Using Approximate String Matching Techniques to Join Street Names of Residential Addresses

Free University of Bolzano-Bozen
Faculty of Computer Science
Bachelor of Science in Applied Computer Science

Roland Innerhofer-Oberperfler
Roland.Innerhofer-Oberper@unibz.it

Student No. 1135

Bachelor Thesis
Academic Year 2003/2004
1st Graduate Session - July 30, 2004

Supervisor Prof. Johann Gamper
Tutor Nikolaus Augsten

July 15, 2004

Abstract

For many administrative tasks at the Municipality of Bolzano-Bozen a number of autonomous databases have to be accessed. In order to compute these tasks more efficiently, the content of these databases should be linked automatically. A promising join attribute are residential addresses, as they appear in more or less all databases. Standard join techniques give poor results for joining address data. Reasons are spelling mistakes, abbreviations, and different naming conventions for street names. The main goal of this work is to overcome these problems by using approximate string matching techniques in order to find good matches for street names.

In this thesis I analyze the accuracy and efficiency of two approximate string matching algorithms for matching street names: q-gram and edit-distance. This analysis is based on experiments using street names of residential databases at the Municipality of Bolzano-Bozen, and on experiments using virtual test sets. Moreover, I present two algorithms to match two sets of street names. These algorithms use the string matching algorithms to calculate the distance between strings. Further, I present a tool, which uses the string matching techniques to match street names from input-databases. This tool provides the user with a good overview of the matches found in the input-databases.

Contents

1 Introduction

A typical situation found in a public administration environment is a number of autonomous databases which store information about the same real-world objects. For example, the registration office, the cadastre, and the electric power company all store data about citizens and apartments. Although they speak about the same objects, they are usually interested in different relations between them, e. g. the registration office stores who lives in an apartment, while the electric power company is interested in who pays the electricity bill for it [ABG04].

The Municipality of Bolzano-Bozen has many autonomous databases, where the content of the databases is very similar, but not equal. For several tasks, these databases have to be joined.

Searching corresponding entries in different databases manually is tedious and often not feasible.

Joining the addresses with standard join techniques (SQL-join) ends up with very poor results, as only exact matches are found. The problem hereby is, that some databases don't have a common key, that could be used as a join attribute. The most common attributes found in the tables of the databases are the residential addresses. So, to get virtually one database, there has to be done a join on addresses.

In my work, I concentrate on street names. I match the street names using approximate string matching techniques to overcome problems like spelling mistakes, different spellings or abbreviations. In my thesis I use the edit distance and the q-gram approximate string matching algorithms.

The paper is organized as follows. Section 2 describes the problems and objectives of the Municipality of Bolzano-Bozen. In Section 3 the used approximate string matching techniques are presented in detail. In Section 4 I show how street names can be matched using the former introduced approximate string matching algorithms. Section 5 presents, how test data is generated. Section 6 describes how the algorithms and the method to match street names were implemented. In Section 7 I evaluate the accuracy of the string matching algorithms and of the algorithms to match the street names. Section 8 draws conclusions and points to future work.

2 Problem Description

Many tasks at the Municipality of Bolzano-Bozen require to access data from different, autonomous databases, e. g. databases of the cadastre, the electric power company, etc. Those databases sometimes hold the same or approximately the same data. However, the databases do not always have key values, which makes it hard to link their contents. The most relevant common attribute, that is stored in the tables of the databases, are residential addresses.

Unfortunately, addresses have some peculiarities. First, they are referenced with a different granularity level. The same apartment might be referenced by street name, house number, and apartment number in one database, but only by street and house number in another one. Second, streets might have been renamed and this change has not been updated in all databases. Finally, different databases might use different labels to enumerate apartments. This makes it very difficult to link data across databases using standard join techniques [ABG04].

2.1 Current Situation

If someone needs to retrieve data from different databases, there are several ways to get the data from the autonomous databases. One is to link the needed databases with standard join techniques (SQL-query). As addresses are the most relevant common attribute, a join on addresses is done. The disadvantage of using this technique is, that not all data can be matched. The reasons for this are manifold:

- spelling mistakes ('Keppler' instead of 'Kepler')
- abbreviations ('Giuliani P. R.' in one database, 'Padre Reginaldo Giuliani' in another)
- additional data ('piazza', 'galleria', 'via' stored together with the street name)
- data do not exist (entry in one database, which is not in the other)

For this reasons, manual lookup is currently the only method to link data that cannot be matched with SQL-joins. This is a very time-consuming and tedious task and does not always lead to good results.

The lookup can only be done with exact string matching or with pattern matching. The problem hereby is that someone who searches for an entry, might not know how an entry was abbreviated or written. Even with regular expressions in SQL-joins an entry might not be found due to a different spelling.

Concluding, neither SQL-joins nor manual lookup solve the problems of getting corresponding data from different databases.

2.2 Objectives

The main objective of the thesis is to match street names of residential addresses using approximate string matching techniques. This matching will be used to join data across the autonomous databases of the Municipality of Bolzano-Bozen using complete residential addresses as join attribute.

To reach the objective I implement two string matching algorithms. In addition I design a method which allows to match street names, using these approximate string matching algorithms. Furthermore, this method supports also the use of any other string matching algorithm.

The evaluation is based on the real street names of the residential addresses and also on virtual test sets. A generator for virtual test sets has also to be implemented. The virtual test sets should have similar properties to the real street names, in order to be relevant test sets.

3 Approximate String Matching Techniques

In this section I describe, which approximate string matching algorithms were used and how they work. For this project I use 2 approximate string matching techniques, namely q-gram and weighted Levenshtein distance (shortly edit-distance).

Let $\sum$ be a finite alphabet of size $|\sum|$. I use lowercase Greek symbols, such as σ, possibly with subscripts, to denote strings in $\sum^*$. Let $\sigma \in \sum^*$ be a string of length n. I use $\sigma[i...j], 1 \leq i \leq j \leq n$, to denote a substring of σ of length $j - i + 1$ starting at position i. λ denotes the *empty string*[GIJ+01b].

3.1 Weighted Levenshtein Distance

The *Levenshtein distance* – also known as *edit distance* – between two strings is the minimum number of edit operations (i. e., *insertions, deletions,* and *substitutions*) of single characters needed to transform the first string into the second [GIJ+01b].

An *edit operation* is a pair $(a, b) \in (\sum \cup \{\lambda\}) \times (\sum \cup \{\lambda\}) \setminus \{(\lambda, \lambda)\}$. It is usually written as $a \rightarrow b$. An *alignment* A of two strings σ_1 and σ_2 is a sequence $(a_1 \rightarrow b_1, ..., a_h \rightarrow b_h)$ of edit operations such that $\sigma_1 = a_1...a_h$ and $\sigma_2 = b_1...b_h$. A *weight function* δ assigns to each edit operation $a \rightarrow b, a \neq b$ a positive real weight $\delta(a \rightarrow b)$. The weight $\delta(a \rightarrow a)$ of an edit operation $a \rightarrow a$ is 0. If $\delta(a \rightarrow b) = 1$ for all edit operations $a \rightarrow b, a \neq b$, then δ is the *unit weight function*. The weight $\delta(A)$ of an alignment A is defined by $\delta(A) = \sum_{a \rightarrow b \in A} \delta(a \rightarrow b)$. The *weighted edit distance* of σ_1 and σ_2 is the minimum possible weight of an alignment of σ_1 and σ_2 [Kur96].

Example 1 *Two strings, σ_1 = 'Galeria' and σ_2 = 'Galleria', are compared with each other. The alignment A of the two strings is:*

$$A = (G \rightarrow G, a \rightarrow a, l \rightarrow l, \lambda \rightarrow l, e \rightarrow e, r \rightarrow r, i \rightarrow i, a \rightarrow a)$$

The weight $\delta(A)$ is then achieved by summing the costs $\delta(a \rightarrow b)$ of all edit operations $a \rightarrow b$ (where a and b are two characters). To get two equal strings, the operation $(\lambda \rightarrow l)$ has to be carried out. In this case this operation is an insertion, and has a cost of $\delta(\lambda \rightarrow l) = 1$. For all other edit operations the cost is $\delta(a \rightarrow b) = 0$, as $a = b$. So the weight $\delta(A) = 1$. If weighted differently, the results also differ.

Let *cins* be the cost of insertion, *cdel* the cost of deletion, and *csubst* the cost of substitution. An algorithm for the calculation of the weighted edit distance of two different strings σ_1 and σ_2 can be seen in figure 1 on page 5. The time-complexity of the algorithm is $O(|\sigma_1| * |\sigma_2|)$, i. e. $O(n^2)$ if the length of both strings is about 'n' [All99].

```
function edit-distance(σ1, σ2)
// Input: two strings σ1 and σ2
// Output: similarity between σ1 and σ2 in cost of operations
// (the smaller the value, the better the match)
Create 2 vectors m1 and m2 of length |σ1|
for i ← 0, ...|σ1| do
    Initialize m1 with i * cdel
endfor
for j ← 0, ...|σ2| do
    m1(0) = m1(0) + cins
    for i ← 0, ..., |σ1| do
        if σ1(i) = σ2(j) then
            v = m1(i)
        else
            v = m1(i) + csubst)
        endif
        m1(i + 1) = min(m1(i + 1) + cins, m2(i) + cdel, v)
    endfor
    Exchange vectors m1 and m2 (m1 gets the m2-values and vice versa)
endfor
return m1(|σ1|)
endfunction
```

Figure 1: Weighted Levenshtein algorithm

3.2 Q-Grams

Given a string σ, its *q-grams* are obtained by "sliding" a window of length q over the characters of σ. Since *q-grams* at the beginning and the end of the string can have fewer than q characters from σ, I introduce a new character "`#`" which is *not* in $\sum$, and conceptually extend the string σ by prefixing and suffixing it with $q-1$ occurrences of "`#`". Thus, each q-gram contains exactly q characters, though some of these may not be from the alphabet $\sum$ [ST95, Ukk92, Ull77]. G_σ denotes the list of all the $|\sigma|+q-1$ q-grams of σ.

Example 1 *We get the q-grams of length* $q = 3$ *for string* σ_1 = `'Street'` *by first prefixing and suffixing the string with* `'##'`*. By sliding a window of length 3 over the resulting string* `'##Street##'` *the following q-grams can be constructed:* G_{σ_1} = {`##S, #St, Str, tre, ree, eet, et#, t##`}

The intuition behind the use of q-grams as a foundation for approximate string processing is that when two strings σ_1 and σ_2 are within a small edit distance, they share a large number of q-grams [GIJ+01a, Ukk92, ST95]. The following example illustrates this observation.

Example 2 *The q-grams of length* $q = 3$ *for string* σ_1 = `'Street'` *are listed above. Similarly, the q-grams of length* $q = 3$ *for string* σ_2 = `'Steret'`*,*

are G_{σ_2} = {`##S, #St, Ste, ter, ere, ret, et#, t##`} . *The two q-gram-lists* G_{σ_1} *and* G_{σ_2} *have 4 q-grams in common. This corresponds to a 50 %-match.*

The q-gram distance $qgram(\sigma_1, \sigma_2)$ between two strings σ_1 and σ_2 is defined as follows:

$$qgram(\sigma_1, \sigma_2) = \frac{1}{2}\left(\frac{|G_{\sigma_1} \bigcap G_{\sigma_2}|}{|G_{\sigma_1}|} + \frac{|G_{\sigma_1} \bigcap G_{\sigma_2}|}{|G_{\sigma_2}|}\right) \quad (1)$$

An algorithm for the calculation of the matching percentage (*matching_quota*) of two different strings σ_1 and σ_2 can be seen in figure 2 on page 6. Equal strings have a *q-gram distance* of 1 (= 100 %). The smaller the matching percentage, the more different the strings are.

function *qgram*(σ_1, σ_2)
// Input: two strings σ_1 and σ_2
// Output: matching-quota between σ_1 and σ_2 in %
$match = 0$
$i = 0$, $j = 0$
Generate the list G_{σ_1} from σ_1 and the list G_{σ_2} from σ_2
Sort G_{σ_1} and G_{σ_2}
while ($i < |G_{\sigma_1}|$ and $j < |G_{\sigma_2}|$)
    if $G_{\sigma_1}(i) = G_{\sigma_2}(j)$ then
        $match++$
        $i++$
        $j++$
    else if $G_{\sigma_1}(i) < G_{\sigma_2}(j)$ then
        $i++$
    else
        $j++$
    endif
endwhile
return ((match / $|G_{\sigma_1}|$ + match / $|G_{\sigma_2}|$) / 2)
endfunction

Figure 2: Q-gram algorithm

To sort the lists G_σ takes $O(n \log n)$, with $n = max(|G_{\sigma_1}|, |G_{\sigma_2}|)$. Going through the resulting sorted lists to find the matches takes $O(n)$.

4 Matching Street Names with Approximate String Matching

This section shows how the former introduced approximate string matching techniques can be used to match the street names of the residential addresses.

Let R_1 and R_2 be the set of all street names of 2 different databases. $R_1.\sigma_1$ refers to the string σ_1 of set R_1. Therefore $R_1.\sigma_1$ and $R_2.\sigma_1$ are not necessarily equal. $d(\sigma_1, \sigma_2)$ is the distance of two strings σ_1 and σ_2.

To match the street names I have designed two different methods. One method is used to get *only the best* matches. With this method, there is the possibility that not all matches are covered. The other method is used to get all *best and equal-best* matches. It is more complicated and it may take longer to collect all matches. However, with this method it is guaranteed that all possible matches are found.

4.1 Matching Only the Best Matches

In this section I introduce a method, which is used to get matches of street names, which is not completely accurate, but works quite fast and gets the most of the matches. It is guaranteed that the very best matches are collected.

Algorithm

Let R_1 and R_2 be 2 different data sources. Let L_p be the vector, where all real matches are saved. Let L be a list, which contains the last found best matches and T a temporary match. At the beginning L is empty.

The algorithm starts with the first string of R_1 and compares it with any other string of R_2. The best matches found have then to be approved to be valid matches. The approval is done by comparing the found matches with all strings of R_1. If a better match is found, the former match is discarded. If all matches are discarded, then there does not exist a match for the first string of R_1. If all former matches are approved, the matching continues with the next string of R_2, which in this case would be the second one.

The algorithm to collect *only the first best* matches is shown in figure 3 on page 8.

Example 1 *Table 1 on page 8 shows a distance matrix of two sets R_1 and R_2. Distance matrix in this sense means a matrix, which holds the matching quotas in percentage of different strings (σ_i) when compared with each other.*

Taking table 1, the matching of street names would work as follows: First $R_1.\sigma_1$ is compared with any other string of R_2. If a match is calculated it is also saved in a separate distance-matrix. The best match found, which

```
matchingStreetNames(R1, R2)
// Input: two string sets, R1 and R2,
//        which hold street names
// Output: a list of all found matches (= the set Lp)
for i ← 1, ..., |R1| do
    L = empty
    for j ← 1, ..., |R2| do
        T = match(R1(i) with R2(j))
        if T better or equal to L then
            if T is better than L then
                L = empty
            endif
            add T in L
        endif
    endfor
    for j ← 1, ..., |R1| do
        T = match(R1(j) with T)
        if T better than L(0) then
            continue with next i
        endif
    endfor
    add all entries of L in Lp
endfor
return Lp
endfunction
```

Figure 3: Algorithm to match only the first best match

$R_1 \ddots R_2$	σ_1	σ_2	σ_3	σ_4
σ_1	70	60	0	20
σ_2	<u>90</u>	80	10	10
σ_3	10	10	30	<u>50</u>
σ_4	0	0	<u>80</u>	20
σ_5	0	0	<u>80</u>	5

Table 1: Collecting only the best matches

is $(R_1.\sigma_1|R_2.\sigma_1)$ *with 70 %, has to be approved, so that it is a valid match. Therefore, it is now compared with any other string of* R_1*.* $(R_1.\sigma_2|R_2.\sigma_1)$ *turns out to be a better match with 90 %, so the former found match is discarded and will not be compared anymore.*

So the next string of R_1*, which is* $R_1.\sigma_2$ *is again compared with any other string of* R_2*. The best match for* $R_1.\sigma_2$ *is now* $(R_1.\sigma_2|R_2.\sigma_1)$ *with 90 %. Again this match is approved to be a valid match, by searching a better match for it in* R_2*. As no better match can be found* $(R_1.\sigma_2|R_2.\sigma_1)$ *is*

saved in L_p.

The calculation of the matches for the other strings is done in the same way. The values of the matches are underlined in table 1.

4.2 Matching all Best and Equal-Best Matches

In this section I introduce a method, which is used to get all possible matches.

Algorithm

Let R_1 and R_2 be 2 different data sources storing street names. Let L_b be a vector, which is used to temporarily save the best matches. Let L_p be the vector, where all valid matches are saved.

In order for a match to be valid, the following conditions have to be true:

- the matching-quota is better than the user-defined matching-boundary
- the matching-quota is the best matching-quota for $R_1.\sigma$ and for $R_2.\sigma$ when compared with all σ of $R_1 \setminus L_p$ and $R_2 \setminus L_p$

The calculation of the matches works as follows: First I take a string σ of R_1. This string σ is then compared with every other string of R_2 ($d(\sigma, R_2(1...|R_2|))$). If all R_2-entries were compared with σ, the resulting match(es), if any, have to be verified to be valid matches. To verify the found matches, I use all found matches for σ and compare them with almost all strings of R_1.

If now no better match is found, the found matches for σ are all valid matches and can be saved. However, if at least one better match is found, all former matches for σ are invalid and therefore not saved. The matches, which are then found, which are better than the matches for σ, again have to be verified. The verification is done as described above with the difference, that now R_2 is searched for better matches.

A better match for σ is not searched until the former illustrated recursion ends. It may happen, that with many recursions σ will be matched without returning from a recursion, but then it was matched due to the fact, that another string was searched for a better match and not σ.

The algorithm to collect *all* matches is shown in figure 4 on page 10.

Example 1 *In table 2 on page 10 there is an example of how this method works. To have a comparison of the two methods I use the same distance-matrix for this example. The underlined values are the finally found matches which are different from the previous example.*

Taking table 2, the matching of street names would work as follows:

First $R_1.\sigma_1$ is compared with any other string of R_2. When a match is calculated it is saved in a separate distance-matrix. The best match found,

```
matchingStreetNames(L_b, R_1, R_2)
// Input: the best matches found until now and two string sets,
//        R_1 and R_2, which hold street names
// Output: a list of all found matches (= the set L_p)
for i ← 1, ..., |L_b| do
    for j ← 1, ..., |R_1| do
        if R_1(j) = null then continue
        if R_1(j) is better than L_b(i) then
            empty L_b and save R_1(j) in L_b
        else if R_1(j) is equal to L_b(i)
            add R_1(j) in L_b
        endif
    endfor
    if no better or equal match was found then
        add all L_b in L_p
        delete all corresponding entries of L_b in R_1 and R_2
        if (|R_2| = 0 or |R_1| = 0) then
            return L_p
        else
            L_b = empty match
            matchingStreetNames(L_b, R_1, R_2)
        endif
    else
        matchingStreetNames(L_b, R_2, R_1)
    endif
endfor
endfunction
```

Figure 4: Algorithm to match all best and equal-best matches

$R_1 \ddots R_2$	σ_1	σ_2	σ_3	σ_4
σ_1	70	<u>60</u>	0	20
σ_2	<u>90</u>	80	10	10
σ_3	10	10	30	<u>50</u>
σ_4	0	0	<u>80</u>	20
σ_5	0	0	<u>80</u>	5

Table 2: Collecting all best and equal-best matches

which is $(R_1.\sigma_1|R_2.\sigma_1)$ *with a matching of 70 % has to be approved, so that it is a valid match. Therefore, it is now compared with any other string of* R_1*.* $(R_1.\sigma_2|R_2.\sigma_1)$ *with a matching of 90 % turns out to be a better match, so the former found match is discarded.*

$(R_1.\sigma_2|R_2.\sigma_1)$ *again has to be approved, so it is searched a better match for* $(R_1.\sigma_2|R_2.\sigma_1)$ *in* R_2*. As there is found no better match,* $(R_1.\sigma_2|R_2.\sigma_1)$

is saved as a valid match, with a matching quota of 90 %. From now on, $R_1.\sigma_2$ and $R_2.\sigma_1$ are not used anymore, when a better match is searched.

The next string, which is compared with all other strings is again $R_1.\sigma_1$. As $R_1.\sigma_1$ was already compared with all other strings of R_2, the best match is found very quickly, as every comparison of strings is saved in the distance-matrix.

The best match for $R_1.\sigma_1$ is now $(R_1.\sigma_1|R_2.\sigma_2)$ with a matching of 60 %. Again the match has to be approved and is therefore compared with any other string of R_1.

This procedure of finding the best matches continues until a match for all strings is found. The resulting matches are indicated in figure 4 by underlined matching values.

Concluding, with this method $R_1.\sigma_1$ and $R_2.\sigma_2$ are matched together, even if there would be a better match for $R_1.\sigma_1$ and a better match for $R_2.\sigma_2$. Due to the fact, that the better matches for $R_1.\sigma_2$ and $R_2.\sigma_1$ are worse than $(R_1.\sigma_2|R_2.\sigma_1) = 90\%$ the match of $(R_1.\sigma_2|R_2.\sigma_1) = 60\%$ is a valid match.

5 Random String Generator for Experiments

I've implemented a tool to generate virtual data for my experiments. It can be used to produce a set of random strings or corresponding strings for a given set of strings.

This generated strings are needed as test data to prove that the designed and implemented method to match the street names (section 6.2) is valid and that it works also with other data than the street names of the residential addresses.

The method I have implemented for this purpose supports a lot of different parameters. It is possible to set maximum and minimum length of the generated strings. Moreover it is possible to say with which distribution the length should be calculated. For example, there is the possibility of choosing Gaussian distribution to set the length of the strings to be generated. This was introduced due to the fact that the real street names have a minimum length of about 10 and a maximum length of about 30. However, the most common length is around 12-16 (see figure 5). It is also possible to generate strings with uniformly distributed length.

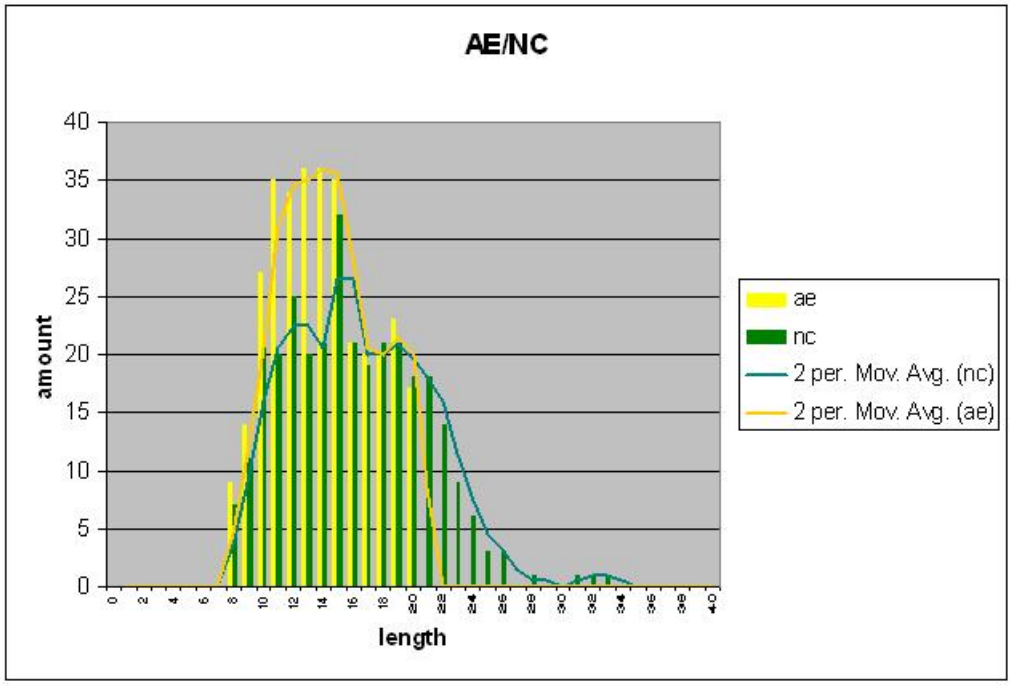

Figure 5: Distribution of the length of real street names (AE and NC are two sets of street names)

In addition to the length, also the distribution of characters can be in-

fluenced. For this purpose there exist two options:

1. the alphabet is equally weighted, so every character has the same probability to appear;

2. the alphabet is weighted, so that every character has a different probability to appear.

The default weights for the second method were determined by analyzing the two sets of street names that were used for the real world experiments.

By generating strings with Gaussian random length and with a distribution of characters as found in street names, good test data can be produced. Moreover, not only street-name-like strings can be produced, but also random strings following other distributions, which is helpful for testing the matching algorithms of section 3.

To have a list of corresponding string pairs, I added a functionality which allows to manipulate a string. The manipulation is done randomly. The user sets only a boundary of manipulations to be carried out on the strings.

The actual amount of manipulations are random, but less then the set boundary. It could be that also no manipulation is done on the string. The manipulations which are carried out are also random and consist of the following: insertion, deletion, and substitution of characters.

So with the latter method it is possible to generate string pairs which are completely different, which are very similar, or which are something in between. These generated string pairs are good test data to prove the matching of street names.

A typical output of the tool can be seen in figure 6 on page 13.

LINE_NUMBER	INITIAL	APPROX
1	tnsiigealaptpvaieaoaitidfe	tvnsiigalapnpvviedaroaoitceddf
2	venudatgaaaooaerinamosb	venuditoaaaooaerinamosb
3	ibiivcodoaotizidievezotai	iabinvcodtolvvilzidieezotai
4	pscpieaglvsvaaucoqenbraa	pslcpieaehlviavaaucoqenbra
5	zoadsivvmnzfilviptafaipeco	olintvvnafliitetfaipico
6	loioaiacvezlilzeanolovbbeg	olodiaiacvezlilznonolovbbeg
7	lozbtsvrpfovglozinoeltad	lztsvrpfoolozinaoeltad
8	airidavltlarladcfaaunvaav	tiridallarlacfaaunvaiv

Figure 6: Output of the string generating tool: random string pairs with at most 20 differences

6 Implementation

This section gives an insight into the realization of the project, explaining basic functionalities, as well as implementation details. This section starts with the implementations of the approximate string matching techniques, *q-gram* and *edit distance*. Then an interface is introduced, which is useful to use any kind of string matching algorithm for matching strings, i. e. street names. Immediately after the interface the method and the accompanying tool of matching the street names is presented. Finally, this section contains also the code for the class, which creates virtual data for experiments, namely random strings, or random string pairs, which can be used as input data for the former mentioned tool.

6.1 Approximate String Matching

In this section the code of the two approximate string matching techniques, *q*-gram and *edit distance*, as well as an interface to use any kind of string matching technique when matching streets, are presented.

6.1.1 Interface

So that the mentioned algorithms, *q-gram* and *weighted Levenshtein* distance, and also every other string matching algorithm can be used to match street names, an interface was implemented. This interface contains all methods needed for a string matching algorithm, when used to match street names.

The interface has the following methods:

Listing 1: Overview of the methods of the interface `StringMatching`:

```
public double distance(String, String)
public double betterDistance(String, String, double)

public boolean isWorse(double, double)
public boolean isWorseOrEqual(double, double)
public boolean isBetter(double, double)
public boolean isBetterOrEqual(double, double)

public double getBoundaryValue()
public void setBoundaryValue(double)
public boolean isValid(double)

public double getBestValue()
```

The method `distance(String,String)` is the most important method of the interface. It should return a distance between the two string-parameters. To allow using any kind of distance, for example either percentage or edit-distance, the methods `isBetter(double,double)` and `isBetterOrEqual(double,double)` are implemented to specify whether a

distance is better or worse than another. With this methods any kind of string matching algorithm can be reused, when matching data-arrays (not only 2 strings).

`betterDistance(String,String,double)` is still experimental. This method should have the functionality of avoiding the comparison of two strings, if it can be calculated (using the length of the strings), that a comparison does never reach a better distance than the `double`-parameter.

The last 4 methods are not needed to match 2 strings. However, they are useful to match whole data-arrays, which both could hold more than 1 string, i. e. this is the case when matching the street names. `setBoundaryValue(double)` and `getBoundaryValue()` are used to set and get the boundary value for a string matching to be valid. This boundary value is used by `isValid(double)`, which has the purpose to limit the found matches. The limitation is done by collecting only the matches, which are better than the boundary value. `getBestValue()` was not used until now. It should return the value, when both strings are equal, ignoring case sensitivity.

6.1.2 Q-Grams

Section 3.2 already shows how the q-gram-algorithm is defined and how it works. In this section I show some peculiarities of the implemented method.

The q-gram-algorithm was implemented in the class *StringMatching_QGram*. The class *implements* the interface *StringMatching*, so it can be used when matching the street names. For an overview of what this class can do, see section 6.1.1.

This class has the following constructors, with which different parameters can be set:

Listing 2: Constructors of `StringMatching_QGram`:

```
public StringMatching_QGram()
public StringMatching_QGram(int)
public StringMatching_QGram(double)
public StringMatching_QGram(int, double)
```

The first constructor is the default constructor. It can be used to create a string-matching-object with $q = 3$ and with a minimum allowed match of $-0,01$.

The other constructors are used to set a user-defined value for q and/or to set a user-defined value for the minimum allowed boundary. The minimum boundary has only sense for matching sets of strings, i. e. the street names. If a match is less than the minimum boundary, it is not recognized as a valid match and is therefore not saved (see section 4).

6.1.3 Weighted Levenshtein Distance

Section 3.1 already shows how the *edit distance* algorithm is defined and how it works. In this section I show some peculiarities of the implemented method.

The *edit-distance* algorithm was implemented in the class *StringMatching_WLD*. The class *implements* the interface *StringMatching*, so it can be used when matching the street names. For an overview of what this class can do, see section 6.1.1.

In order to make the distance symmetric with different weight of operations, I calculate:

$$distance = \frac{1}{2}\Big(rightDistance(\sigma_1, \sigma_2) + rightDistance(\sigma_2, \sigma_1)\Big)$$

`rightDistance` in this case is the standard method to calculate the distance of two strings $\sigma_1 \rightarrow \sigma_2$. If unit weighted, the edit distance can be calculated, by simply using `rightDistance` once.

The class has the following constructors with which different parameters can be set:

Listing 3: Constructors of `StringMatching_WLD`:

```
public StringMatching_WLD()
public StringMatching_WLD(int, int, int)
public StringMatching_WLD(double)
public StringMatching_WLD(int, int, int, double)
```

The first constructor is the default constructor. It can be used to create a string-matching-object with the following weight of operations: $insertion = 1$, $deletion = 1$ and $substitution = 1$. The maximum amount of operations is `Double.MAX_VALUE`.

The other constructors are used to set a user-defined value for the weight of operations and/or to set a user-defined value for the maximum amount of allowed operations. The maximum amount of operations has only sense for matching arrays of strings, i. e. the street names. If a distance is greater than the maximum amount of allowed operations, it is not recognized as a valid distance and is therefore not saved as a match (see 4).

6.2 Data Linking - Matching Street Names with Approximate String Matching

In this section I present a tool (`Data Linking`) which I've designed to match street names of different data sources (e. g. databases or text files). How the matching is done is written in section 4 on page 7.

When starting the tool, the main window appears. The main window consists of two parts: the menubar and the quick menu.

The quick menu has a particular role. It changes after the user carries out an operation. If, for example, a set of street names is loaded into "data(1)", either from file or from a database, the text on the first button changes to "Show Data (1)" (figure 7).

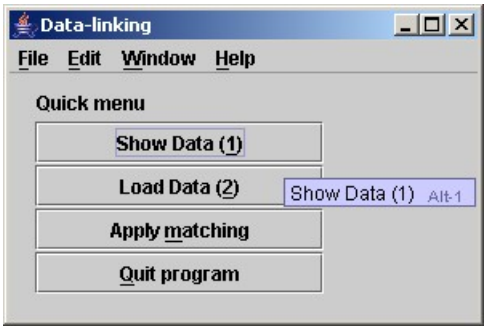

Figure 7: The quick menu changes, if an operation was carried out

Even if the quick menu now only shows "Show Data (1)" it is still possible to load data into the data(1)-array. The corresponding option can be chosen from the menubar.

So, to conclude the advantages of the quick menu: The quick menu should be a signpost for all who have never used the tool, but it can be also helpful for those who want to do their work quite fast.

The tool supports import and export of data. The import can be done with files or from a database. Supported files are all which are delimited by a character or character sequence, e. g. '\t' (for tabulator), ';', etc. It is possible to set also field delimiter, if there are needed any. Moreover, it can be decided, whether a specific column is imported. This decision can be made by simply putting the number of column to import or by putting the column-name to import into the appropriate fields (figure 8). For now the *import-from-database* option works only with MySQL and MSAccess. It is marked "still experimental" due to the fact that a user has to set many options outside of the program, e. g. an ODBC-datasource has to be set, if an MSAccess-database has to be accessed. However, the import from databases option works, if all necessary requirements are given.

In the window of "Apply matching" the different approximate string

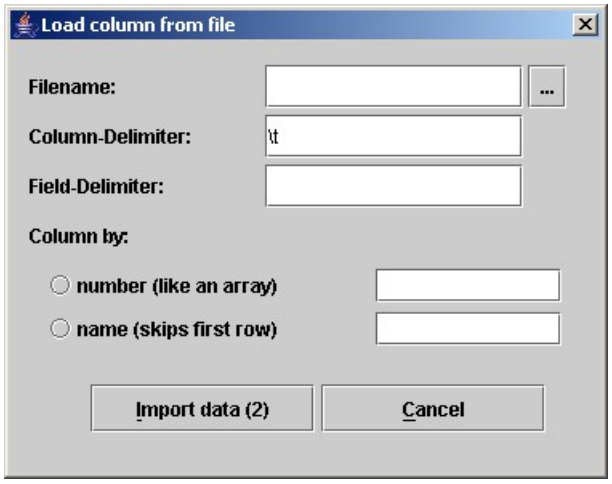

Figure 8: The *import-from-file* dialog and its options

matching techniques can be chosen. For now only q-gram and *edit distance* are in the combobox, but due to the implemented interface, it is possible to add any string matching algorithm.

If a string matching algorithm is chosen, the window changes slightly due to the fact that different string matching algorithms need a different amount of parameters. For example, using *edit distance* requires to set the weight of the operations which need 3 textfields (1 for insertion, 1 for deletion and 1 for substitution). However, *q-gram* only takes one textfield to set q. In this case both algorithms have also the field *boundary value*.

After setting all preferences for the string matching algorithm, the data can be matched. The dialog "Apply matching" remains open until the matching is finished.

After the matching is done, the "Apply matching" window closes and the quick menu changes again. Now it is possible to show all found matches. From the window, where all matches are listed, it is also possible to export the data.

At every time it is also possible to review the imported data. In fact the frame, which is used to show the matches and the frame, which is used to show the imported data, are the same. Therefore also both windows support the option of exporting the data.

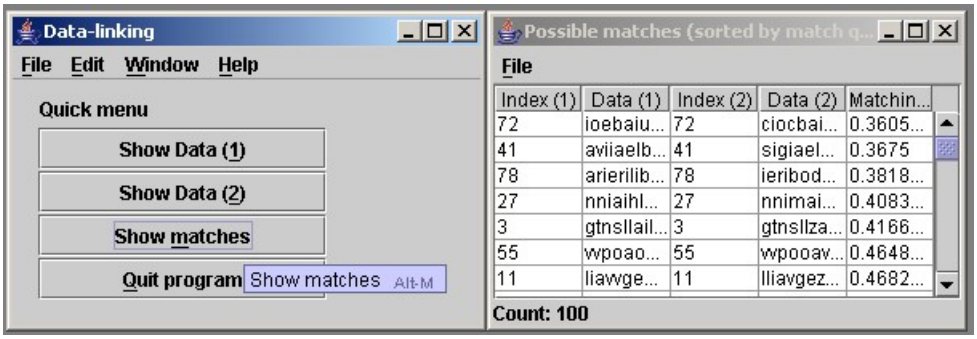

Figure 9: Left: The *Apply matching* button changed to *Show matches* — Right: The list of all found matches, beginning with the worst match

7 Evaluation

In this section I evaluate the different approximate string matching algorithms to find good matches for street names. For the tests I use the implementation described in section 4.2.

7.1 Accuracy

Both approximate string matching techniques, *q-gram* and *edit distance*, gave good results with real street names of residential addresses as well as with random generated string pairs.

When speaking of *"matches"* I mean *all* the matches, which were found. With *"valid matches"* I refer only to a part of *all* matches, namely the matches, where real objects point to each other correctly. So, *all found matches* without the *valid matches* would give a list of all incorrect matches.

In this section I show the results of three experiments, which all used the *all best and equal-best matches*-method: One experiment studies the impact of parameters. The parameters are q for the *q-gram algorithm* and the different weights for the *weighted Levenshtein distance*. The two other experiments compare the different approximate string matching algorithms, to identify, which algorithm works better for matching street names.

Experiment 1

In the first experiment I've used the *q-gram algorithm* and *weighted Levenshtein distance* with different parameters to match the real street names.

Starting with the *q-gram algorithm*, I matched the street names with different values for q.

Figure 10 shows the matching quota of the q-gram-algorithm, when used to match the street names with different values for q. I set the boundary

value to 60 %, which means that every match which has a better match than 60 % is collected and every match below 60 % is not collected. "Q-gram matching quota" shows the number of all found matches, whereas "actual matching quota" is the overall number of valid matches. So, the "actual matching quota" is the more interesting quota.

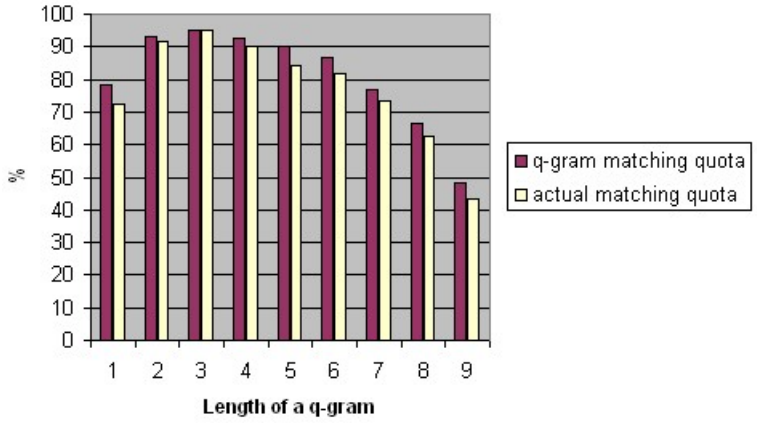

Figure 10: Matching quotas of the q-gram-algorithm with different values for q

If q is around 3 a lot of matches are found. However, these matches are not all valid matches. Only for $q = 3$ all found matches are also valid matches.

Summarizing, the best value for q for matching street names is 3. In addition all these found matches are valid matches for a lower boundary of 60 %.

The *edit distance algorithm* was tested with different parameters, i. e. the weight of operations (cost of insertion, cost of deletion, cost of substitution) was altered. I've written a small program, which tests many of the different possibilities of the parameters for matching the street names. However the *edit distance algorithm* never reached the same quality when matching, as the *q-gram-algorithm*. The best result was got, when cost of insertion was 1, cost of deletion 5 and cost of substitution 6. However, the matches found were not all valid.

Summarizing, when using *edit distance* for matching street names the weights should not be unit weight. However, I cannot suggest any values for the weighting as with different street names or test data the result may vary.

Concluding, the *q-gram algorithm* is the better choice for matching street names as the best value for q is 3. Additionally, the *q-gram algorithm* is faster and more accurate.

Experiment 2

In the second experiment I show how the matching of street names is done using the *q-gram algorithm*. In this experiment I do not set a lower boundary for the matches, which means that every match is collected, even if $q\text{-}gram(\sigma_1, \sigma_2) = 0$.

The results of the experiment are the following:

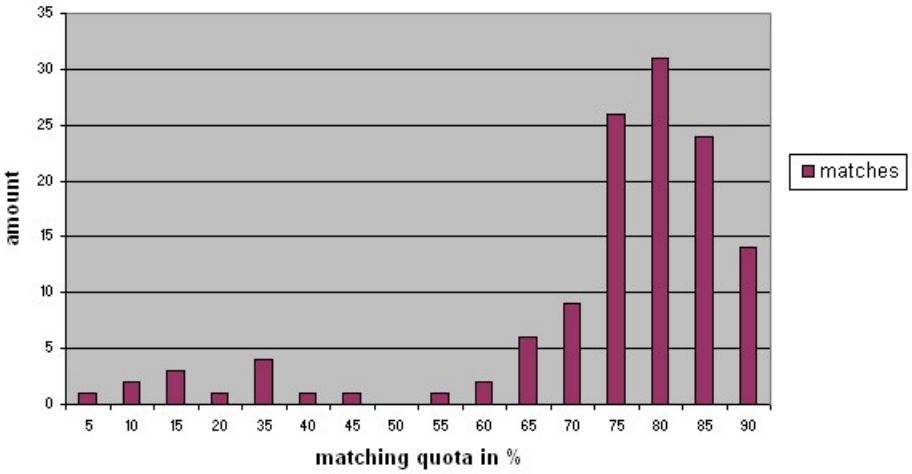

Figure 11: Matching street names with the *q-gram algorithm*

Completely equal matches are not listed in the figure, because the most matches are completely equal and I want to concentrate on the matches, which are achieved using approximate string matching.

From figure 11 we can see that at 50 % there is no match, but below this % there are again matches. In this case all matches below 50 % are all incorrect matches and above 50 % are valid matches. This allows a clean separation of valid and incorrect matches.

Concluding, matching street names with the *q-gram algorithm* is best used when setting a lower boundary with at least 50 %.

Experiment 3

In the third experiment I show how the matching of street names is done using the *weighted Levenshtein distance*. In this experiment I do not set a lower boundary for the matches, which means that every match is collected, even if the distance is very high. In addition I use the unit weighted edit distance.

When looking at the *edit distance algorithm* most matches were also valid, but between some real matches there were some false matches with

a quite low edit distance (note: small values for edit distance are better matches!). There is no threshold that allows us to classify valid and incorrect matches, as with the *q-gram algorithm*. Within *edit distance* = 4 all matches are valid.

In figure 12 the matching of street names using the *weighted Levenshtein distance* is shown. Again the completely equal matches are not listed.

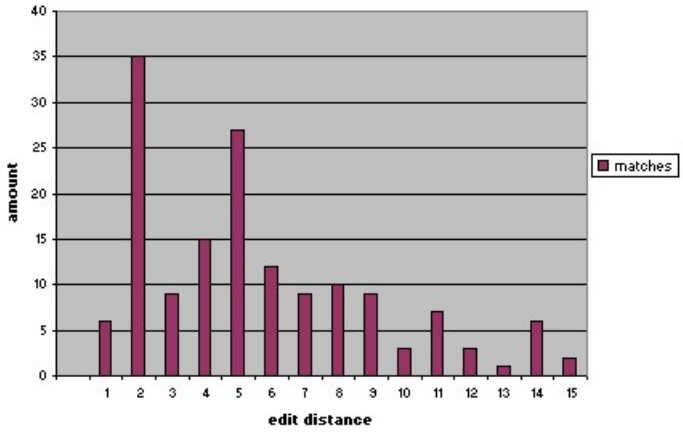

Figure 12: Matching street names with *weighted Levenshtein distance*

Discussion of Experiment 2 and 3

When matching random generated data with few differences both approximate string matching techniques lead to good results. The higher the differences of the random string pairs got, the more error-prone the techniques work. However, in most cases the q-gram algorithm gives better results than the edit-distance algorithm.

Comparing the *q-gram technique* with the *weighted Levenshtein distance*, one can see that with the *weighted Levenshtein distance* there are two hot spots, while with *q-gram* the good matches show a bell-shaped distribution.

Additionally, for edit distance I can not partition valid and incorrect matches by simply choosing a threshold on the distance. In fact invalid matches already appear with an edit distance of 5.

Concluding, when matching street names the *q-gram algorithm* leads to better results. Moreover, with the *q-gram algorithm* a threshold can be fixed to separate valid from incorrect matches. The experiments on virtual test sets confirm these results.

The *weighted Levenshtein distance* instead gets valid matches only below an edit distance of 4 if unit weighted. If the weight of operations changes, the boundary value has to be changed too. The *weighted Levenshtein distance*

was tested with different weighted costs for operations, but always got worse matches than the *q-gram technique*.

7.2 Performance

The O-notations of the approximate string matching algorithms were already introduced in section 3. Concluding, the implemented *q-gram-algorithm*, which is $O(n \log n)$ in time, is much faster than the *edit-distance*, which is $O(n^2)$ in time. However, both approximate string matching algorithms are adequately fast enough for matching the street names.

When choosing good matches performance becomes an issue. The algorithm is reasonably fast only, if about $R_1 = 1000 * R_2 = 1000$ strings are matched. Beyond this boundary the algorithm gets continuously slower. The reason for that is, that the method was implemented using a recursive call. If the matching is done by collecting only *the first best matches*, the runtime is faster. If larger arrays of data are matched, an `OutOfMemoryError` might occur for both methods. This error can be avoided if the program is started with additional parameters for the java virtual machine in which the heap-size is enlarged.

8 Conclusion and Future Work

In this thesis I describe two tools, which I've developed. One tool is used to match street names stored in databases or in text files. The other tool is used to create virtual data for experiments, namely random strings or random string pairs. These can be used as input data for the former mentioned tool. The distribution for the random strings was given by analyzing the street names of the databases given from the Municipality of Bolzano-Bozen.

The method with which the street names are matched is quite fast with small amount of data. "small" in this case means less than 1000 times 1000 strings. Beyond this boundary the method gets slower and slower and the heap size for the virtual machine must be increased, so that the matching can be done.

The main method which is used to match the street names should be replaced with a non-recursive function if larger data arrays have to be matched.

Other approximate string matching algorithms which implement the interface `StringMatching` could be written to have a greater variety of string matching algorithms. So there might be a string matching algorithm for every purpose.

References

[ABG04] Nikolaus Augsten, Michael Böhlen, and Johann Gamper. Reducing the integration of public administration databases to approximate tree matching. In *proceedings of the Third International Conference on Electronic Government*, 2004.

[All99] Lloyd Allison. Dynamic programming algorithm (dpa) for edit-distance. 1999.
`http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Dynamic/Edit/`
(from 09/07/04).

[GIJ+01a] Luis Gravano, Panagiotis G. Ipeirotis, H. V. Jagadish, Nick Koudas, S. Muthukrishnan, Lauri Pietarinen, and Divesh Srivastava. Using q-grams in a DBMS for approximate string processing. *IEEE Data Engineering Bulletin*, 24(4):28–34, 2001.

[GIJ+01b] Luis Gravano, Panagiotis G. Ipeirotis, H. V. Jagadish, Nick Koudas, S. Muthukrishnan, and Divesh Srivastava. Approximate string joins in a database (almost) for free. In *Proceedings of the 27th International Conference on Very Large Data Bases*, pages 491–500. Morgan Kaufmann Publishers Inc., 2001.

[Kur96] Stefan Kurtz. Approximate string matching under weighted edit distance. In *Proceedings of Third South American Workshop on String Processing August 1996 Carlton University Press*, Recife, Brazil, August 1996.

[ST95] E. Sutinen and J. Tarhio. On using q-gram locations in approximate string matching. *Proceedings of Third Annual European Symposium (ESA '95)*, pages 327–340, 1995.

[Ukk92] E. Ukkonen. Approximate string matching with q-grams and maximal matches. *Theoretical Computer Science (TCS)*, 92(1):191–211, 1992.

[Ull77] J. Ullman. A binary n-gram technique for automatic correction of substitution, deletion, insertion, and reversal errors in words. *The Computer Journal*, 20(2):141–147, 1977.