucjan Janowski, Mikołaj Leszczuk, Michał Lutwin, Piotr Romaniak, Viliam Simko, Zdzisław Papir. *OASIS Archive – Open Archiving System with Internet Sharing*. December 13, 2007

[2] *European Digital Library Project.* <http://www.edlproject.eu/>. September 10, 2008.

[3] *Europeana.* <http://www.europeana.eu/>. September 10, 2008.

[4] Tom Van Court, Martin C. Herbordt. *Text Retrieval: Theory vs. Practice. Families of FPGA-Based Algorithms for Approximate String Matching* Boston University, ECE Dept.

[5] Javier Arruego Muñoz, Ester Llorente San Juan, José Luis Medina Carretero. *FUMAS (FUzzy MAtching System).* Villaviciosa de Odón, Junio de 2007.

[6] Gonzalo Navarro. *A Guided Tour to Approximate String Matching*. University of Chile.

[7] Daniel Yacob. *Application of the Double Metaphone Algorithm to Amharic Orthography*. International Conference of Ethiopian Studies XV.

[8] Brijesh Shanker Singh. *Search Algorithms*. Documentation Research and Training Centre Indian Statistical Institute. Bangalore-560 059.

[9] David A. Chappell. *Enterprise Service Bus*. Copyright © 2004 O'Reilly & Associates.

[10] Michel Deriaz and Giovanna Di Marzo Serugendo. *Semantic Service Oriented Architecture*. University of Geneva, Switzerland. November 10, 2004.

[11] José M. Martínez, Rob Koenen, and Fernando Pereira. *MPEG-7: the generic Multimedia Content Description Standard.* Copyright © 2002 IEEE. Reprinted from IEEE Computer Society, April-June 2002.

BIBLIOGRAPHY:

Sachin Agarwal, Shilpa Arora. *Context Based Word Prediction for Texting Language*. Language Technologies Institute, School of Computer Science, Carnegie Mellon University. 5000 Forbes Avenue, Pittsburgh, PA 15213.

Ricardo Baeza-Yates. *Approximate String Matching*. Center for Web Research www.cwr.cl. Depto. de Ciencias de la Computación, Universidad de Chile.

Ilaria Bartolini, Paolo Ciaccia, and Marco Patella. *String Matching with Metric Trees Using an Approximate Distance*. DEIS - CSITE-CNR, University of Bologna, Italy.

Maxime Crochemore, Christos Makris, Wojciech Rytter, Athanasios Tsakalidis, Kostas Tsichlas. . *Approximate String Matching with Gaps*. Institut Gaspard Monge, Université de Marne-la-Vallée, France, and Computer Engineering & Informatics Department, University of Patras, Computer Technology Institute (CTI), Patra, Greece, and Department of Computer Science, University of Liverpool, Liverpool, L69 7ZF, UK.

Michael Dittenbach, Dieter Merkl, Helmut Berger. *A Natural Language Query Interface for Tourism Information*. Electronic Commerce Competence Center – EC3, Donau-City-Straße 1, A-1220 Wien, Austria. Institut für Softwaretechnik, Technische Universität Wien, Favoritenstraße 9-11/188, A-1040 Wien, Austria.

A. J. Lait and B. Randell. *An Assessment of Name Matching Algorithms.* Department of Computing Science, University of Newcastle upon Tyne.

Dr. Wing Lam, Dr. Venky Shankararaman. *Enterprise Architecture and Integration - Methods, Implementation and Technologies*. Universitas 21 Global. Singapore Management University. Information Science Reference. June 2007

Wilson Wong,Wei Liu and Mohammed Bennamoun. *Enhanced Integrated Scoring for Cleaning Dirty Texts*. School of Computer Science and Software Engineering, University of Western Australia, Crawley WA 6009.

Sun Wu, Udi Manber, Gene Myers, Webb Miller . *Sequence Comparison Algorithm*. Department of Computer Science, University of Arizona. Tucson, AZ 85721. Department of Computer Science,The Pennsylvania State University. University Park, PA 16802. August 1989.

7. APPENDICES

7.1 APPENDIX A. GAMA (GATEWAY TO ARCHIVES OF MEDIA ART)

This chapter introduces the GAMA (Gateway to Archives of Media Art) project. The functioning and the architecture of the platform is presented, giving an overview of its whole structure and working process. Content-based multimedia indexing modules, an important element of the platform, are treated with more detail.

7.1.1 PLATFORM OVERVIEW

Metadata on the user's and archive side follow Dublin Core standards or have been formatted according to proprietary solutions. Database adapters are used to convert existing metadata schemes that are available in the archive to the search and query language of the gateway (mostly Perl and PHP). So a Database Adapter is the translation module between the partner institutions databases and the platform, a software component that translates the existing tables, lists and metadata categories of the partner's databases on a common scheme.

Once the platform is connected with the databases through the Database Adapters, its basic role is to combine and to channel search requests (the interface between DB-Adapters and the platform consists of several PHP and Perl scripts), and prepare search results and relay them to the Graphical User Interface (GUI).

The system architecture enables users to put in a query on the client part and then it forwards the request to all archives using a certain common protocol and metadata mapping, and the server at the archives' ends is implemented to understand the request and offer a common interface to the agent, regardless of the respective underlying local technology and repository formats.

A mapping module is also set up in each archive that defines how a given request that is based on a common language offered in the central interface, is mapped to the local metadata. Once that mapping modules have been installed and configured, the local server knows what metadata to look for in the archive's database and what to return upon a given request.

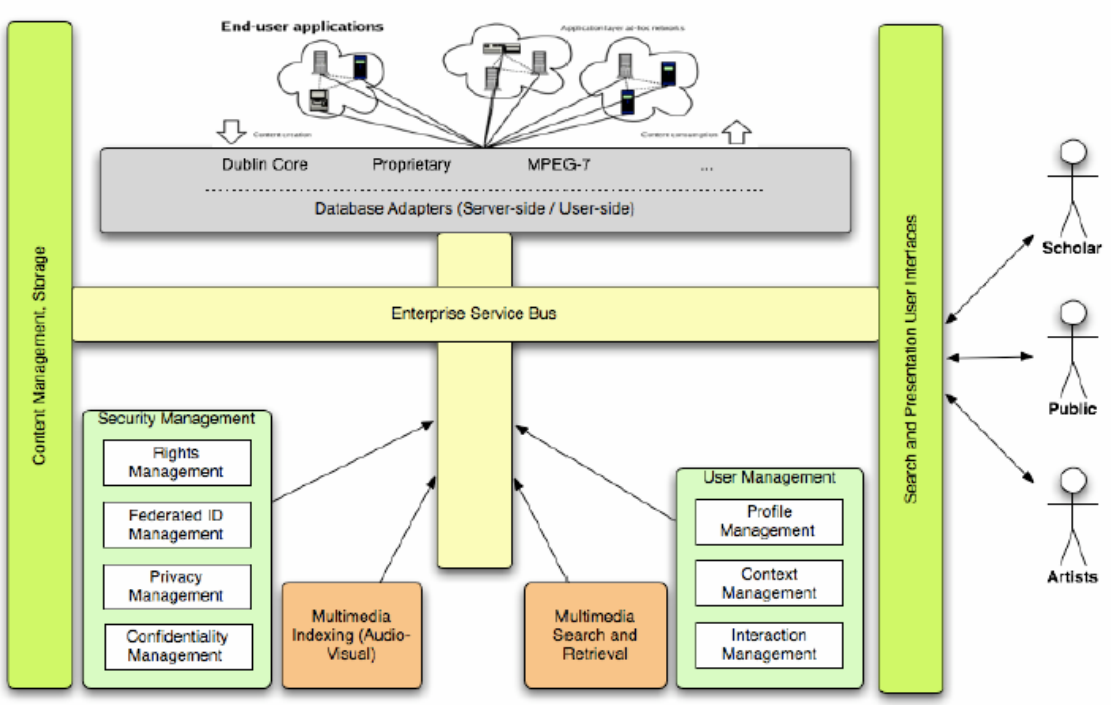

Fig. 33: Platform overview

When a request has been issued, the answers from all archives (or from the archives that the user has selected) are gradually collected and passed on to the user interface for visual representation (Fig.33).

7.1.2 CONTENT-BASED MULTIMEDIA INDEXING

User queries are not only based on text and keywords, but also on visual and audio features that are extracted automatically by some algorithms. These algorithms are coupled with corresponding search methods that allow for search based on visual similarity which has already been applied to video data, images, etc. For audio, similar algorithms that extract audio visual features are used. These include detection of loudness, sudden volume changes, separating of sequences with spoken text and music, etc.

Automatically extracted metadata that is based on image and video indexing feature extracting methods is mostly described in MPEG-7[11], a standard for describing audio-visual content. So MPEG-7 is be the server-side format for audio-visual descriptions.

7.1.2.1 OPTICAL CHARACTER RECOGNITION

Optical Character Recognition (OCR) is computer software which converts images into machine-editable text. An OCR system is capable of recognizing text occurring in video frames like: subtitles, actors' lists, labels, etc. Results are the keywords with their mapping on video time code, where keywords are recognized.

Recognition can be applied to text in any language, but best results can be obtained for English. As the quality of the recognition has to be as high as possible, one idea is to make two or more OCR engines to work in parallel and combine the results. The systems considered to be used in GAMA were Tesseract, Orcopus, and GOCR (a modified version of GOCR text recognition engine, in order to improve performance in the meaning of the time of recognition). The one finally chosen has been Tesseract.

7.1.2.2 AUTOMATIC SPEECH RECOGNITION

Automatic Speech Recognition (ASR) allows one to recognise (to convert to a textual format) words or phrases spoken in any audio-visual content. Similar to OCR, it is possible to get keywords mapped on video time code.

Deployment for languages different from English is problematic, as speech recognition engines work first and foremost for English. With use of commercial engines ASR can be extended to other languages, like French, Spanish, German, Japanese and Chinese, but it requires purchasing optional licenses.

As in the OCR case, the quality of recognition has to be as high as possible and one idea is to make two or more ASR engines to work in parallel and combine the results. The GAMA ASR module is an adapted and modified version of Microsoft Speech SDK speech recognition engine running on Windows.

7.1.2.3 FACE RECOGNITION

Face Recognition (FR) system detects and recognizes persons in video content. The system needs the following components: actors' database, face detection, and face recognition.

First, person faces database need to be created. Then, candidate faces are detected in new video content and compared to faces stored in database. Results are names of the persons mapped on video time code, which provides automatic information on actors' appearances in the video.

The systems that are being considered to use for this purpose are: CSU-FIE, Intel OpenCV, and Open Biometry. As in the case of OCR and ASR, it is also possible to make two or more face recognition engines to work in parallel and combine the results, in order to improve the quality of the detection.

7.1.2.4 IMAGE QUERY-BY-EXAMPLE

The advanced content-based search techniques utilise the MPEG-7 low-level features of the media, such as dominant colour, shape histogram and

similar. This gives the user the opportunity to perform search by providing an example.

The principles of the QbE systems are as follows: the user provides an example of a searched item to the system. The system calculates, on the client side, the features that describe the example. These features are routed through the network to the queried side. The queried side is in possession of the calculated (in advance) features of the media being in its possession. Once received the features of the example, the queried side calculates the distances of the features of the example from the features of the media in its possession (in a defined metric) and returns the list of distances to the querying side.

It uses PictureFinder (see 2.2.6) as a fast pre-filter, to improve the performance of the generation of precalculated result lists that are then exported to the central RDF repository.

7.1.2.5 VIDEO QUERY-BY EXAMPLE

Video QbE offers similar functionality to MPEG-7 Image QbE, but for videos or segments of videos. It combines visual features from MPEG-7 Image with video-specific features like descriptors for motion. It will also be possible to include MPEG-7 audio descriptors extracted from audio tracks or input videos. MPEG-7 Video Query-by-Example is a language-independent functionality.

In order to query by example the user chooses an exemplary shot from the database contents. The system then returns a ranked list of results (video artworks) containing similar shots. Similarly to the Image QbE module result lists are also pre-calculated in the MPEG-7 video module per shot, for improving the speed of the system.

7.1.2.6 PICTUREFINDER (QbE)

PictureFinder implements QbE for still images based on distribution of colour and texture features. It is also possible to perform sketch-based queries based on the extracted features. It is specially optimized for very large image databases, and it is integrated within the MPEG-7 image module. It is also language-independent.

7.1.2.7 SHOT BOUNDARY DETECTION AND KEYFRAME EXTRACTION

The Shot Boundary Detection and Keyframe Extraction module does not directly introduce new search functionalities but extracts basic features for use in combination with other modules. Shots extracted by the Shot Boundary Detection split videos into meaningful segments that can be used, for example, in presentation of results sets.

Keyframe Extraction is based on the result of the Shot Boundary Detection. One or more representative frames are extracted per shot. The Keyframes can be used in the user interface for navigation (e.g. as thumbnails) and can also be used for video QbE: MPEG-7 video and audio features are extracted and matched on a shot basis. Queries for video QbE are shots for which associated key frames can be displayed in the GAMA portal and linked with QbE. Extracted key frames are fed into the MPEG-7 image module for feature extraction and used as a part of video QbE.

7.1.2.8 AUDIO QUERY-BY-EXAMPLE

The MPEG-7 Audio module extracts audio descriptors from the audio tracks or input videos. It would be possible to perform QbE exclusively based on audio descriptors, or do it in addition with other descriptors.

7.1.2.9 SOUND EVENT DETECTION AND AUDIO SEGMENTATION

The Sound Event Detection is a generic component that can be trained to detect certain predefined audio events. The user could then search for video segments where certain audio events occur. Additionally, this could be a feature used for video QbE.

The Speaker & Music Segmentation finds segments with spoken text and/or background music in the audio tracks or input videos. This could be used in combination with ASR and also as a feature for video QbE.

7.1.3 GAMA ARCHITECTURE

Two possible scenarios were considered for implementing the platform: centralised (Fig. 34) or decentralised service for content-based indexing of media data. Finally, an architecture with centralised content-based indexing was chosen.

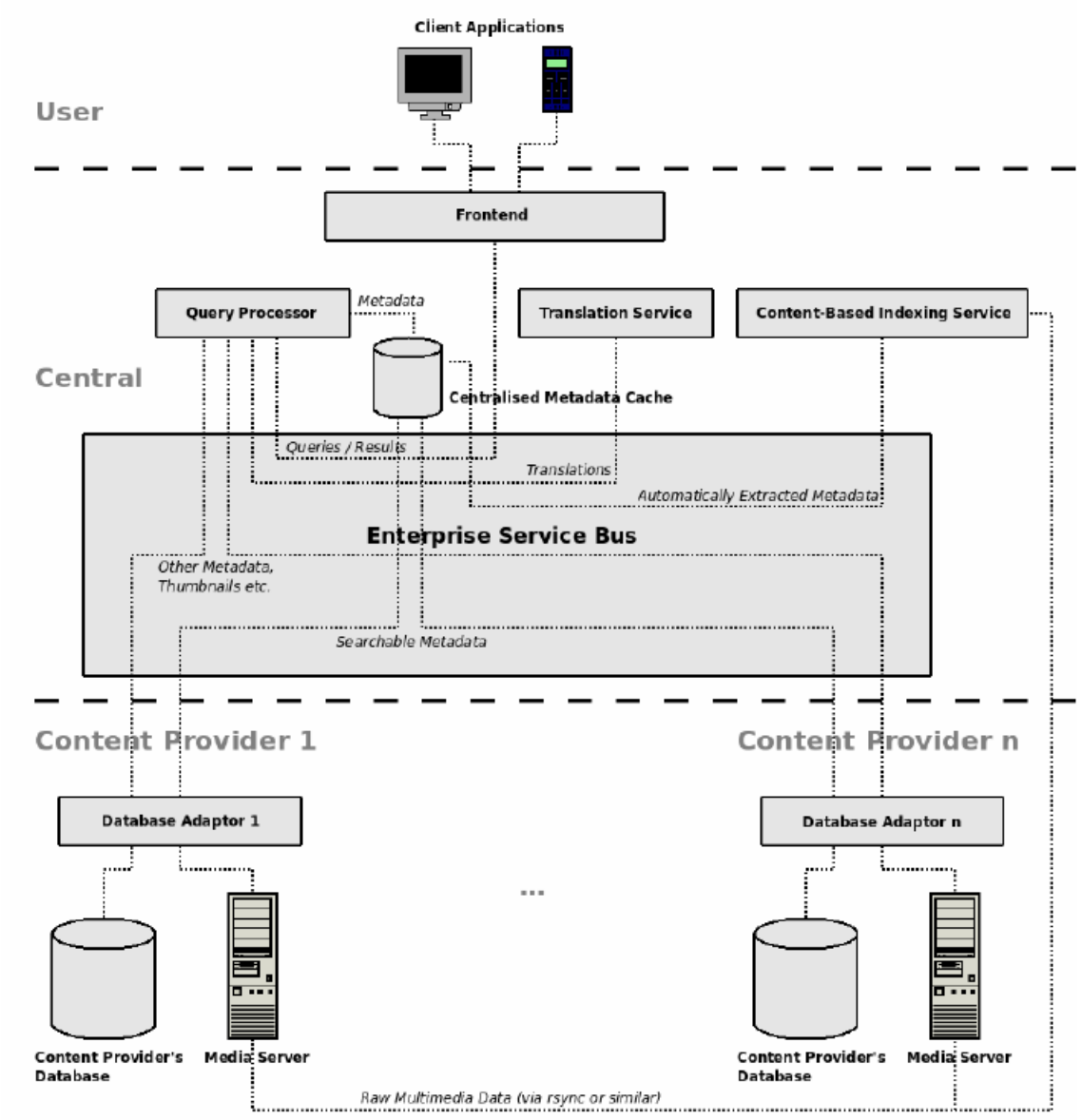

Fig. 34: System architecture with centralised content-based indexing service

There are two phases in the project: the phase 1, with a voluminous amount of content to be analysed, and a more static phase when only new content have to be processed. In a decentralised scenario, the content partners would have to make the data processing locally. This would complicate the installation procedure for the content providers, although it would allow them not to store their data in another server. In a centralised scenario, all the media data will have to be transferred to the central cache. Due to the considerable amount

of data that is in the content providers data bases, this transfer can prove problematic. However, it eliminates any installation problem for the content partners (no installation needed at all), and it also reduces the time of user's queries processing.

In the long term, a centralised architecture offers also advantages regarding scalability and maintenance. Any improvements and updates will only be necessary in the central (if it is the case of commercial components, it would mean the purchase of one license in the centralised case, and of one per archive in the decentralised one). Computational requirements will also change during the platform's life, and for a central indexing service all the required changes would be simpler as well.

7.2 APPENDIX B. LEVENSHTEIN EDIT DISTANCE ALGORITHM

Step	Description
1	Set n to be the length of s. Set m to be the length of t. If $n = 0$, return m and exit. If $m = 0$, return n and exit. Construct a matrix containing $0..m$ rows and $0..n$ columns.
2	Initialize the first row to $0..n$. Initialize the first column to $0..m$.
3	Examine each character of s (i from 1 to n).
4	Examine each character of t (j from 1 to m).
5	If s_i equals t_j, the cost is 0. If s_i doesn't equal t_j, the cost is 1.
6	Set cell $d(i, j)$ of the matrix equal to the minimum of: a. The cell immediately above plus 1: $d(i-1, j)+1$. b. The cell immediately to the left plus 1: $d(i, j-1)+1$. c. The cell diagonally above and to the left plus the cost: $d(i-1, j-1)+cost$.
7	After the iteration steps (3, 4, 5, 6) are complete, the distance is found in cell $d(n, m)$.

7.3 APPENDIX C. VOCABULARY OF TERMS

Alphabet: 1. A set of letters or other signs used in a writing system, each letter or sign being used to represent one or sometimes more than one phoneme in the language being transcribed. **2.** Any set of characters.

Approximate string matching: Approximate string matching is a string matching that allows errors. The objective is to perform a string matching of a pattern in a text where one or both sources have suffered some kind of corruption.

Automatic Speech Recognition: Automatic conversion of spoken words to machine-readable input (for example, to key presses, using the binary code for a string of character codes).

Common Object Request Broker Architecture: Mechanism in software for normalizing the method-call semantics between application objects that reside either in the same address space (application) or remote address space (same host, or remote host on a network).

Distance (between strings): The distance $d(x, y)$ between two strings x and y is the minimal cost of a sequence of operations to transform x into. The cost of a sequence of operations is the total sum of the costs of the individual operations. Operations are defined for each distance function.

Distributed Architecture: Software architectural concept where applications and programs are split up into parts that run simultaneously on multiple computers communicating over a network.

Double-Metaphone: Phonetic algorithm that, given a string, generates two codes. The generation of the codes is based on the phonetic features of the string.

Dublin Core standards: Standard for cross-domain information resource description. It presents a simple and standardised set of conventions for characterizing things online, in ways to make them easier to find.

Endpoint: Entry point to a service, a process, or a queue or topic destination.

Enterprise Application Integration: Use of software and systems architecture principles to integrate a group of applications. In general, they have three components: an integration broker that serves as a hub for intersystem communication and performs some functions like multi-format translation, transaction management, monitoring and auditing. A set of adapters that enables different systems to interface with the integration broker, and an underlying communication infrastructure, such a reliable high-speed network,

which enables systems to communicate with each other using a variety of different protocols.

Enterprise Resource Planning: Enterprise-wide information system designed to coordinate all the resources, information, and activities needed to complete business processes such as order fulfillment or billing.

Enterprise Service Bus: Standards-based integration platform that combines messaging, Web services, data transformation, and routing, to reliably connect and coordinate the interaction of a significant number of diverse applications.

Episode distance: Distance function that allows only insertions, which cost 1.

Face Recognition: Computer application for automatically identifying or verifying a person from a digital image or a video frame from a video source

Fuzzy search: Kind of search that, given a pattern string, finds strings that approximately match that pattern string.

Gateway: Computer or network that allows or controls access to another computer or network.

Hamming distance: Minimum number of substitutions required to change one string into another.

Harmonization: Process of establishing relations between different entries in databases, in order to group the ones that refer to a similar concept.

Image Query-by-Example: Content-based search technique where the user performs a search by providing an example of a searched item (an image).

Information retrieval: Science of searching for documents, for information within documents and for metadata about documents, as well as that of searching relational databases and the World Wide Web.

JAVA: It is a programming language originally developed by Sun Microsystems. The language derives from C and C++, but it has a simpler object model and fewer low-level facilities. Java applications are typically compiled to byte-code that can run on any Java virtual machine, regardless of computer architecture, and it uses object-oriented programming methodology.

Keyframe extraction: Technique that extracts representative frames from video shots.

Levenshtein distance: Metric for measuring the amount of difference between two sequences. It is given by the minimum number of operations needed to

transform one string into the other, where an operation is an insertion, deletion, or substitution of a single character.

Longest Common Subsequence distance: Distance function that allows only insertions and deletions, which cost 1. It calculates the length of the longest pairing of characters that can be established between both strings, so that the order of the letters is respected. The distance is the number of unpaired characters.

Metaphone: Phonetic algorithm that, giiven a string, generates a code of variable length that represents its pronunciation. It is based on Soundex, but the encoding rules are more complex.

Optical Character Recognition system: Computer software which converts images into machine-editable text.

PERL: Interpreted language optimized for scanning text files, extracting information from those text files, and printing reports based on that information. It's also useful for many system management tasks

Phonetic algorithm: Algorithm for indexing of words by their pronunciation.

PHP: Computer scripting language, originally designed for producing dynamic Web pages. It can be used in standalone graphical applications or from a command line interface, but is mainly used in server-side scripting. It is specially suited for Web development and can be embedded into HTML. In most cases it runs on a Web server, where PHP code is taken as its input and Web pages are created as output. PHP acts primarily as a filter, taking input from a stream or a file containing text and/or PHP instructions, and outputs another stream of data (most commonly the output will be HTML).

Precision: Measure for evaluating the quality of results in information retrieval scenarios. It is defined as the number of relevant documents retrieved by a search divided by the total number of documents retrieved by that search.

Recall: Measure for evaluating the quality of results in information retrieval scenarios. It is defined as the number of relevant documents retrieved by a search divided by the total number of existing relevant documents.

Service-Oriented Architectures: Service-oriented architectures are a kind of software architecture that is designed to establish a dynamically organized environment, based on networked services that are interoperable and composable. In a SOA, services are separated from their implementation, using the concept of an interface.

Shot Boundary Detection: Technique that extracts shots from a video. Partitioning a video sequence into shots is the first step toward content-based video browsing and retrieval, and video-content analysis.

Soundex: Phonetic algorithm that convert words to a four-character code. Is the basis for many modern phonetic algorithms.

Sound Event Detection: Technique for detecting certain predefined audio events. It can be used, for example, for searching video segments where cerain audio events occur.

Speaker and Music Segmentation: Technique for finding segments with spoken text and/or background music in audio tracks or videos.

Video Query-by-Example: Content-based search technique where the user performs a search by providing an example of a searched item (for example, the user chooses a shot from a video), and the system returns a ranked list of results (videos) containing similar shots.

Web services: Collection of protocols and standards used for exchange data between applications. The organizations OASIS and W3C are the responsible committees for the architecture and regulating of Web services. Web services enable systems to communicate over the Internet or an intranet, but they present some barriers: the standards are relatively new and continue to evolve rapidly, they are not usually suitable for high-volume transaction processing, and though it may be useful to develop new systems using Web services, existing systems may need to be redesigned to conform to a Web Services model.

7.4 APPENDIX D. LIST OF ACRONYMS

This section provides a list of acronyms, to complement the vocabulary of terms presented in the previous appendix:

AGH: *Akademia Górniczo-Hutnicza*

CIANT: *Internacional Center for Art and New Technologies*

CORBA: *Common Object Request Broker Architecture*

EAI : *Enterprise Application Integration*

EDLproject: *European Digital Library project*

ERP: *Enterprise Resource Planning*

HTML: *HyperText Markup Language*

GAMA: *Gateway to Archives of Media Art*

LCS: *Longest Common Subsequence distance*

SQL: *Structured Query Language*

PHP: *Hypertext Pre-processor*

QoS: *Quality of Service*

SOA: *Service Oriented Architecture*

7.5 APPENDIX E. LIST OF FIGURES AND TABLES