A Geocoding Algorithm Based On A Comparative Study Of Address Matching Techniques

by
Name: Bas Ranzijn
Student number: 371528
Master Thesis Operations Research and Quantitative Logistics
Erasmus Universiteit Rotterdam
October, 2013

Supervisors Erasmus Universiteit Rotterdam
Dr. Viorel Milea
Dr. Flavius Frasincar

Supervisors CQM
Stijn Duijzer MSc.
Dr. Johan M. M. van Rooij
Dr. Ir. Jacob Jan Paulus CPIM

Preface

This thesis is the conclusion of my Master Operations Research and Quantitative Logistics at the Erasmus University of Rotterdam. The subject of the thesis, the development of a geocoding algorithm, follows directly from practice. CQM, a consultant company providing quantitative solutions to diverse problems other companies encounter, asked me to make an improvement in their geocoding algorithm used in a route planner. More specific, the improvement should be made by a better and faster retrieval of the correct addresses in the database.

For six months, from the first of April 2013 until the end of September 2013, I worked with great pleasure on the thesis at the office of CQM in Eindhoven. I could not wish for a better place to write my thesis, despite the long daily trips between Rotterdam and Eindhoven. Many thanks go to my three supervisors of CQM, whom enthusiasm, experience, analytic skills and friendliness were an excellent help: Stijn Duijzer, Johan van Rooij and Jacob Jan Paulus.

I would also like to thank my supervisors from the university: Viorel Milea, my supervisor during the project, for the pleasant and useful meetings and the discussions on the many ideas during the project, and my co-reader Flavius Frasincar, for the last iteration of constructive feedback.

At last, thanks to my family and girlfriend, for the not to be underestimated mental support.

Abstract

A common problem in geocoding is that the description of a location as requested by a user differs from the description of the location in the database, due to misspellings, alternative spelling, abbreviations, different word orders, or addition or omission of words. This thesis is about the development of an algorithm to determine which location in the database is the targeted location and is motivated by the question of CQM to improve their current algorithm, applied in a route planner.

The thesis consists of two parts. In the first part, we compare several similarity measures on the quality of matching town or street names. Testing on four datasets showed that the trigrams Jaccard and trigrams Dice's similarity measures are best to use for both town and street names. In this part also a zip code matching algorithm is proposed to determine the best zip code in the database, given an erroneously input zip code.

In the second part, we propose a matching algorithm to find the targeted location in the database, given an input location. The algorithm uses the findings from the first part and focuses on strings in the database that match exactly with an input string. An important assumption is that the input strings from different types are presented in different fields and the country is given via a land code. The scope of the algorithm is Europe. Comparisons with the current matching algorithm of CQM on three datasets showed a significant improvement. The percentage of correctly returned locations is for all three sets above 97%, yielding improvements of 16.76%, 14.86% and 4.52%.

Contents

1 Introduction and Problem Description

1.1 Introduction

Geocoding is the process of assigning geographic coordinates to postal addresses and is applied in a wide variety of domains such as health care, crime analysis, political science and computer science [11]. One important and obvious application is geocoding in route planning. The geocoding algorithm in a route planning device, translates the addresses given as input by the user to their corresponding geographical coordinates, the longitude and latitude, such that the fastest or shortest route between two points can be calculated.

A problem that arises in geocoding is that the description of a location as requested by a user, may differ from the description of the location in the database. This may be due to misspellings, alternative spelling, abbreviations, different word order, and addition or omission of words. This thesis investigates the problem of matching identical addresses that differ due to these errors.

A nice example of inconsistencies that may occur in reality, selected from the datasets that are used in the thesis, is given in Table 1. Note that the first alternative differs from the original in the street name, due to an abbreviation and two typographical errors. The second alternative differs in all three string types. Instead of the name of the town, the community Terneuzen is given together with an additional substring. In the street name, the substring "Henry" is omitted and there is a typographical error in the zip code.

	Representation in database	**Alternative 1**	**Alternative 2**
Town	Hoek	Hoek	Terneuzen (Slippenspolder)
Street	Herbert Henry Dowweg	Herbert A. Dokweg	Herbert Dowweg
Zip code	4542	4542	4532

Table 1: Example of erroneous input addresses

1.2 Problem description

The subject of the thesis follows directly from a problem encountered in reality. CQM[1] is a company in Eindhoven, the Netherlands, with around thirty consultants having a mathematical background, providing quantitative solutions to problems other companies encounter. CQM has developed a route planner device, based on geographical data from OpenStreetMap[2], an open source

[1] www.CQM.nl
[2] www.OpenStreetMap.org

provider of geographical data. The geocoding part of the route planner device needs to be improved in dealing with the inconsistencies that could occur between an address given by a user and the address information of the targeted location in the database. The device is used by international transporters and is able to calculate routes between locations in Europe.

The input given by the planners of the transporters consists of the following parts:

- The country, which is a required field and should be entered correctly via a code. The search for the corresponding location can therefore focus on the chosen land. Europe is the scope of the route planning device.
- The other fields are all optional and entered **separately** of each other:
  - Town
  - Street
  - Zip code

We assume that we have a correct and complete database. Advice on how to deal with a database with errors is given at the end of the thesis, as well as an advice on how to incorporate a community input field.

When is written down *town*, *street* or *zip code*, we sometimes mean a town, street or zip code *string* and sometimes an existing town, street or zip code. We decided not to make a visual distinction between those two types, for the purpose of readability. The word *town* is used as generic term for a city, town, or village.

When referring to a *location*, this can be on all levels of detail. For example, a location can be only a town, but also a street or a zip code (regardless how big the region is to which the zip code refers).

1.3 Research questions

The thesis basically consists of two parts, in both of which a research question is answered. The main research questions investigated are:

- How to determine which string in a database is the targeted string, if an input string is a part of a postal address? (First part; Chapter 3 and Chapter 4)
- How to determine which location in a database is the targeted location, if the input location consists of strings of at least one and at most all of those types: town, street and zip code? (Second part; Chapter 5 and Chapter 6)

The first main research question can be split into three sub-research questions:

- How can an input string from the type town be matched in the best possible way with the targeted town string?
- How can an input string from the type street be matched in the best possible way with the targeted street string?
- How can an input string from the type zip code be matched in the best possible way with the targeted zip code strings

A sub-research question answered in the second part is:

- How can the developed algorithm to define the targeted location be made fast enough to be applicable in practice (i.e., <1 second to present suggested locations to the user)?

The matching algorithm proposed in the second part makes use of the findings in the first part. All implementations are done in C++, together with cross-platform application framework Qt[3].

There are three important assumptions, restricting the problem setting (and restricting our set of possible solution methods to answer the research questions):

- We cannot get the feedback if the targeted location is among one of the to the user presented locations, which makes it impossible to develop a learning matching algorithm.
- We do not have any knowledge about the language used in a country, except for the information we can extract from strings in the database.
- We do not have any geographical information about where in the country a location in the database is located (e.g., we do not know if two towns are far or close to each other).

1.4 Methodology

To answer the research questions how to match strings of the types town and street, we used four datasets to compare the performance of different similarity measures selected from literature. Similarity measures measure the degree of similarity between strings. The measures having the best performance are suggested to use for matching town and street strings.

For matching zip code strings, we propose an algorithm that exploits the property of the zip code of most countries, that the beginning of the zip code indicates in which region the zip code is

[3] `http://en.wikipedia.org/wiki/Qt_(software)`

located. It also makes use of a similarity measure to correct for errors.

In the second part, we use the best performing similarity measures to match town and street strings, and the zip code matching algorithm, in the development of an algorithm to determine the targeted location given an input location. The algorithm is based on the observation that a matching input string is likely the targeted input string. A preprocessing method is proposed to filter out clearly non-targeted locations, to reduce the computation time of the algorithm. The performance of the algorithm is investigated by a comparison with the current matching algorithm used by CQM. For both algorithms, the percentage of correctly returned targeted locations is determined, using three datasets with locations requested in reality. Which location in the database is targeted, is for every input address manually determined by the author of the thesis, as a gold standard (i.e., the truly targeted locations) is not available and it was not possible to go through the datasets by more than one person.

1.5 Outline thesis

In Chapter 2 we do a general literature study, after which we focus on the properties of similarity measures proposed in the literature, in Chapter 3. In Chapter 4, the best similarity measures to match strings of the types town and street are determined, by testing on datasets used in reality, consisting of locations in the Netherlands. Also in Chapter 4, an algorithm is proposed to match strings of the type zip code. In Chapter 5, the best similarity measures and the zip code algorithm are used in a "matching algorithm", developed to match input locations with the targeted locations in the database. A preprocessing method is given, which considerably speeds up the search. Also, a comparison is made with the current matching algorithm CQM uses. In Chapter 6 we give advice on practical considerations and small problem extensions. Finally, the conclusions are given and possible future work is discussed.

2 Literature Review

The literature research for this thesis consists of two parts. The first part, presented in this chapter, gives an overview of the relevant literature. The second part consists of an overview of all similarity measures and distance measures available in literature, which is given in Chapter 3.

Goldberg [11] gives a general overview of the concept of geocoding, including historic developments and future challenges. Goldberg defines geocoding as finding the geographically referenced code representing an input address, determined by a processing algorithm. The processing algorithm consists in general of the following steps:

- Normalization and standardization. In this step, the input address is transformed into a standard form, for example by removing punctuation and recognizing abbreviations. The words in the input string (or, substrings of the input string) are also tokenized (i.e., classified as belonging to a specific input type).
- Weighting attributes. The attributes obtained from tokenizing are weighted, to be able to measure the degree of similarity of the input address with addresses in the reference data set.
- Search the reference set for a match and output the geographical coordinates.

One of the most important difficulties in geocoding is how to deal with inaccurate data in the reference set and erroneous or inconsistent input data. Goldberg emphasizes the necessity of massaging the input data to be able to find accurate matches.

Churches [7] states that geocoding is a special case of record linkage. Record linkage is concerned with how to find entities referring to the same real-world entity, in the same or different databases [9]. In the general record linkage setting, the information about the entities is presented in different fields, corresponding to different properties of the entity. The input addresses we consider in the thesis, are also separated into different fields, which makes the theory about record linkage especially interesting.

Fellegi and Sunter developed the main theory of record linkage, in which pairs of entities are classified as matches, possible matches, or non-matches, given the values in the different fields of the entities [10]. Several authors used the main theory as framework to develop record linkage algorithms. An example is Winkler [26], who proposed a method to estimate the weights assigned to the different fields, based on the EM algorithm.

Recently, both Winkler [28] and Elmagarid et al. [9] have written a survey of record linkage. Elmagarid et al. state that one of the most important aspects of record linkage is the (typographical) variety in string data and give an overview of the most important character based, token based and phonetic similarity measures, to calculate the similarity between strings in two different fields. To detect which entities are duplicates, probabilistic matching models, machine learning techniques, distance based techniques and rule-based techniques are summarized. One of the distance based techniques is analyzing the record as one long string and using a similarity measure to determine the similar entities, see for example Monge and Elkan [20]. A disadvantage of this approach is that it ignores information that can be obtained when treating each field separately.

For the approach in which each field is analyzed separately, an overall similarity measure can be defined as a weighted distance of the distances (according to one or more similarity measures)

between the individual fields. An application of the weighted distance rank is written by Koudas [18], in a paper about how to deal with problems in record linkage encountered in practice. Koudas' conclusion was that the use of dynamic weights, in which information given in a field is taken into account (e.g. Main Street is less specific than the corresponding zip code), is highly favored over the use of static weights. Another approach to determine the most similar entity, is by using a merged ranking. For every field a rank is based on a similarity measure, after which an algorithm merges the ranks into an overall rank, see [12]. Jin [16] proposes a different merging rule: if the similarity values for all different fields are within specified thresholds, the records are considered matching.

Another interesting study area from which we might obtain useful information, is approximate string matching. The most extensive and well-known survey for approximate string matching is written by Navarro [21]. Navarro broadly explains the history of approximate string matching and application domains, among which the most important are computational biology, signal processing and text retrieval. He compares several techniques from the different types of existing algorithms: dynamic programming, automata, bit-paralellism and filtering. Although closely related to the area of the subject in the thesis, the theory is not well appropriate, because the focus in approximate string matching is on finding the identical string in a text, instead of finding identical strings in a database with records.

There are several papers written about the more to the thesis related subjects, address matching and/or name matching, in which the identical address or person needs to be found in the database. Davis Jr. [17] proposes a method applicable on both name matching and address matching. The assumption is made that both the input string and all strings in the database are of the same type (note that this is not a record linkage setting in which the different fields need to be weighted to determine a final similarity value). Their approach is capable of dealing with (non-)standard abbreviations, stop words, omission of substrings and substring swaps. After a preprocessing step to standardize the data and limit the search space, the similarity measure used to determine the final similarity between the input entity and the entity in the database depends on if the input string consist of one or multiple words.

Christen [4, 5], and Churches [7] propose algorithms to match addresses in a geocoding setting. In the papers it is assumed that the input address is given in one single field. To determine which word of the address belongs to which type (examples of a type are town and street), trained hidden Markov Models [23] are used in the standardization phase to classify. The paper of Churches gives a detailed description of how the hidden Markov Models work and can be trained. Except for Buckley [3], who takes a completely different approach based on fuzzy set theory and clustering of the database, an application of address matching in a multiple field record linkage setting is not

available.

In the literature described applications on name matching are more similar to the problem investigated in the thesis. For example, Yancey [29] applied name-matching in a record linkage setting (the fields are: surname and last name) and compared several similarity measures. The overall similarity score of two records is calculated as the sum of the similarities for each field. Christen [6] and Cohen [8] also compared different similarity measures in name-matching, but for single field records. There is no clear conclusion which measure is best. Jaro-Winkler, longest common substring, unigrams Overlap and bigrams Overlap can all be recommended to match personal names. An important note is that in these three papers and in the survey paper of Elmagarid [9], it is stated that which similarity measure is best highly depends on the type of data and that further research needs to be done.

To summarize, we did find papers on address matching, but only for single field input (in which the complete address is given). There are papers on string matching in a multiple field record linkage setting (i.e., the problem we encounter in the thesis), but they are applied only on personal names and not on postal addresses. It would be interesting to know what the best measure is to match address strings that are from the same type (e.g., how to match strings of the type town or of the type street) and how to define the overall similarity between addresses that are represented in a multiple field record linkage setting.

3 Overview similarity measures

This chapter describes distance measures and similarity measures available in literature, which determine the degree of similarity between two strings. In Chapter 4, for each type of input in the route planner (town, street and zip code strings) several suitable similarity measures are selected. These measures are implemented, after which testing determines the best measure(s) for string matching on the different input types. Measurements are rated based on a combination of matching quality given the errors, and calculation speed. The evaluation of the performance of a measure on a specific error type will be a trade-off between matching quality on the error type and an estimation of how frequently the error occurs.

The measures presented in this chapter are categorized in character-based, token-based and phonetic measures. The majority belongs to the category of character-based measurements. Phonetic measures are the least interesting, due to the multilingual problem setting we have in the thesis, as the scope of the project is Europe. The important properties of the measurements in each category

are discussed. A summary is given at the end of the chapter, in Table 2.

The terms distance measure and similarity measure are used without distinction in this thesis. However, a similarity measure can be seen as the inverse of a distance measure. The smaller the distance between strings, measured by a distance measure, the better the agreement is. A similarity measure is a number between zero and one, with one indicating perfect agreement of the strings and zero no agreement. It is often trivial to transform a distance measure into the corresponding similarity measure, because distance measures are typically bounded due to finite lengths of the strings.

The following notation is used in the descriptions of the measures:

- $s1$ is string 1
- $s2$ is string 2
- n is the length of the shorter string
- m is the length of the longer string
- i is the letter at the i-th position in string 1
- j is the letter at the j-th position in string 2

3.1 Character based

In this section we propose the character based similarity measures. We categorized the similarity measures in three different groups, each discussed in its own subsection. The first group consists of measures that are related to the edit distance, the second group consists of measures of the Jaro-Winkler type and in the last group q-gram measures are given.

3.1.1 Edit distance

Edit distance or Levenshtein distance
The most well-known and widely used distance metric. It is defined as the smallest number of edit operations required to turn $s1$ into $s2$. Edit operations are insertions, deletions and substitutions of characters. In its basic form, every edit operation used is assigned cost 1. However, the costs can be chosen arbitrarily. The Levenshtein distance equals $lev_{s1,s2}(|s1|, |s2|)$, with

$$lev_{s1,s2}(i,j) = \begin{cases} max(i,j) & \text{if } min(i,j) = 0, \\ min \begin{cases} lev_{s1,s2}(i-1,j)+1 \\ lev_{s1,s2}(i,j-1)+1 \\ lev_{s1,s2}(i-1,j-1)+[s1_i \neq s2_j] \end{cases} & \text{otherwise.} \end{cases}$$

and is solved by dynamic programming. The first element in the minimum corresponds to a deletion, the second to a insertion and the last to a substitution. $[s1_i \neq s2_j]$ in the last element is an indicator function equal to one if the character at position i in $s1$ is unequal to the character at position j in $s2$, and zero otherwise.

The maximum edit distance is $max(|s1|, |s2|)$ and the minimum edit distance is $abs(|s1| - |s2|)$. This minimum bound on the edit distance is commonly used as a filter, if the edit distance may not be larger than k.

A useful property is that the Levenshtein distance is a true metric, because it satisfies the triangle inequality. Also the triangle inequality can be used to filter when determining the best match(es) out of a dictionary of strings, by selecting strings as "pivots". A pivot is a string for which the edit distance between other strings and the pivot is determined beforehand, such that a lower bound on the edit distance between input query and the other strings can be quickly calculated. For a detailed explanation of the triangle inequality and how to use it in filtering when applying the Levenshtein metric, see [14].

To turn the Levenshtein distance metric in a similarity measure the distance should be divided by the upper bound on Levenshtein distance, which is the length of the longer string [6]:

$$simLev(s1, s2) = \frac{lev_{s1,s2}}{m}.$$

The time and space complexities are $O(|s1| * |s2|)$. If the edit distance between the strings is only of interest if its smaller than an integer k, the time complexity can be reduced to $O(min(|s1|, |s2|) * k)$ [13]. The Levenshtein distance soon gets too slow to use in practice, if the strings compared are larger. Several techniques exist to reduce the calculation time when one input string needs to be compared with all strings in a dictionary [1].

Damerau-Levenshtein distance

The Damerau-Levenshtein distance metric, is an extension of the Levenshtein distance metric, as it also allows transpositions (i.e., swaps) of adjacent characters at cost 1 [1, 6]. Just as the basic

edit distance, Damerau-Levenshtein satisfies the triangle inequality and is calculated in $O(|s1|*|s2|)$.

Hamming distance
The Hamming distance measures in linear time with respect to the size of the strings the number of different characters at the same positions in two equally sized strings. This can be seen as the number of substitutions needed to turn one string into the other string [21].

Bag distance
A bag is defined as the multiset of the characters in a string. Let x be the bag of $s1$ and y the bag of $s2$. Then the bag distance is defined as $BagDist(s1, s2) = max(|x - y|, |y - x|)$, where $|a - b|$ is the number of characters in a that is not in b, taking into account letters occuring more than once.

The bag distance returns a lower bound on the Levenshtein distance in $O(|s1| + |s2|)$ time and is therefore often used as a cheap approximation for the edit distance, or as filtering step before application of a different more sophisticated similarity measure [6].

Coherence edit distance
The coherence edit distance is another extension of the basic edit distance and takes more context into account by checking if similar characters appear close to each other [29]. The coherence edit distance is calculated in the same way as the basic edit distance measure, except for the substitution part, which is replaced by:

$$\min_{1 \leq a \leq N, 1 \leq b \leq N} \{lev_{s1,s2}(i - a, j - b) + V_{a,b}(i, j)\},$$

where N is the maximum number of characters back in the strings to compare with (often set to 4) and V a so-called edit cost potential function defined as:

$$V_{a,b}(i, j) = \begin{cases} 3/4 * (a + b - 2/3) - c & \text{if t is even} \\ 3/4 * (a + b - 1) - c & \text{if t is odd} \end{cases},$$

with c the number of common characters found when comparing the characters up to respectively a and b positions back in the strings. Note that the edit cost potential function increases when comparing characters positioned further back in the strings and decreases in the number of common characters found.

The idea is that if the strings have more characters in common within a close range, the strings are more equal. The basic edit distance will in that case possibly underestimate the true similarity, as it only allows the basic edit operations.

Because only edit operations are allowed to transform $s1$ into $s2$, the edit distance metrics proposed above are especially suited for correcting typographical errors, but not for other types of discrepancies between strings, such as omission of words, abbreviations or different word orders.

Longest common subsequence
The longest common subsequence of two strings is the maximum number of common, not-necessarily adjacent, characters that are in the same order in both strings [29]. A similarity measure can be calculated in three ways: by dividing the longest common subsequence by the length of the shorter string, the length of the longer string or the average string length.

Longest common substring
The longest common substring is defined in a similar way as the longest common subsequence, but the measure calculates the number of common characters that are adjacent and in the same order in both strings.

Another method [6] to measure the similarity between two strings, is to repeatedly find and remove the longest common substring if the length of it exceeds a threshold value, which is most often set to 2 or 3. The similarity measure is then found by adding the lengths of the detected substrings and dividing the outcome by the length of the shorter string, the length of the longer string, or the average string length. This method can deal with swapped words. It is possible to calculate the longest common substring similarity in this way in $O(|s1| * |s2|)$ time.

Combination edit distance
The combination edit distance is a simple combination of the edit distance metric and longest common subsequence measure. It is mentioned here, because in a string similarity measure comparison study on matching names, it increased matching quality opposed to the edit distance and longest common subsequence measures [29]. The combination edit distance is calculated as follows:

$$sim(s1, s2) = a * (1 - \frac{e}{n}) + (1 - a) * \frac{l}{m},$$

where e is the edit distance between $s1$ and $s2$, l the longest common subsequence, and a a parameter between zero and one to set the weights for the edit distance metric and longest common subsequence measure.

Affine gap distance
The affine gap distance is developed for matching strings when one is truncated or shortened. It is

an extension of the basic edit distance, with two extra edit operations: open gap and extend gap. The cost for extending a gap should be set lower than the cost for opening a gap. The measure is usually applied in comparing DNA sequences [9, 25].

Smith-Waterman distance

Smith-Waterman is an extension of the affine gap distance. The difference with the affine gap distance is that characters match in several degrees: exact, approximately (e.g., based on Soundex, see below) and no-match. How the Smith-Waterman measure works exactly differs in the literature. See for examples and more details: [6, 9, 20].

One way to calculate the similarity value is as follows:

$$Smith\text{-}Waterman(s1, s2) = \frac{BS_{sw}}{DIV_{sw} * matchScore},$$

where BS_{sw} is the highest value in the dynamic programming matrix resulting from the edit distance calculation, DIV_{sw} is the length of the shorter string, length of the longer string, or the average string length and $matchScore$ the value assigned when two characters match exactly. The time complexity is $O(|s1| * |s2|)$.

3.1.2 Jaro-Winkler

Jaro

The Jaro distance (and the extension Jaro-Winkler, see below) is mostly applied on matching personal names [6, 9, 29]. In general, it has a good performance on matching relatively short strings. It is not a true distance metric, because it does not obey the triangle inequality. In this measure, strings are more similar if they have matching characters within a specific range, depending on the length of the longer string. A penalty is given if matching characters are transposed (i.e., if they are not in the same order).

The Jaro measure can be calculated in $O(|s1| + |s2|)$ time as follows:

$$Jaro(s1, s2) = \begin{cases} 0 & \text{if } c = 0 \\ 1/3 * (\frac{c}{|s1|} + \frac{c}{|s2|} + \frac{c-t}{c}) & \text{otherwise,} \end{cases}$$

where c is the number of matching characters and t the number of matching characters that is transposed. Two characters are considered matching if they are equal and positioned within the range $\lfloor \frac{max(|s1|,|s2|)}{2} \rfloor - 1$ of each other.

Note that the Jaro distance is actually an inverse distance measure (or, similarity measure), as its value increases with string similarity.

Jaro-Winkler

Jaro-Winkler distance is an often used extension of the Jaro distance [6, 9, 29]. Jaro-Winkler is equal to the value of the Jaro distance, plus a score if the prefixes of the strings are equal. This is because the developers of the measure assumed that errors tend to occur less often at the start of the strings [27]. The distance measure is calculated as follows:

$$Jaro\text{-}Winkler(s1, s2) = Jaro(s1, s2) + (l * \rho * (1 - Jaro(s1, s2)),$$

where l is the number of equal characters at the beginnings of the strings with a maximum often set to four characters. ρ is a parameter usually 0.1 to assign a weight to the common prefix. Also Jaro-Winkler can be calculated in $O(|s1| + |s2|)$ time.

Jaro-Winkler with longer string adjustment

This extension of Jaro-Winkler is developed in [29] and is applicable to longer strings (i.e, in this case at least five characters) that do meet some criteria, see below. The idea is that longer strings need to have a higher similarity score if characters match, also when they are not in the prefix. The intuition is that the contribution of a common prefix for the similarity value decreases when the sizes of the compared strings get larger.

The criteria the strings should meet are:

- $n \geq 5$
- $c - l \geq 2$
- $c - l \geq \frac{n-l}{2}$,

where l the length of the common prefix and c the number of matching characters. The first criterion assures the strings are long enough and the second that enough characters match besides the characters in the common prefix. The last criterion checks if the number of matching characters that is not in the common prefix is large enough, relative to the size of the shorter string.

The similarity score has a similar structure as the regular Jaro-Winkler distance measure:

$$Jaro\text{-}WinklerLSA(s1, s2) = Jaro(s1, s2) + (1 - Jaro(s1, s2)) * \frac{c - (l + 1)}{n + m - 2 * (l - 1)}.$$

Sorted Jaro-Winkler
The sorted Jaro-Winkler is the same similarity measure as the original Jaro-Winkler, but first $s1$ and $s2$ are sorted in alphabetical order if they consist of more than one word [6]. This measure will solve the problem of an input string with swapped words.

Permuted Jaro-Winkler
Similar to the sorted Jaro-Winkler, the permuted Jaro-Winkler measures the basic Jaro-Winkler distance for all possible permutations of the words in the strings [6]. Returned is the highest score. This measures is also developed to solve the problem if words in a string are not in the right order.

3.1.3 Q-grams

Q-grams: Dice's coefficient, Jaccard similarity and Overlap coefficient
Q-grams, also called n-grams, are substrings of length q [6, 9, 22]. A similarity measure is developed by creating the multisets of q-grams from $s1$ and $s2$ (multisets, because we need to take into account q-grams occurring more than once). The number of q-grams both strings have in common is then divided by the average number of q-grams in both strings (Dice's coefficient), the number of q-grams in the longer string (Jaccard similarity), or the number of q-grams in the shorter string (Overlap coefficient). The time and space complexity of the q-gram similarity measure is $O(|s1| + |s2|)$.

In formulas, the three given similarity measures are as follows. With S and T the multisets of q-grams of $s1$ and $s2$ respectively:

$$\text{Overlap coefficient} = \frac{|S \cap T|}{\min\{|S|, |T|\}},$$

$$\text{Jaccard similarity} = \frac{|S \cap T|}{\max\{|S|, |T|\}},$$

$$\text{Dice's coefficient} = \frac{|S \cap T|}{\frac{|S|+|T|}{2}}.$$

If a trigram is in both strings, but in string $s1$ it occurs more than in string $s2$, the number of times it occurs in string $s2$ is counted (of course, the same holds the other way around).

Another application of q-grams, which is very often used in practice, is to filter candidate strings by not exploring strings that have less than a threshold value T q-grams in common with the input

string [19]. After the filtering step, a similarity measure (note that this may also be a measure which uses q-grams) is applied to determine the best matching string.

Padded Q-grams
Padded q-grams are substrings of length q after attaching $(q-1)$ spaces at the start and end of the string. This has empirically proven to give better results for matching and indexing than "normal" q-grams [6]. The idea of padding is to make it more robust against errors. For example, in a string of three characters a small error (e.g., insertion or deletion of a character) may lead to zero shared trigrams. If making use of padded q-grams, there would still be some trigrams shared, which will lead to a useful distinction between the targeted string and strings that do not have any padded trigrams in common with the input string.

Positional Q-grams
Positional q-grams [6] include the positional information of the q-grams in the strings (e.g., the positional trigrams of "Rotterdam" are ("rot", 1), ("ott", 2), etc.). Similar q-grams must be within a maximum distance of each other to count as match. Of course, also to padded q-grams positional information can be attached. A disadvantage of this measure is that the measure may not give the desired results when having a substring addition or omission or, especially, when words are swapped.

Skip-grams
The skip-grams set of a string is the set of 2-grams (known as bigrams) plus all combinations of two characters while skipping up to a chosen amount of character(s) in between [6]. To illustrate, the string "Amsterdam" with up to one skipped character has the following set of skip-grams: ("am", 0), ("ms", 0), ("st", 0), ..., ("am", 0), ("as", 1), ("mt", 1), ("se", 1), ... and ("dm", 1).

Skip-grams are developed to deal with simple typographical errors such as an insertion, a transposition, a deletion or a substitution. As already explained when discussing the padded q-grams, the basic q-grams measure will not perform well if the strings are very short and a typographical error occurs.

3.2 Token based

In this section, we propose two different measures based on tokenizing the strings.

3.2.1 Term frequency - inverse document frequency (Tf-idf)

Tf-idf

Tf-idf (term frequency-inverse document frequency) uses on one side that more frequently occurring parts of the string are important and on the other side that less frequently occurring parts give more specific information about the string [9, 24]. The measure is mainly used to match a longer string query with patterns in a text. The use of the tf-idf to determine the similarity values between one input string and many strings in a list is not a regular application, but is still interesting to investigate.

The tf-idf makes use of cosine similarity, which is the cosine of the angles between the vectors obtained by projecting the strings in a high dimensional space. The closer the vectors are, the higher the similarity.

For each token of a string, for example a word or a q-gram, the frequency of the token in the string is the term frequency (note that this is often one). The inverse document frequency, calculated as the number of times the token appears in all strings in the database is used as a factor to give lower weight to more common tokens.

The similarity between two strings is calculated as:

$$sim(s1, s2) = \frac{\sum_{j=1}^{|T|} v_{s1}(j) * v_{s2}(j)}{||v_{s1}||_2 * ||v_{s2}||_2},$$

where $|T|$ is the number of tokens in the input string $s1$ and

$$v_s(w) = log(tf_w + 1) * log(idf_w),$$

where tf_w is the frequency of the token in the string and $idf_w = \frac{|N|}{N_w}$, with N_w the number of strings in the database containing token w and $|N|$ the number of records in the database. $||v_s||_2$ is the euclidean norm[4] of the vector of all tokens in the string.

We will not explain each part of the formula in full detail, but instead note that the intuition is that the similarity score increases when the strings have tokens in common. If a token that the strings have in common does not occur much in the database, it will have a higher contribution to the similarity score.

A different type of cosine similarity measure, with S and T the multisets of q-grams of $s1$ and $s2$ respectively, is:

[4] http://en.wikipedia.org/wiki/Euclidean_distance

$$sim(s1, s2) = \frac{|S \cap T|}{\sqrt{|S| * |T|}}.$$

This last measure [19] could also be placed in the q-grams subsection, as it divides the number of common q-grams by the square root of the multiplication of the number of q-grams in both strings.

Soft Tf-idf

In case words are chosen as tokens for the tf-idf similarity measure, it is possible no corresponding words can be found in the database if there is an error in the input string [2]. In the soft tf-idf, a similarity measure different than the tf-idf measure is first used as a filter. If the similarity value between a string in the database and the input string is large enough, the string is compared with the input string. This final comparison is done by using the similarity values resulting from both the tf-idf measure and the other similarity measure.

3.2.2 Monge and Elkan's recursive matching scheme

Monge and Elkan's recursive matching scheme

Monge and Elkan proposed this measure to calculate the similarity of two strings consisting of several words [15, 20]. The degree of similarity of each word in the input string with all the words of the string in the database is calculated using a similarity measure. The maximum of all the measures is taken for every word of the input string, after which the final degree of similarity is obtained by taking the average over all maximums. In formulas:

$$MongeElkan(s1, s2) = \frac{1}{|A|} \sum_{i=1}^{|A|} max \left\{ sim(a_i, b_j) \right\}_{j=1}^{|B|},$$

where $|A|$ is the number of words in $s1$, $|B|$ the number of words in $s2$, a_i word i in $s1$, b_j word j in $s2$ and sim is a similarity measure (Levenshtein, Jaro-Winkler, etc.).

The measure does not take into account the difference in number of words of both strings, which could produce unexpected results.

3.3 Phonetic

Soundex

Soundex is a phonetic similarity measure, initially designed for English language, making use of the sound of letters when pronounced. It is the most common phonetic coding scheme. Letters

that are sounding similar are assigned to the same group (the group is indicated with a number). The input string and the strings in the database are transformed into a code, consisting of the first letter of the string and three numbers corresponding to the groups to which the other letters in the string belong. See details in [6, 9] for how to define which three numbers are appended to the letter.

The Soundex measure is mostly used in a filtering step. If the code corresponding to a string in the database has more characters in common with the code of the input string than a specified threshold value, the string is kept as candidate. Soundex can be applied usefully on names of different origins, but not on Asian names.

Editex
Editex is a combination of soundex and edit distance based methods. The edit distance costs are 0 if the letters are equal, 1 if the letters belong to the same group and 2 otherwise [6].

As phonetic distance metrics are not applicable to the problem investigated in the thesis due to the multilingual setting, see for more details [6, 9].

3.4 Complexities and expected matching quality on different error types

In Table 2, the complexity of each similarity measure is given, as well as comments about the expectation of the matching quality. In the last column, we provide an example for each measure we tested later in the thesis, illustrating on which error type the measure does not provide good results.

Similarity measure	**Expected matching quality**	**Time complexity**	**Weakness**
Edit distance or Levenshtein distance	Mainly suited for correcting typographical errors.	$O(\|s1\| * \|s2\|)$	Error-type: substring addition Goal: Find targeted town out of all towns Input: Tilburg (Industrieterrein Vossenberg) Targeted: Tilburg Returned: Driebergen-Rijsenburg
Damerau-Levenshtein	Mainly suited for correcting typographical errors.	$O(\|s1\| * \|s2\|)$	Same as edit distance
Hamming distance	Only applicable if $s1$ and $s2$ have the same size.	$O(\|s1\|)$	
Bag distance	It returns a lower bound on the edit distance and is often used as filtering step, not as measure itself.	$O(\|s1\| + \|s2\|)$	
Coherence edit distance	Mainly suited for correcting typographical errors. More flexible than the basic edit distance.	$O(\|s1\| * \|s2\|)$	
Longest common subsequence	Can deal with several errors, depending on how the similarity measure is calculated. Not applicable on input strings with swapped words.	$O(\|s1\| * \|s2\|)$	Error-type: substring addition Goal: Find targeted town out of all towns Input: Rijswijk ZH (Ypenburg) Targeted: Rijswijk Returned: Slijkenburg
Longest common substring	Can deal with several errors, depending on how the similarity measure is calculated.	$O(\|s1\| * \|s2\|)$	Error-type: abbreviation Goal: Find targeted street out of all streets Input: D. Sonoyweg Targeted: Diederik Sonoyweg Returned: Sontweg

Similarity measure	**Expected matching quality**	**Time complexity**	**Weakness**
Combination edit distance	Is not applicable on a string with swapped words. Matching quality on other error types depends on parameter setting and how the longest common subsequence similarity value is calculated.	$O(\|s1\| * \|s2\|)$	Error-type: substring addition Goal: Find targeted town out of all towns Input: Rotterdam (Rubroek) Targeted: Rotterdam Returned: Hattemerbroek
Affine gap distance	Mainly suited for correcting abbreviations and typographical errors.	$O(a^2 * \|s1\| * \|s2\|)$ (a is maximum length of gap)	
Smith-Waterman distance	Extension of affine gap distance. In comparison with the affine gap distance, it can better correct for errors made due to substituted letters that have the same sound when pronounced.	$O(a^2 * \|s1\| * \|s2\|)$	
Jaro	Developed to correct for short personal names with errors. Not applicable if similar characters are not around the same position (may occur if there is an addition or omission and when words are swapped).	$O(\|s1\| + \|s2\|)$	Error-type: abbreviation Goal: Find targeted street out of all streets Input: Prof. H. Van De Hoevenstraat Targeted: Professor H. van der Hoevenstraat Returned: Van der Hoevenstraat

Similarity measure	**Expected matching quality**	**Time complexity**	**Weakness**
Jaro-Winkler	Same as Jaro.	$O(\|s1\|+\|s2\|)$	Error-type: abbreviation Goal: Find targeted street out of all streets Input: MGR. Geurtsstraat Targeted: Monseigneur Geurtsstraat Returned: Gerritsstraat
Jaro-Winkler with longer string adjustment	Same as Jaro.	$O(\|s1\|+\|s2\|)$	Same as Jaro-Winkler
Sorted Jaro-Winkler	Same as Jaro, but it is developed to correct for strings with swapped words.	$O(\|s1\|+\|s2\|)$	Error-type: character-edit Goal: Find targeted town out of all towns Input: Nieuw Lekkerland Targeted: Nieuw-Lekkerland Returned: Lekkerkerk
Permuted Jaro-Winkler	Same as Sorted Jaro-Winkler, but it may be better in correcting if an addition or omission occurs.	$O(w1 * w2 * \|s1\| + \|s2\|)$ (wi the number of words in string i)	
Q-grams	No obvious weakness, except for typographical errors in relative short strings. Quality for correcting a specific error depends on how similarity is calculated.	$O(\|s1\|+\|s2\|)$	
Padded q-grams	Same as "normal" q-grams, but assigns more value to beginning and end of string (hence, it is more vulnerable to mistakes at the beginning or end, but more robust against errors in the middle).	$O(\|s1\|+\|s2\|)$	All padded q-grams measures are sometimes weak on abbreviations Goal: Find targeted street out of all streets Input: A. Plesmanweg Targeted: Albert Plesmanweg Returned: Plesmanweg

Similarity measure	**Expected matching quality**	**Time complexity**	**Weakness**
Positional q-grams	Same as normal q-grams. The measure is more robust against coincidental common q-grams (i.e., common but on different positions), but it may go wrong when a substring is added or omitted and when words are swapped.	$O(\|s1\| + \|s2\|)$	
Skip grams	Same as normal q-grams, but better suited against typographical errors.	$O(\|s1\| + \|s2\|)$	
Tf-idf	Matching quality on different error types not clear beforehand. Will depend on which tokens are deleted and/or added to the set of tokens when an error is made.	$O(\|t1\| + \|t2\|)$ (ti is number of tokens in string i)	Error-type: character-edit Goal: Find targeted street out of all streets Input: Lengelsweg Targeted: Lengelseweg Returned: Mergelsweg
Soft Tf-idf	Same as tf-idf, also depending on the similarity measure chosen before the tf-idf is applied.	at least $O(\|t1\| + \|t2\|)$	
Monge and Elkan's Recursive Matching Scheme	Matching quality depends on similarity measure chosen. It is likely the measure is better in correcting for substring omissions than correcting for substring additions, because for every word in the input string the highest similarity value is used.	$O(\|w1\| * \|w2\| * O(sim))$ (wi is number of words in string i)	
Soundex	Not applicable in the thesis	-	
Editex	Not applicable in the thesis	-	

Table 2: Summary of similarity measures

4 Testing similarity measures

In this chapter, we propose which distance measure(s) are best to match town and street strings. For zip codes we propose an algorithm to determine the most similar zip code, because a similarity measure does not seem to be appropriate for matching this type of strings. For the reason behind the choice not to test similarity measures for zip codes and a description of the matching algorithm, see Subsection 4.2.3.

The outline of the chapter is as follows. First, it is explained how the tests are performed to determine the best town and street strings. Second, an overview is given which measures are selected to test for both string types and why they are chosen and others are not. Thereafter, the zip code matching algorithm is described. Then, the characteristics of the datasets used for testing are described, and finally the test results and the conclusions which measures are best are presented.

4.1 Test design

The tests are performed with the use of four datasets, with locations typed in reality. See Section 4.3 for a detailed description of the sets.

The first thing to notice is that town strings and street strings can be analyzed separately. Hence, we split every location present in the database in strings of the different types and put per database all strings from the same type in a list. Town and street strings matching exactly with a string in the database, are not relevant for a comparison of the measures and are neglected. Next to this, we considered only the unique town and street names in each dataset, to prevent we get unbalanced results. For all strings left over, every selected similarity measure outputs the three locations from the database it defines as most similar. See Figure 1, in which we depicted this procedure.

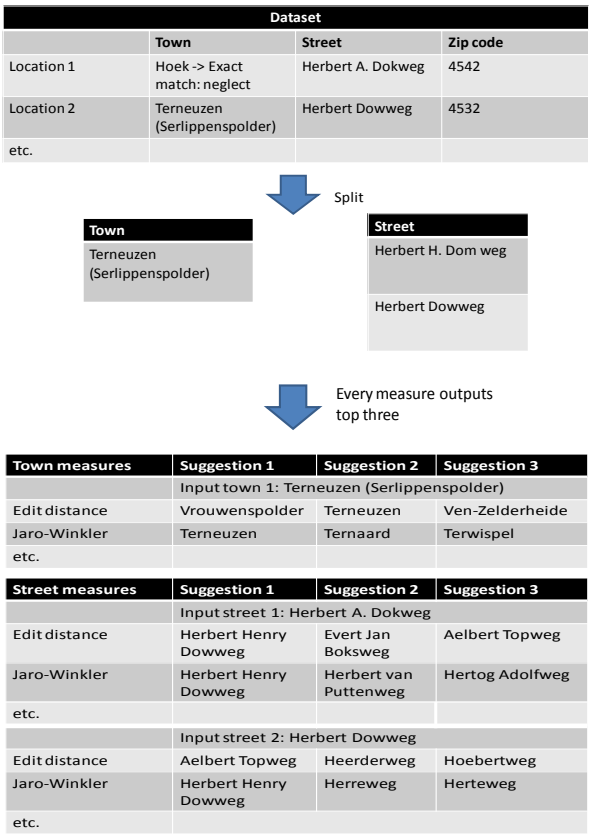

Figure 1: Test design

To be able to compare the performance of the measures, we first made a classification of different error types by analyzing the datasets. Error types are for example "addition of a substring", "abbreviation" and "character-error" (a character-error can be seen as a typographical error). Per error type, it is for every measure counted how often the targeted string is rated as "best" (i.e., most similar), second best and third best. We refer to the "best" returned string as the string given as first suggestion, the second best as second suggestion, etc. As said before, we do not have a gold standard, which indicates that we do not know which string is the targeted string. Besides, we did not have the possibility to go through the datasets by more than one person. Therefore, the author of the thesis defined manually which string is the targeted string. The results on the different datasets are merged. The conclusion which measure is the best can be drawn from the

performance of the measures on the different error types and how often errors of the types occur. The different error types for both string types are defined after analysis of the datasets, see Section 4.3.

We did not assign a score for each rank in the list of suggestions, but interpreted the results in a more qualitative way. Intuitively, the decrease in matching quality between second and third suggestion is smaller than the decrease between first and second suggestion and smaller than between third and "no suggestion". This is because we absolutely prefer to get the targeted string as first suggestion, but if it is not given as first suggestion, we prefer to at least get the string as suggestion. It does not matter much if it is in second or third position.

Also a qualitative analysis of the strings given as output by the measures is of high importance, as some "mistakes" are worse than others. If a measure returns a string that is obviously wrong, while there is a simple typographical error made in quite a long string, the penalty should be larger than the case where a measure is incapable of finding the targeted string if multiple words are abbreviated. Firmly stated, if the user thinks the erroneous string is obviously equal to the targeted string, but the measure returns a totally different string, the measure is not applicable in practice.

Strings that could not be mapped to a place or a street are not considered in the analysis. Also strings that correspond with alternative spellings of places are not considered (e.g., it cannot be expected that a measure can match "Den Haag" with its synonym "'s-Gravenhage").

To summarize, the results of the tests are analyzed by comparing the different similarity measures on three subjects. The first is a comparison on how often the measures returned the targeted string as first, second and third suggestion for each error type. The second is the capability of finding "obvious" matches. If we think the targeted string should be easily recognizable from the input string, but a measures fails to do so, this will greatly reduce the practical applicability. At last, the measures are compared on computation speed. All analysis are rather qualitative than quantitative, because of the difficulty to assign scores to the different performance measures. The in-depth discussion of the analysis can be found in Section 4.4. In that section, we first present all results for both town and street matching, after which we draw the conclusions. The main reason to draw no intermediate conclusions for matching towns, is that the results on matching street names might also provide valuable information about the performance of the measures on matching towns.

A very important note to make, is that the interpretation of the results should be done with caution. In Section 4.3, it is shown that some datasets are biased towards specific error types (on which certain measures do perform well and others do not), while these error types do not occur

much in other datasets. When interpreting the results, these dataset-specific properties will be taken into account. By analyzing the performance of the measures on different error types, it becomes possible to select suitable similarity measures to match town and street strings in a dataset in which a certain error type often occurs. Our goal however, is to determine which measures are best to match town and street strings in general.

4.2 Selected measures for testing

4.2.1 Town

For input strings of the type town, we have chosen the similarity measures in Table 3 to compare:

Edit distance	Jaro-Winkler with longer string adjustment
Damerau-Levenshtein distance	Bigrams overlap coefficient
Combination edit distance	Bigrams Jaccard similarity
Longest common substring	Bigrams Dice's coefficient
Longest common subsequence	Trigrams overlap coefficient
Jaro	Trigrams Jaccard Similarity
Jaro-Winkler	Trigrams Dice's coefficient
Sorted Jaro-Winkler	Tf-idf

Table 3: Similarity measures for town

From all measures of the edit distance type, we did not implement the coherence edit distance. In [29], in which a comparable study on personal names is performed, the coherence edit distance always performed worse than the standard edit distance. In the application domain considered in this thesis, we do not expect a large difference. Also the affine gap and Smith-Waterman distances we do not consider, because not much abbreviations in city names are expected. Besides, if abbreviations occur, other measures such as the longest common subsequence, the q-grams measures and the tf-idf should be able to deal with it quite well. Furthermore, the measures have a considerable larger computational complexity than other measures and for Smith-Waterman, Soundex is needed to determine similar characters. As determined earlier, Soundex is not appropriate, as we propose an algorithm independent of language.

From all Jaro-Winkler types we did not implement the permuted Jaro-Winkler distance, because it takes a lot more time to determine the degree of similarity between two multiple word strings. We think it is not likely that the better matching quality will outweigh these costs in computation time. A same argument holds for Monge and Elkan's recursive matching scheme. We did implement

all measures by ourselves.

Some remarks on the implementation of the measures and on the chosen parameter settings:

- The parameter a in the combination edit distance, which determines the distribution of the weight of the edit distance and longest common subsequence, is set to 0.5.
- The longest common substring and the longest common subsequence are divided by the average string length to define the corresponding similarity measures. Dividing by the length of the shorter string would overly favor the shorter strings, and dividing by the length of the longer string would overly favor the longer strings.
- We have chosen to implement the longest common substring measure that repeatedly finds and removes the longest common substring. The minimum length of a longest common substring is set to two.
- The maximum length of the common prefix in Jaro-Winkler is set to four.
- The Jaro-Winkler with longer string adjustment measure, uses the original Jaro-Winkler distance to compare two strings if they do not meet the criteria. The comparison between a string that is compared with the Jaro-Winkler with longer string adjustment measure and a string that is compared with the original Jaro-Winkler distance measure, is made one-to-one (i.e., the highest similarity score is considered the best match).
- The Jaro-Winkler with longer string adjustment measure can only be applied when the strings do meet some criteria (see Subsection 3.1.2). If the strings do not meet the criteria, the original Jaro-Winkler measure is used to measure the distance between the strings. In case some strings in the database are compared with the input string using the Jaro-Winkler with longer string adjustment measure, and others are compared with the input string using the regular Jaro-Winkler measure, the string that has the highest absolute similarity value is defined as more similar (note that both the Jaro-Winkler with longer string adjustment measure and the regular Jaro-Winkler measure return a number between zero and one).
- The bi- and trigram measures are based on padded grams.
- The Tf-idf is based on padded trigrams as tokens.

4.2.2 Street

For input strings of the type street, we have chosen the similarity measures in Table 4 to compare:

Edit distance	Bigrams Overlap coefficient
Longest common substring	Bigrams Jaccard similarity
Longest common subsequence	Bigrams Dice's coefficient
Jaro	Trigrams Overlap coefficient
Jaro-Winkler	Trigrams Jaccard Similarity
Sorted Jaro-Winkler	Trigrams Dice's coefficient
Jaro-Winkler with longer string adjustment	Tf-idf

Table 4: Similarity measures for street

The list of tested similarity measures for street names differs only in measures of the edit distance type:

- We only test the edit distance itself, but no extensions. We expect the edit distance not to perform well on street names, as the variety in inconsistencies between strings of the type street is large. As the edit distance is mainly suitable for correcting typographical errors, it will not be able to deal with omissions, abbreviations and extensions in strings, which we expect to occur often in street names. The affine gap distance can correct for abbreviations, but has the same disadvantages as the basic edit distance measure, which is the reason we did not implement the affine gap distance.

There are no differences in parameter settings in comparison with town matching.

4.2.3 Zip code algorithm

For zip code matching we determined it is rather difficult to compare different similarity measures. Most often, an erroneous zip code can be changed into many other zip codes by simply inserting, deleting or substituting a number/letter or by transposing two characters. It is therefore more convenient to let the "zip code matching algorithm" correct in the way we want it to correct, than to perform a comparison by manipulating zip codes and check how many times the different similarity measures find the correct zip code. Testing zip codes with real data is also difficult, as errors are not easy to spot, because a typing error in the zip code will often lead to a zip code pointing to a different location. If the zip code does not point to a location, determining which zip code is targeted might be difficult, if not impossible (note that we needed to define manually which location is targeted by the user).

Zip codes do often have the same type of structure, with numbers/letters at the beginning of the string corresponding to a region in the country, and numbers/letters further in the string pointing to a specific location in the region (see for example the zip code structure in the United Kingdom, Germany, France, Italy, Belgium and The Netherlands). This hierarchical structure we exploit by

assigning more value to characters at the beginning of the string. In this part we describe the algorithm we developed for matching zip codes.

We distinguish two cases:

- Only the zip code is given as input (Algorithm 1).
- Next to the zip code, also the name of the town and/or a street name are given (Algorithm 2).

```
Input: Input zip code and all zip codes in the database
Output: Most similar zip code (may be exactly matching), or "no similar zip code is found"
if input zip code matches exactly with a zip code in the database then
    return the zip code;
else
    find zip code(s) in the dataset with the longest common prefix;
    if longest common prefix is at least half the size of a regular zip code (rounded up) then
        for all zip codes having the longest common prefix do
            remove common prefixes;
            perform Damerau-Levenshtein on the remaining parts;
        end
    else
        return "No Match";
    end
end
```

Algorithm 1: Only the zip code

The idea is that the beginning of the zip code string is most important in defining the location of the zip code. If an error is made in the beginning of the string, it is hard to define which region is meant by the user and we return a "no-match". Otherwise, we assume that the characters in the longest common prefix are correctly entered and consider only those zip codes that have the longest common prefix in common with the input zip code. To determine the best zip code from that subset of zip codes, Damerau-Levenshtein is used on the part of the zip codes that is not in the common prefix. This is because of the assumption we make that the zip code differs most of the time from the targeted zip code due to a transposition, substitution, insertion or deletion, which the Damerau-Levenshtein can handle well.

For matching zip codes when also the name of the town and/or the name of the street are given, we use the same algorithm, except that we apply Damerau-Levenshtein in all cases (see details in

the description in Algorithm 2).

```
Input: Input zip code and set of candidate zip codes (see below for definition)
Output: Most similar zip code (may be exactly matching)
if input zip code matches exactly with a candidate zip code then
    return the zip code;
else
    find zip code(s) in the candidate set with the longest common prefix;
    if longest common prefix is at least half the size of a regular zip code (rounded up) then
        for all zip codes having the longest common prefix do
            remove common prefixes;
            perform Damerau-Levenshtein on the remaining parts;
        end
    else
        perform Damerau-Levenshtein on all complete zip codes in the candidate set;
    end
end
```

Algorithm 2: Zip code, town and street

In case also a town and/or street are given as input, we assume that first a preprocessing procedure is applied on the database by removing locations that are clearly not targeted, with the goal to speed up the search. For details about the applied preprocessing procedure, see Section 5.2. For now, it is enough to know that only a subset of zip codes remains when a town and/or street are given as input, from which we need to determine the best zip code(s) (the "best" zip code(s) are the zip codes that are most similar to the input zip code). This subset will often be something like: the zip codes in a certain town or the zip codes corresponding to a certain street. In the algorithm, a zip code in the subset is called a *candidate* zip code.

The algorithm is based on the same idea as Algorithm 1, in which only a zip code is given as input. The main difference is that we do not return "no-match", if no zip code can be found having a long enough prefix in common with the input zip code. This is because the set of candidate zip codes is often rather small and applying the Damerau-Levenshtein measure on the complete zip codes in the subset, to determine which zip codes are most similar, will be appropriate.

In the worst case, the Damerau-Levenshtein distance needs to be calculated between the input string and all strings in the database. The time complexity of the zip code algorithm is therefore $O(|S_i| * |S_r| * |zipcodes|)$, with S_i the size of the input zip code, S_r the size of a regular zip code

(we assume here that the size of a zip code in the database is always the same) and $|zipcodes|$ the number of zip codes in the database. The search for an exact match ($O(log(|zipcodes|))$, and calculation of the longest common prefix ($O(|S_r| * |zipcodes|)$, although often much smaller because only a small subset of the zip codes will have the longest common prefix), have a smaller time complexity. The search for an exact match costs ($O(log(|zipcodes|))$, because all zip codes can be stored in a set in C++ (a set is an ordered list). In a set, it is possible to find in logarithmic time if it contains a certain value[5].

4.3 Datasets

To test the similarity measures we selected to match town and street names, we make use of four datasets consisting of locations that are presented to the route planner by different companies. CQM gave permission to use the datasets in the thesis. The numbers of locations in The Netherlands in the datasets is respectively 2173, 2039, 2026 and 617, summing up to a total of 6855 locations.

First we determined the error types that do occur in the datasets. The main reason we did not define error types beforehand, is that it was unclear how specific an error type needs to be defined (e.g., in dataset 1 and 2 there is a clear distinction between "long substring additions" and "short substring additions"). Next to this, also combinations of error types may occur, which may or may not need to be analyzed in their own category. For example, a character-error in a string with multiple abbreviations will likely have not much influence on the matching and can better be categorized as an "abbreviation error". In Table 5, the categorized error types are given, together with their relative frequency per dataset (proportional to the total number of errors in the dataset for that string type). An error is defined as "addition of a long substring", if the addition was six or more characters long. The selected measures are tested on the error types given in the table.

[5] `http://www.cplusplus.com/reference/set/set/find/`

Town				
	Dataset (# unique errors)			
Error type	1 (417)	2 (613)	3 (38)	4 (20)
1- & 2- character errors	8.04%	3.42%	36.84%	40.00%
Addition of long substring	81.80%	92.82%	44.74%	0%
Addition of short substring	10.16%	3.76%	2.63%	60.00%
Abbreviation or Omission	0%	0%	15.79%	0%
Street				
	Dataset (# unique errors)			
Error type	1 (444)	2 (573)	3 (233)	4 (35)
1- & 2- character errors	25.68%	17.98%	47.21%	51.43%
Abbreviations	10.81%	4.71%	9.01%	14.29%
Abbreviations + character errors	2.25%	1.22%	21.46%	2.86%
Addition of substring	57.66%	73.12%	11.16%	28.57%
Omission of substring	2.7%	2.27%	6.43%	0%
Omission + addition	0.9%	0.70%	4.72%	0%

Table 5: Error types and relative frequency of the errors per dataset

In table 6 an example is given for each error type.

Town	
Error type	**Example**
1- & 2- character errors	Input: Nieuw Lekkerland. Targeted: Nieuw-Lekkerland
Addition of long substring	Input: Ridderkerk (Ridderkerk-West). Targeted: Ridderkerk
Addition of short substring	Input: Rozenburg ZH. Targeted: Rozenburg
Abbreviation or Omission	Input: Alphen A/D Rijn. Targeted: Alphen aan den Rijn
Street	
Error type	**Example**
1- & 2- character errors	Input: Dommelsraat. Targeted: Dommelstraat
Abbreviations	Input: P. Joostenstraat. Targeted: Pieter Joostenstraat
Abbreviations + character errors	Input: Verl.Hoofdweg. Targeted: Verlengde Hoofdweg
Addition of substring	Input: Welplaatweg , havennummer. Targeted: Welplaatweg
Omission of substring	Input: Herbert Dowweg. Targeted: Herbert Henry Dowweg
Omission + addition	Input: Groot Mijdrechtweg Targeted: Groot Mijdrechtstraat

Table 6: Examples of error types

When analyzing the datasets (both quantitative and qualitative), the first thing to notice is that the two largest datasets are very similar with respect to error type and corresponding frequency for both place and street names. Next to this, some errors occur way more often than others. We define this last property as a *skewed* error distribution. To give an example, in the largest dataset there are 563 town strings with a long additional substring between round brackets after the town itself (26% of all locations, 92.82% of the total number of errors). The other dataset has the same error in 16.7% of the town names (81.8% of the total number errors in town names in the dataset). The additional substrings consist in almost all cases of information where the targeted location is in the place (e.g., "city center", "port", or the name of the quarter). In those two datasets this error occurs 904 times in town names. We decided to analyze 180 (+/- 20%) of the strings with this error type. Analyzing 180 strings having this error gives a good view on how the measures perform on the specific error type, while analyzing all would not be suitable due to time limitations. It seems unlikely that these errors are made by a user typing locations in the route planner. Instead, it is more likely that the users paste the address information into the input fields. For the analysis this does not make difference, as we have categorized on error types. The frequency of the error types will however be different than when all locations would be really typed by a user.

The other frequently occurring inconsistencies in the two datasets for towns are: additions of

the abbreviation of the province (short additions) and the omission of a "-" between two words, while the number of clear typographical errors is almost negligible. Both omission of a "-" and typographical errors are categorized as *character-edit error*, which is the term we use in the thesis for these type of errors due to an edit operation, such as insertion, deletion, substitution or transposition.

We also noticed that in those two datasets the street errors are more varying than errors occurring for town strings. Again, street names often have an additional substring with information about the locations (mostly "port" or "industrial estate"), though not between round brackets. This error type occurs in 413 street names in the largest dataset and in 237 street names in the other one. Due to time limitations we considered only 130 errors of this type (20% of the total number of errors of this type). Compared with place names there are more abbreviations used (+/- 2.5% of the total number of street names) and more typographical errors (+/- 5% of the street names has a typographical error). The two datasets are also similar in the sense that they have a large number of common locations. While interpreting the test results, we took into account the skewness and similarity of the two sets.

The third dataset has only a few errors in the place names, in the majority consisting of typographical errors or an addition of a small substring. In the street names errors occur more frequently (often a typographical error (6%) or an abbreviation (3.5%), but also omissions and additional substrings are present). In general, this set is less *skewed* in errors than the other two large sets.

The smallest dataset does not have many errors in town and street names, but the strings that do not agree with a string from the database differ due to typographical errors and small additional substrings.

The question rises how representative these four datasets are. This is not an easy question to answer however, because it mainly depends on the application and types of errors expected in the application. For all different error types, we have enough records to compare the performance of different measures, which makes it possible to define which measures are best to use in an encountered application with its own characteristics. Only for the error type "omissions and abbreviations for towns" we have only a few locations in the dataset. However, the performance of similarity measures on omissions and abbreviations on street names will also give some information about the performance on town names having these errors. For street names we have more errors of these types (169 abbreviations and 40 omissions). Next to this, one may argue that these error types are less common for town strings, although, again, this will probably depend on the application.

Also interesting but difficult, is how representative errors are with respect to their error type. The main observation is that for many substring additions in the datasets the addition is in the same way appended to the targeted string (for both towns and streets). In the majority of the strings having this error, the appended substring is placed between parentheses after the targeted string. We have to take this into account when analyzing the results and be careful with drawing general conclusions about the performance of the similarity measures on this error type. For all other errors, we belief that the results give a good representation of the performance of the measures on the error type, as the errors are made on several different positions in strings with different characteristics.

4.4 Test results

In this section the test results are given and interpreted, leading to the conclusion concerning which measures are best to match town and street strings. We give the main conclusions already in advance, leaving the detailed analysis in the next pages for the interested reader:

- Trigrams Jaccard and trigrams Dice's turn out best for matching both towns and street names.
  - If a lot of omissions or additions of substrings are expected, the trigrams Dice's is preferred above the trigrams Jaccard.
  - If the majority of the errors are small typographical errors, trigrams Jaccard is preferred.
  - Both require the same computation time and are fast enough to be applicable in practice.
- All measures are *dominated* by trigrams Jaccard and/or trigrams Dice's, except for the longest common subsequence measure and the tf-idf measure. Dominated means that the performance is worse or equally good on all error types.
- The longest common subsequence measure is only slightly better than the trigrams measures in dealing with typographical errors, but is clearly worse in dealing with other error types, such as additions or omissions of substrings. The measure requires more computation time than the trigrams measures.
- The tf-idf measure is better than the trigrams measures in dealing with abbreviations, but performs worse in correcting for character-errors and additional substrings. The main reason we do prefer the trigrams measures above the tf-idf measure, is that the tf-idf measure sometimes fails to return a trivial location, when an error introduces a trigram that occurs only a few times in the database. Strings in the database having this rare trigram are occasionally overly favored and returned as output. Besides, we expect that the trigrams measures are still able to find the targeted location a large percentage of the time when one of the input strings is abbreviated. We assume this, because with the information about the location that can be

extracted from the other input fields, a small subset of candidate strings remains, from which the trigrams measure will often select the targeted string. It is likely that from the small subset, the targeted string has the most trigrams in common with the abbreviated string and will be rated as most similar. Another reason is that abbreviations are likely to occur less often than character-edit errors.

4.4.1 Results Town

In this section we give the detailed interpretation of the results of matching town strings. In Table 7 to Table 10, the results are given for the four error types we defined for town strings. In each table the following information is provided: the total number of errors of the error type and for each measure how often the targeted string is given as first, second or third suggestion, or not given as suggestion at all, with the corresponding percentages. Each table will be interpreted in its own subsection, with the goal to determine at the end which similarity measure has the best overall performance for matching towns.

Total number of errors: 83				
Similarity measure	# first	# second	# third	# no suggestion
Edit distance	78 (93.98%)	2 (2.41%)	0 (0%)	3 (3.61%)
Damerau-Levenshtein distance	78 (93.98%)	2 (2.41%)	0 (0%)	3 (3.61%)
Jaro	76 (91.57%)	6 (7.23%)	0 (0%)	1 (1.20%)
Jaro-Winkler	77 (92.78%)	3 (3.61%)	2 (2.41%)	1 (1.20%)
Sorted Jaro-Winkler	58 (69.88%)	5 (6.02%)	2 (2.41%)	18 (21.69%)
Jaro-Winkler with Longer String Adjustment	75 (90.37%)	5 (6.02%)	1 (1.20%)	2 (2.41%)
Longest Common Subsequence	78 (93.98%)	4 (4.82%)	0 (0%)	1 (1.20%)
Longest Common Substring	77 (92.78%)	1 (1.20%)	0 (0%)	5 (6.02%)
Combination Levenshtein	80 (96.39%)	2 (2.41%)	0 (0%)	1 (2.20%)
Bigrams Overlap	64 (77.11%)	9 (10.84%)	8 (9.64%)	2 (2.41%)
Bigrams Jaccard	79 (95.18%)	4 (4.82%)	0 (0%)	0 (0%)
Bigrams Dice's	78 (93.98%)	3 (3.61%)	2 (2.41%)	0 (0%)
Trigrams Overlap	70 (84.34%)	6 (7.23%)	5 (6.02%)	2 (2.41%)
Trigrams Jaccard	78 (93.98%)	4 (4.82%)	1 (1.20%)	0 (0%)
Trigrams Dice's	77 (92.78%)	3 (3.61%)	3 (3.61%)	0 (0%)
Tf-idf	79 (95.18%)	0 (0%)	0 (0%)	4 (4.82%)

Table 7: Town: 1-character-edit and 2-character-edit errors

Total number of errors: 205				
Similarity measure	# first	# second	# third	# no suggestion
Edit distance	99 (48.29%)	23 (11.22%)	9 (4.39%)	74 (36.10%)
Damerau-Levenshtein distance	98 (47.80%)	23 (11.22%)	9 (4.39%)	75 (36.58%)
Jaro	191 (93.16%)	11 (5.37%)	2 (0.98%)	1 (0.49%)
Jaro-Winkler	199 (97.07%)	4 (1.95%)	1 (0.49%)	1 (0.49%)
Sorted Jaro-Winkler	45 (21.95%)	5 (2.44%)	3 (1.46%)	152 (74.15%)
Jaro-Winkler with Longer String Adjustment	166 (80.98%)	18 (8.78%)	13 (6.34%)	8 (3.90%)
Longest Common Subsequence	117 (57.07%)	16 (7.80%)	8 (3.90%)	64 (31.23%)
Longest Common Substring	130 (63.41%)	17 (8.29%)	8 (3.90%)	50 (24.40%)
Combination Levenshtein	110 (53.66%)	20 (9.76%)	8 (3.90%)	67 (32.68%)
Bigrams Overlap	190 (92.68%)	9 (4.39%)	6 (2.93%)	0 (0%)
Bigrams Jaccard	134 (65.37%)	38 (18.54%)	4 (1.95%)	29 (14.14%)
Bigrams Dice's	171 (83.14%)	10 (4.88%)	9 (4.39%)	15 (7.32%)
Trigrams Overlap	198 (96.58%)	5 (2.44%)	2 (0.98%)	0 (0%)
Trigrams Jaccard	194 (94.63%)	6 (2.93%)	2 (0.98%)	3 (1.46%)
Trigrams Dice's	197 (96.10%)	5 (2.44%)	3 (1.46%)	0 (0%)
Tf-idf	189 (92.19%)	8 (3.90%)	2 (0.98%)	6 (2.93%)

Table 8: Town: Addition of long substring

Total number of errors: 81				
Similarity measure	# first	# second	# third	# no suggestion
Edit distance	72 (88.89%)	5 (6.18%)	3 (3.70%)	1 (1.23%)
Damerau-Levenshtein distance	72 (88.89%)	5 (6.18%)	3 (3.70%)	1 (1.23%)
Jaro	75 (92.60%)	3 (3.70%)	0 (0%)	3 (3.70%)
Jaro-Winkler	76 (93.83%)	2 (2.47%)	0 (0%)	3 (3.70%)
Sorted Jaro-Winkler	70 (86.41%)	5 (6.18%)	0 (0%)	6 (7.41%)
Jaro-Winkler with Longer String Adjustment	76 (93.83%)	2 (2.47%)	0 (0%)	3 (3.70%)
Longest Common Subsequence	77 (95.07%)	1 (1.23%)	0 (0%)	3 (3.70%)
Longest Common Substring	79 (97.53%)	2 (2.47%)	0 (0%)	0 (0%)
Combination Levenshtein	77 (95.07%)	0 (0%)	1 (1.23%)	3 (3.70%)
Bigrams Overlap	73 (90.12%)	5 (6.18%)	1 (1.23%)	2 (2.47%)
Bigrams Jaccard	78 (96.30%)	3 (3.70%)	0 (0%)	0 (0%)
Bigrams Dice's	81 (100.00%)	0 (0%)	0 (0%)	0 (0%)
Trigrams Overlap	81 (100.00%)	0 (0%)	0 (0%)	0 (0%)
Trigrams Jaccard	81 (100.00%)	0 (0%)	0 (0%)	0 (0%)
Trigrams Dice's	81 (100.00%)	0 (0%)	0 (0%)	0 (0%)
Tf-idf	78 (96.30%)	3 (3.70%)	0 (0%)	0 (0%)

Table 9: Town: Addition of short substring

Total number of errors: 6				
Similarity measure	# first	# second	# third	# no suggestion
Edit distance	3	0	2	1
Damerau-Levenshtein distance	3	0	2	1
Jaro	3	0	1	2
Jaro-Winkler	3	0	1	2
Sorted Jaro-Winkler	2	0	0	4
Jaro-Winkler with Longer String Adjustment	1	2	1	2
Longest Common Subsequence	5	1	0	0
Longest Common Substring	5	0	1	0
Combination Levenshtein	4	1	0	1
Bigrams Overlap	4	1	0	1
Bigrams Jaccard	5	1	0	0
Bigrams Dice's	5	1	0	0
Trigrams Overlap	3	2	1	0
Trigrams Jaccard	5	1	0	0
Trigrams Dice's	5	1	0	0
Tf-idf	5	1	0	0

Table 10: Town: Abbreviations, or omission of a substring

1-character-edit and 2-character-edit errors (Table 7)
The conclusion that can be drawn immediately from Table 7 is that most measures perform rather well in correcting for typographical errors, as you would expect. The combination Levenshtein finds the right string in no less than 80 out of 83 comparisons. Out of the other three cases, the right string is given as second suggestion twice.

The three measures that perform clearly worse than others are sorted Jaro-Winkler, bigrams overlap and trigrams overlap. The reason sorted Jaro-Winkler is often not returning the correct string, is because quite a few errors consist of a missing "-" between two words, or an added space in a single word string, both leading to 2 separate words. If sorting the words leads to a different word order, sorted Jaro-Winkler gives a low matching value to the original string, as the prefix is not common and the identical characters may not be within matching range. Bigrams and trigrams overlap have the property to highly favor shorter strings in the database, from which the trigrams are also in the set of trigrams in the input string. This is because the number of common trigrams is divided by the number of trigrams of the shorter string. For example, bigrams and trigrams overlap rate the pair ("Amsterdam", "Amsterdam Zuidoost") more similar than the pair ("Amsterdam-Zuidoost", "Amsterdam Zuidoost"). To conclude, Sorted Jaro-Winkler, Bigrams Overlap and Trigrams Overlap are no candidates to match towns, because a good measure should be able to deal with these inconsistencies. Hence, we neglect them in our further analysis.

Addition of long substring (Table 8)
First note that from the 205 errors of this type, 180 are additions between parentheses after the targeted string, as described before in the datasets section. In total, 203 out of the 205 long additions are added after the string corresponding to the town. For clarity, we repeat that an addition is categorized as a long addition if it contains at least six characters.

The Levenshtein and Damerau-Levenshtein distance do not perform well on these type of errors, as expected. The minimum Levenshtein distance is the difference between the string lengths, which lead to a Levenshtein distance between the input and targeted string of at least six in all cases. The longest common subsequence and longest common substring are also not performing very well, with the correct string in first position in around 60% of the cases. This is because they take into account subsequences and substrings of the additions, which overly favors longer strings having characters in common with the addition. Also the combination of the Levenshtein distance and longest common subsequence, the combination Levenshtein, does not produce solid results.

Although it looks like Jaro and Jaro-Winkler perform excellent on matching with added substrings, it is mainly due to the characteristics of the datasets. As described before, 203 of the

205 additions are behind the original string. If additions would be before the string, Jaro has less matching characters and Jaro-Winkler not a common prefix anymore. For the two input strings where the addition was before the actual town string, the targeted string was once given as third suggestion and once not given as suggestion at all. So we have to be careful not to over-evaluate Jaro and Jaro-Winkler. Jaro-Winkler with longer string adjustment is not an appropriate measure, as it sometimes gives the preference for a (clearly non-targeted) very short string, while the input string is long. This indicates that the normal Jaro-Winkler score and the adjusted score cannot be compared one-to-one. Hence, we neglect the Jaro-Winkler with longer string adjustment.

Both the grams and tf-idf measures yield good results. Notice that the trigrams Jaccard and trigram Dice's clearly outperform the bigrams Jaccard and bigrams Dice's, with trigrams Dice's returning the correct string in the top three in all cases.

Addition of short substring (Table 9)
Notice that in general, the measures produce clearly better results than when matching with longer added substrings. The Levenshtein and Damerau-Levenshtein performing worst and the trigrams, again, giving the best results. For the three small substring additions in front of the town strings, the Jaro and Jaro-Winkler measures do not perform well (leading to three cases where the correct town is not given as a suggestion). Not much additional information about the strengths of the measures can be deducted from the results.

Abbreviation, or omission of a substring (Table 10)
The number of abbreviations and omission of substrings is only six (two omissions and four abbreviations). Therefore, these error types are grouped together. Another reason to do this, is because the error types are related. When abbreviating, also a substring (or more substrings) is omitted. The difference with an omission is that an abbreviation preserves a piece of the original string (often one or a few characters with a dot), while an omission removes the complete substring. Analyzing omissions and abbreviations when matching street names will also give information about matching towns when facing this type of error. The number of town strings having this error is simply too small to draw conclusions.

4.4.2 Results Street

In this subsection we give the detailed interpretation of the results of matching street strings. On page 41 and page 42, from Table 11 to Table 16, the results are given for the six error types we defined for street strings. The tables have the same lay-out as the tables used to present the results for matching town strings.

Total number of errors: 345				
Similarity measure	# first	# second	# third	# no suggestion
Edit distance	326 (94.50%)	8 (2.32%)	3 (0.86%)	8 (2.32%)
Jaro	305 (88.40%)	12 (3.48%)	6 (1.74%)	22 (6.38%)
Jaro-Winkler	312 (90.43%)	14 (4.06%)	5 (1.45%)	14 (4.06%)
Sorted Jaro-Winkler	284 (82.32%)	11 (3.19%)	6 (1.74%)	44 (12.75%)
Jaro-Winkler with Longer String Adjustment	300 (86.96%)	15 (4.35%)	6 (1.74%)	24 (6.95%)
Longest Common Subsequence	335 (97.10%)	4 (1.16%)	1 (0.29%)	5 (1.45%)
Longest Common Substring	319 (92.46%)	6 (1.74%)	1 (0.29%)	19 (5.51%)
Bigrams Overlap	157 (45.51%)	40 (11.59%)	19 (5.51%)	129 (37.39%)
Bigrams Jaccard	327 (94.78%)	10 (2.90%)	2 (0.58%)	6 (1.74%)
Bigrams Dice's	319 (92.46%)	9 (2.61%)	4 (1.16%)	13 (3.77%)
Trigrams Overlap	231 (66.96%)	27 (7.83%)	11 (3.19%)	76 (22.03%)
Trigrams Jaccard	328 (95.07%)	9 (2.61%)	2 (0.58%)	6 (1.74%)
Trigrams Dice's	317 (91.88%)	9 (2.61%)	5 (1.45%)	14 (4.06%)
Tf-idf	289 (83.76%)	19 (5.51%)	15 (4.35%)	22 (6.38%)

Table 11: Street: 1-character-edit and 2-character-edit errors

Total number of errors: 100				
Similarity measure	# first	# second	# third	# no suggestion
Edit distance	33 (33.00%)	6 (6.00%)	1 (1.00%)	60 (60.00%)
Jaro	24 (24.00%)	8 (8.00%)	2 (2.00%)	66 (66.00%)
Jaro-Winkler	34 (34.00%)	7 (7.00%)	0 (0%)	59 (59.00%)
Sorted Jaro-Winkler	37 (37.00%)	6 (6.00%)	2 (2.00%)	55 (55.00%)
Jaro-Winkler with Longer String Adjustment	22 (22.00%)	6 (6.00%)	2 (2.00%)	55 (55.00%)
Longest Common Subsequence	56 (56.00%)	10 (10.00%)	4 (4.00%)	30 (30.00%)
Longest Common Substring	27 (27.00%)	9 (9.00%)	6 (6.00%)	58 (58.00%)
Bigrams Overlap	20 (20.00%)	9 (9.00%)	10 (10.00%)	61 (61.00%)
Bigrams Jaccard	46 (46.00%)	6 (6.00%)	4 (4.00%)	44 (44.00%)
Bigrams Dice's	44 (44.00%)	7 (7.00%)	5 (5.00%)	44 (44.00%)
Trigrams Overlap	36 (36.00%)	11 (11.00%)	2 (2.00%)	51 (51.00%)
Trigrams Jaccard	54 (54.00%)	6 (6.00%)	4 (4.00%)	36 (36.00%)
Trigrams Dice's	53 (53.00%)	10 (10.00%)	3 (3.00%)	34 (34.00%)
Tf-idf	88 (88.00%)	0 (0%)	1 (1.00%)	11 (11.00%)

Table 12: Street: Abbreviations

Total number of errors: 69				
Similarity measure	# first	# second	# third	# no suggestion
Edit distance	22 (31.88%)	3 (4.35%)	3 (4.35%)	41 (59.42%)
Jaro	13 (18.84%)	4 (5.80%)	2 (2.90%)	50 (72.46%)
Jaro-Winkler	18 (26.09%)	8 (11.59%)	3 (4.35%)	40 (57.97%)
Sorted Jaro-Winkler	19 (27.54%)	5 (7.25%)	3 (4.35%)	42 (60.87%)
Jaro-Winkler with Longer String Adjustment	12 (17.39%)	3 (4.35%)	2 (2.90%)	52 (75.36%)
Longest Common Subsequence	28 (40.58%)	5 (7.25%)	7 (10.14%)	29 (42.03%)
Longest Common Substring	19 (27.54%)	8 (11.59%)	2 (2.90%)	40 (57.97%)
Bigrams Overlap	10 (14.49%)	6 (8.70%)	1 (1.45%)	52 (75.36%)
Bigrams Jaccard	26 (37.68%)	7 (10.14%)	4 (5.80%)	32 (46.38%)
Bigrams Dice's	25 (36.23%)	9 (13.05%)	8 (11.59%)	27 (39.13%)
Trigrams Overlap	20 (28.97%)	3 (4.35%)	3 (4.35%)	43 (62.33%)
Trigrams Jaccard	32 (46.38%)	7 (10.14%)	6 (8.70%)	24 (34.78%)
Trigrams Dice's	31 (44.93%)	5 (7.25%)	10 (14.49%)	23 (33.33%)
Tf-idf	51 (73.91%)	6 (8.70%)	1 (1.45%)	11 (15.94%)

Table 13: Street: Abbreviations with character-edit errors

Total number of errors: 198				
Similarity measure	# first	# second	# third	# no suggestion
Edit distance	98 (49.49%)	9 (4.55%)	7 (3.54%)	84 (42.42%)
Jaro	160 (80.80%)	7 (3.54%)	1 (0.51%)	30 (15.15%)
Jaro-Winkler	155 (78.27%)	7 (3.54%)	1 (0.51%)	35 (17.68%)
Sorted Jaro-Winkler	81 (40.90%)	7 (3.54%)	3 (1.52%)	107 (54.04%)
Jaro-Winkler with Longer String Adjustment	111 (56.05%)	13 (6.57%)	9 (4.55%)	65 (32.83%)
Longest Common Subsequence	118 (59.60%)	10 (5.05%)	5 (2.52%)	65 (32.83%)
Longest Common Substring	95 (47.97%)	13 (6.57%)	3 (1.52%)	87 (43.94%)
Bigrams Overlap	114 (57.58%)	18 (9.09%)	10 (5.05%)	56 (28.28%)
Bigrams Jaccard	90 (45.45%)	11 (5.55%)	6 (3.03%)	91 (45.97%)
Bigrams Dice's	135 (68.18%)	20 (10.10%)	3 (1.52%)	40 (20.20%)
Trigrams Overlap	172 (86.86%)	7 (3.54%)	3 (1.52%)	16 (8.08%)
Trigrams Jaccard	161 (81.31%)	17 (8.59%)	4 (2.02%)	16 (8.08%)
Trigrams Dice's	175 (88.38%)	9 (4.55%)	2 (1.01%)	12 (6.06%)
Tf-idf	156 (78.79%)	11 (5.55%)	10 (5.05%)	21 (10.61%)

Table 14: Street: Addition of a substring

Total number of errors: 40				
Similarity measure	# first	# second	# third	# no suggestion
Edit distance	6 (15.00%)	3 (7.50%)	2 (5.00%)	29 (72.50%)
Jaro	15 (37.50%)	2 (5.00%)	2 (5.00%)	21 (52.50%)
Jaro-Winkler	16 (40.00%)	3 (7.50%)	0 (0%)	21 (52.50%)
Sorted Jaro-Winkler	17 (42.50%)	3 (7.50%)	1 (2.50%)	19 (47.50%)
Jaro-Winkler with Longer String Adjustment	13 (32.50%)	2 (5.00%)	2 (5.00%)	23 (57.50%)
Longest Common Subsequence	19 (47.50%)	4 (10.00%)	4 (10.00%)	13 (32.50%)
Longest Common Substring	24 (60.00%)	3 (7.50%)	1 (2.50%)	12 (30.00%)
Bigrams Overlap	27 (67.50%)	4 (10.00%)	3 (7.50%)	6 (15.00%)
Bigrams Jaccard	21 (52.50%)	3 (7.50%)	4 (10.00%)	12 (30.00%)
Bigrams Dice's	33 (82.50%)	2 (5.00%)	1 (2.50%)	4 (10.00%)
Trigrams Overlap	35 (82.50%)	2 (5.00%)	0 (0%)	3 (7.50%)
Trigrams Jaccard	28 (70.00%)	2 (5.00%)	1 (2.50%)	9 (22.50%)
Trigrams Dice's	36 (90.00%)	1 (2.50%)	0 (0%)	3 (7.50%)
Tf-idf	36 (90.00%)	2 (5.00%)	0 (0%)	2 (5.00%)

Table 15: Street: omission of a substring

Total number of errors: 12				
Similarity measure	# first	# second	# third	# no suggestion
Edit distance	5	2	0	5
Jaro	4	0	1	7
Jaro-Winkler	4	0	1	7
Sorted Jaro-Winkler	3	2	1	6
Jaro-Winkler with Longer String Adjustment	4	1	0	7
Longest Common Subsequence	6	1	0	5
Longest Common Substring	6	0	0	6
Bigrams Overlap	3	0	0	9
Bigrams Jaccard	7	1	0	4
Bigrams Dice's	8	0	0	4
Trigrams Overlap	3	0	1	8
Trigrams Jaccard	7	1	0	4
Trigrams Dice's	8	0	0	4
Tf-idf	8	0	1	3

Table 16: Street: omission of a substring + addition of a substring

1-character-edit and 2-character-edit errors (Table 11)
For the 1-character-edit and 2-character-edit errors we would expect similar matching quality as for matching town strings with the same error.

We first note that the sorted Jaro-Winkler, Jaro-Winkler with longer string adjustment, and the bigrams overlap and trigrams overlap measures are all not suitable to match street names, because of the same reasons why they are not appropriate to match town strings. The sorted Jaro-Winkler is not returning the targeted string when the input string consists of multiple words (e.g., due to an addition) and is sorted such that the order of words does not correspond to the targeted string. In that case the strings do not have a common prefix and the sorted Jaro-Winkler measure almost always fails to return the targeted string. The Jaro-Winkler with longer string adjustment sometimes over-evaluates a very short string, although it is clearly not targeted. The bigrams overlap and trigrams overlap similarity measures highly favor shorter strings in the database, because the number of common q-grams is divided by the number of q-grams in the shorter string. It may occur that q-grams of a short string are present in the input string, such that the short string is rated as most similar, while it is clearly not targeted. These observations are reflected in the test results. The bigrams overlap and trigrams overlap measures perform very bad when facing a character-edit error, the sorted Jaro-Winkler yields significantly worse results than Jaro and Jaro-Winkler on matching with a substring addition, as does the Jaro-Winkler with longer string adjustment.

The longest common subsequence measure performs best (the targeted string is first suggestion in 97.10% of the cases), but all other measures perform reasonably too (ranging from 95.07% as first suggestion for bigrams Jaccard to 83.76% for tf-idf). The bigrams are equally good as trigrams, with the Jaccards producing better results than Dice's. Jaro-Winkler performs slightly better than Jaro.

The reason that tf-idf has lower matching quality than the other similarity measures, is that it sometimes fails to return a trivial location when the character-edit error introduces a trigram that occurs only a few times in the database. In some occasions a string in the database having this trigram is then preferred above the, for the human eye obvious, targeted string. For example, tf-idf rates the pair (Mergelsweg, Lengelsweg) as more similar than the pair (Lengelseweg, Lengelsweg). Because "lsw" is a rarely occurring trigram, the tf-idf assigns much value to it. As a consequence, the street "Merge**lsw**eg" is rated as most similar.

Abbreviations (Table 12)
In the performance analysis, we merged abbreviations of single word strings (e.g., Voorstr. instead of Voorstraat), abbreviations of one word in a multiple word string (C. Drebbelstraat instead of

Cornelis Drebbelstraat), and abbreviations of more words in a multiple word string (e.g., Burg. v.d. Heuvelstraat instead of Burgemeester van den Heuvelstraat). Their frequencies are respectively 12, 69 and 19. It should be noticed that most measures are good in matching single word abbreviations (often "...straat" vs. "...str.") and most errors are made when several abbreviations are used, as you would expect.

Clearly the best measure to correct for abbreviations is the tf-idf measure, because it returns the targeted string in 88.00% of the cases as first suggestion. To compare, the longest common subsequence measure, second best in matching abbreviations, only finds the best string as first suggestion 56.00% of the time. Slightly worse than the longest common subsequence measure are trigrams Jaccard and trigrams Dice's. Bigrams Jaccard and bigrams Dice's perform significantly worse than the trigrams measures, but significantly better than all other measures. For example, Jaro fails to give the correct match as suggestion for 66 of the 100 input strings.

The reason tf-idf performs so well, is that it focuses on the trigrams in the input string that are in common with a string in the database. Suppose that from two strings in the database, the one that is most similar to the input string needs to be determined. From the structure of the tf-idf measure, see Subsection 3.2.1, we can see that if both strings share the same trigrams with the input string, but one of the two strings shares one extra trigram, that string is always valued most similar regardless the lengths of the strings. It is possible to show mathematically that this is true, but we will not go into it here. This observation applies directly on abbreviations. To clarify, consider the following example:

Assume that the targeted string is "Cornelis Drebbelstraat" and the input string is the abbreviated "C. Drebbelstraat". Both trigrams Jaccard and trigrams Dice's, would consider the pair (Drebbelstraat, C. Drebbelstraat) more similar than the pair (Cornelis Drebbelstraat, C. Drebbelstraat), due to the trigrams in "Cornelis" that are not shared, which lowers the similarity value. However, "Cornelis Drebbelstraat" and "C. Drebbelstraat" share one more trigram than "Drebbelstraat" and "C. Drebbelstraat", which is the (__C). Consequently, the tf-idf similarity measure values the pair (Cornelis Drebbelstraat, C. Drebbelstraat) as more similar.

Abbreviations with character-edit errors (Table 13)
The performance on abbreviations with character-edit errors is quite similar to the performance on matching abbreviations without character-edit errors. Tf-idf performs again clearly best. A small difference is that the trigrams Jaccard and trigrams Dice's measures produce better results than the longest common subsequence measure, which is a little surprising because the longest common subsequence measure has the best results for matching street names facing a character-edit error.

Substring additions (Table 14)
As described before in the datasets section, due to time limitations we selected a sample of 20% of the street names with a long additional substring (resulting in 130 street names). After this, we merged all substring additions: short, long, both before and after the name of the town itself.

The trigrams Dice's measure performs clearly best (88.38% of the targeted strings are given as first suggestion). The tf-idf, trigrams Jaccard, Jaro and Jaro-Winkler measures have a comparable performance on this error type, and are significantly better than the other measures. An interesting result is that if Jaro or Jaro-Winkler does not return the correct string as first suggestion, it often does not returns the string at all. This is due to additional substrings before the place name itself, which causes trouble for the matching range for Jaro and common prefix Jaro-Winkler, such that they are unable to detect the targeted string.

Substring omissions (Table 15)
For correcting substring omissions, we see that the tf-idf measure and trigrams Dice's have again the best performance (both returned the targeted string in 36 out of 40 occasions), followed by the other q-gram measures and the longest common substring measure. The longest common subsequence, Jaro, Jaro-Winkler and especially the edit distance (only 6 out of 40 times the edit distance returned the targeted string in first place) cannot deal well with an omission of a substring.

Omission and addition of substring (Table 16)
For this last error type, for which we have only 12 records, we analyze strings that have both an omission and addition of a substring. This may be at the same location in the string, such as a changed common suffix (e.g. "straat" instead of "weg"), but also at different locations in the string.

Although the sample size is small for this error type, we do not see much unexpected results, as the q-grams measures and tf-idf do again have good matching quality. Given the results for additions and omissions, this is what we would expect.

4.5 Conclusions

In this section, we determine if we can select the best similarity measure(s), applicable on town and/or street matching, based on all described results in the previous two subsections.

We already stated that sorted Jaro-Winkler, Jaro-Winkler with longer string adjustment, bi-

grams overlap and trigrams overlap are not suitable measures for both matching town and street names. Not taking into account these measures, we first determine if measures are *dominated* by others. With dominated by, we mean that the matching quality is equally good or worse on all error types.

Jaro and Jaro-Winkler perform worse than trigram Dice's for all encountered error types, both for town and street matching. Similarly, bigrams Jaccard is outperformed by trigrams Jaccard, and bigrams Dice's is outperformed by trigrams Dice's. In correcting for character-edit errors the bigrams seem to be equally good as the trigrams, but for all other error types the trigrams are substantially better. Exactly the same argument holds for the Levenshtein distance and its extensions: for character-edit errors the measures perform just about as good as the trigrams Jaccard measure, but for all other error types they are performing clearly worse. At last, the longest common substring measure does not have any advantages over the trigrams, as trigrams Jaccard yields better results for all error types. We can now safely conclude that those dominated measures have no advantages over the trigrams measures with respect to matching quality, which make them not the best candidates to select for matching town and/or street strings.

This simple check if some measures are dominated, results in the following list of only four candidate measures for both towns and streets, to analyze in more detail:

- Longest common subsequence
- Trigrams Jaccard
- Trigrams Dice's
- Tf-idf

We start the analysis of the matching quality of this four measures, by first comparing the longest common subsequence and trigrams Jaccard measure.

The difference in matching quality between the longest common subsequence and trigrams Jaccard measure if the inconsistency is a character-edit error, is quite small for both towns and street names. In that case, both measures perform rather well. The longest common subsequence measure yields slightly better results on character-edit errors in street names. The main difference is when additions or omissions occur, because the trigrams Jaccard measure performs clearly better than the longest common subsequence measure for those errors (more than 20% more correct first suggestions for both towns and streets on substring additions). The additions are often at the end of the strings in the database, as repeatedly stated before. Because this does not increase the performance of the trigrams measures, there is not much reason to relate the difference to data-issues.

The measures do not differ much in correcting abbreviations. Because the difference is small for character-edit errors and abbreviation errors, but large for additions or omissions, we prefer the trigrams Jaccard measure over the longest common subsequence measure for both streets and towns.

We now compare the trigrams Dice's and tf-idf measure and check if we can draw a similar conclusion.

The tf-idf measure has a poorer performance on matching character-edit-errors for street names, with sometimes unexpected results on seemingly trivial matches (note the example we gave before: "Lengelsweg" is not matched with "Lengelseweg"). The tf-idf also returns the targeted string less often when an addition occurs. Although the tf-idf yields much better results when strings are abbreviated, we expect that the trigrams Dice's measure should in that case be able to find the correct location, if more information is available about the location due to other input strings. Besides, character-edit errors and substring additions occur more often than abbreviations in the datasets. Of course, this can not be a general conclusion, but the assumption that a character-edit errors occur more often than abbreviations seems acceptable. The property of the tf-idf that it sometimes does not return the correct string when trivial, makes the use of the tf-idf measure in practice even less attractive. The conclusion is that we prefer the trigrams Dice's measure over the tf-idf measure for matching street strings.

For town matching we also prefer trigram Dice's over tf-idf. Although not visible in the datasets, it could easily be that some errors that may occur will lead to unexpected results for the tf-idf measure. Furthermore, the main advantage of tf-idf over trigrams Dice's, correcting for abbreviations, is less useful for towns, as little abbreviations are expected.

We now have only trigrams Jaccard and trigrams Dice's left as possible similarity measures for both town and street matching. The measures are of course similar in structure, with trigrams Jaccard dividing the number of common trigrams by the number of trigrams in the longer string and trigrams Dice's dividing the number of common trigrams by the average number of trigrams in both strings. Both measures do also often give the same suggestions. Consider again the results given in the tables. The trigrams Jaccard measure is better in dealing with character-edit errors than the Trigrams Dice's measure, but Trigrams Dice's is better suited for correcting additions and omissions. This result is as expected, considering the structure of both measures: trigrams Jaccard gives a larger "penalty" if two strings differ in length than the trigrams Dice's measure does. To see why this is the case, an example:

Assume the input string consists of the targeted string with an additional substring. The input

string has 20 trigrams and the targeted string only 8, of which 7 agree with the input string. How is the similarity of the targeted string valued, compared with a string having 18 trigrams of which 8 agree with the input string? As a reminder: the trigrams Jaccard similarity measure divides the number of common trigrams by the number of trigrams in the longer string and trigrams Dice's divides the number of common trigrams by the average number of trigrams in both strings. Trigrams Jaccard values the longer string as more similar, with $\frac{8}{20} > \frac{7}{20}$. Dice's defines the smaller (i.e., the targeted) string as clearly more similar: $\frac{7}{14} > \frac{8}{19}$. Here Dice's turns out to perform better, considering the assumption of an addition (note that omissions yield the same result). On the other side it may occur that trigrams Dice's overvalues a string that has a different string length, while the error does not consist of an addition or omission. Which measure is best, depends on the application. If the regular input strings are similar to the ones in the two largest datasets, with a lot of additional substrings, the trigrams Dice's measure is clearly the better choice. If input strings are considered to be of similar length as the targeted string in the large majority of the comparisons, trigrams Jaccard should be preferred.

We did not see room for improvement by testing different other measures. The similarity measures described in Chapter 3 which do not make use of q-grams will have similar disadvantages as other measures tested in this chapter. As it is already shown that padding q-grams increases performance, we will not test regular q-grams measures. Skip-grams we will not consider, because they are specialized in correcting for typographical errors. Next to this, also changing the parameter settings would not likely yield a different conclusion about which measures are best. As said before, it is clear that the trigrams measures perform better than other measures not making use of q-grams. Parameter settings such as for example, how to define the similarity measure for the longest common substring and longest common subsequence measure are unlikely to change this conclusion. Also tf-idf with bigrams we will not consider, because the trigrams seem to give more information about the strings than bigrams, which will also be reflected in the tf-idf scores. Hence, we conclude that trigrams Dice's and trigrams Jaccard are the best options for both town and street matching. Which of the two should be used, depends on the types of errors that are expected.

Computation time comparison

In Chapter 3, Table 2, the computational complexities are given for all measures. The trigrams measures are all calculated in $O(|s1|+|s2|)$, which is fastest of all measures given in the table. Note that it can be defined beforehand which padded trigrams the strings in the database consist of. This will considerably speed up the calculation of the trigrams Dice's measure and makes it definitely fast enough to be applicable in practice. See Chapter 5, in which the algorithm developed for matching locations will be described, for details about an efficient implementation of the trigrams measures. Trivially, the computation time of both the trigrams Jaccard and trigrams Dice's measure is similar.

5 Matching algorithm

In this chapter the matching algorithm is described. The algorithm decides which locations from the database are presented to the user, given the input strings. The difference with the previous chapter is that now may have several input strings (e.g. a town and a zip code) and we need to match it with a location in the database. The matching algorithm uses the trigrams Dice's similarity measure to match towns and streets, based on the test results in chapter 4. The trigrams Dice's similarity measures is preferred above the trigrams Jaccard measure, because of the practical reason CQM often encouters long additional substrings in its datasets. The algorithm also uses the developed zip code matching algorithm (described in Subsection 4.2.3).

The outline of the chapter is as follows. First, the so-called "output flowcharts" are presented, schematically representing the matching algorithm. Thereafter, the in the matching algorithm used preprocessing step is presented. The goal of the preprocessing step is to reduce the computation time of the matching algorithm. In the third part, implementation details are given and at the end, a comparison is made with the current matching algorithm of CQM, based on the final matching results.

We assume that we have a complete and correct database of locations. Advice on how to adapt the algorithm when this is not the case, is given in the "Practical Considerations" chapter; Chapter 6.

5.1 Output flowcharts

For all combinations of input fields that can be used (town, street and zip code), a flowchart is given which represents the output that the algorithm returns to the user.

The main observation on which the output flowcharts are created, is that if an input string is an exact match with a string of the same type in the database, it is likely that the user targets the corresponding location(s), unless information from other input strings indicates otherwise. The idea is to check if there are exact matches and first explore the locations corresponding to the exact matching string(s). The other locations given as output by the algorithm should make sure that the targeted location will be returned, in case the exact matching string(s) does not correspond to the targeted location. Note that with exploiting exact matching strings, we will not perform a merge

of the similarity values of the different individual fields to determine an overall similarity value, as is usually done in record linkage (see the Literature review chapter; Chapter 2).

To clarify why we want to make use of finding exact matching strings, the following example:

Assume there exist two places with the names "Epe" and "Ede" and the input of the user consists of the name of a town, a street name, and/or a zip code. If the user types "Ene" in the town field, a town that does not exist, it is justifiable that "Epe" and "Ede" are equally likely (all measures we tested in Chapter 4 would value the strings equally likely). However, if the user types "Epe" instead of "Ene", the probability that "Ede" is targeted drops significantly, although the absolute similarity value stays the same (again, this is the case for all measures we tested in the thesis). It is of course still possible that the user actually targets "Ede", but it is intuitively right to check for the best matching location that can be found in the place "Epe". Besides, the user will not blame the algorithm for returning a location in "Epe" if that is what is typed. If no matching location can be found in "Epe", the algorithm should exploit the information that can be extracted from the given street name and/or zip code, for which a similar reasoning holds.

Another important observation used in building the flowcharts, is that the information that can be extracted from an input string differs. If an error is made in a town string, it is often possible to find the correct match using the trigrams Dice's measure. This makes an erroneous town more valuable than for example an erroneous zip code, which is, as explained before, hard to match with the correct zip code. On the other hand, if a zip code in the Netherlands is correctly given as input, it immediately points to the specific targeted location, while a correctly entered town leads only to a subset of possible locations. Consequently, we highly value an exactly matching zip code, but we do make little use of a zip code that is not matching. An erroneous town can on the contrary still be of high value.

It is difficult to determine how well the matching algorithm performs, and to proof why this is the best way to set-up the matching algorithm. Important to note is that the flowcharts are developed over time. We continuously adapted the flowcharts, when the output given in a part of the flowcharts was unsatisfactory. An indication of the quality of the algorithm will be given in the section with the results (Section 5.4).

We will first give general comments about the properties of (all) the flowcharts, after which we focus on the different flowcharts itself.

5.1.1 General comments about flowcharts

- In the flowcharts we make a difference between a "match" and "location found". A "match" means that an input string matches with a string from the database of the same type. The "location is found", if a location in the database matches with the input location on all used fields. As said before, a location can be on all possible levels of detail.

- If the "location is found", the algorithm returns only the location. As a consequence, if for example only a town is given as input and that town matches exact with a town in the database, we only return the town (i.e, we do not have to return a specific street and/or zip code in the town). Otherwise, always three locations are returned to the user. If the same location should be given more than once, it is presented to the user just one time. This may for example happen when "best town having best street" is the same location as "best street in best town".

- An exactly matching town, street, or zip code given as input is also defined as the "best" input string of the type. A "best" town, street, or zip code does not have to be exactly matching. The "best" town or street is the town or street that is most similar according to the trigrams Dice's similarity measure. The "best" zip code is the best zip code according to the developed zip code algorithm.

- When a town and/or street string are given as input, a preprocessing procedure will select locations that are candidates to be returned to the user. In the preprocessing step, the locations that differ so much from the input location that they could never be the targeted location, are deleted from the set of locations. The goal is to reduce the computation time and the procedure is applied before the trigrams Dice's measure and the zip code matching algorithm will determine the best locations. For the details about the preprocessing procedure, see Section 5.2. For now, it is enough to remember that only a subset of the locations remains, from which the targeted location will be determined.

- If no location is left after the preprocessing, sometimes "No location found" is returned to the user. As a consequence, the number of locations returned to the user varies from zero to three. Note that never more than the number of locations left after preprocessing can be returned. The last presented flowchart, in Subsection 5.1.7, represents what the algorithm needs to give as output, when no locations are left after the preprocessing.

- If the "best" zip code needs to be returned, only those are considered that are corresponding to locations left as candidates after the preprocessing procedure. An exact match on zip code will however be searched in the complete list of zip codes, regardless the locations left after preprocessing. The preprocessing procedure is only applied on town and street strings. As a

consequence, if only the zip code field is used, the "best" zip code will be returned from all zip codes in the country.

- If the trigrams Dice's similarity measure or the zip code matching algorithm defines several strings as equally similar, ties are broken based on the frequency of occurring in the database. The idea is that if a big city and small village are rated as equally similar, it is more likely that the user targets the big city. The number of locations a town has in the database, the larger the town likely is. The same reasoning holds for streets and for zip codes (if in the country the zip code points to more than one location). How often a string of a certain type occurs in the database should be calculated beforehand and be stored as a number together with the string. In the C++-implementation, we stored the strings corresponding to a location in an object together with the numbers indicating the frequency of occurring.

- In continuation of the previous point: if for example the algorithm searches for example the "best street in best town", and more than one town is rated as "best", than the streets considered are the streets in all "best" towns. If there is also a tie for best street, the finest level of detail will be used first as a tie-breaker with their frequency of occurrences. This holds for all combinations of street, town and zip code.

- In every output field in the flowcharts, the location presented first is the location that should be returned to the user. The others should be given as suggestions.

The legend for the flowcharts is given in Figure 2.

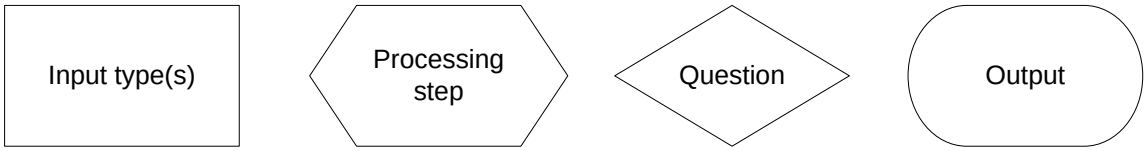

Figure 2: Legend of the output flowcharts

To summarize, in the next subsections we depicted the flowcharts representing the output of the matching algorithm. Depending on from which types the input strings are, and which of the input strings match exactly with a string in the database, the algorithm selects the locations from the database that it will return to the user. The number of locations returned varies from zero to three. For every possible combination of input strings from the string types town, street and zip code, we created a different flowchart:

- Town, or street, or zip code (merged in one figure): Subsection 5.1.2, Figure 3.

- Town and street: Subsection 5.1.3, Figure 4.

- Town and zip code: Subsection 5.1.4, Figure 5.
- Street and zip code: Subsection 5.1.5, Figure 6.
- Town, street and zip code: Subsection 5.1.6, Figure 7.

5.1.2 Town, street, or zip code

The flowcharts for input locations consisting of a single string, shown in Figure 3, are rather simple. If the string matches with a string in the database, the location should be given as output (note that a "location is found" if the string matches). Otherwise, the top three locations should be given. For zip codes, the zip code matching algorithm should produce the best three zip codes, or return "No location found" (see details of the algorithm in Subsection 4.2.3). The three best towns and streets should be determined according to the trigrams Dice's measure.

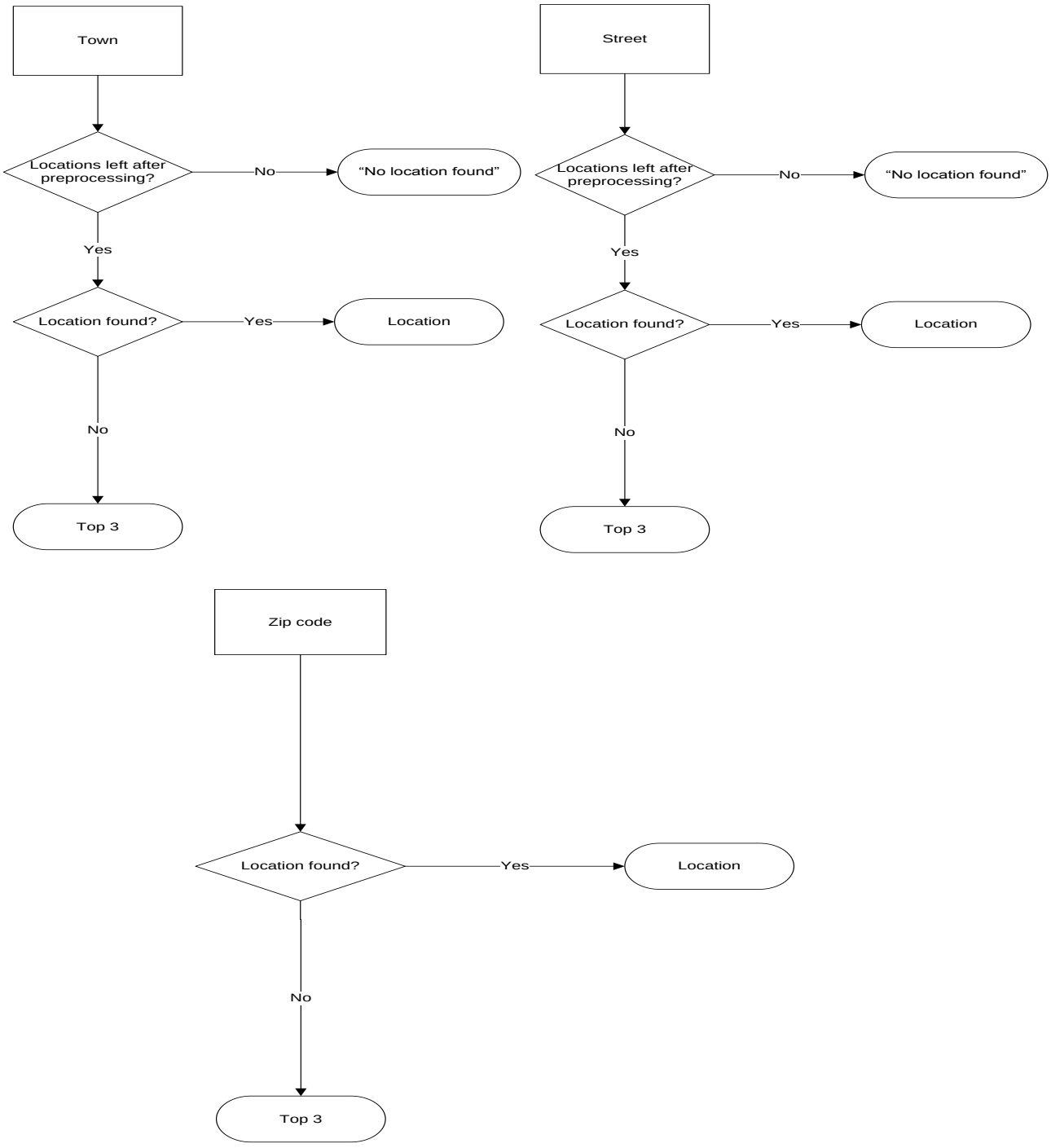

Figure 3: Single string input

5.1.3 Town and street

The flowchart for the town and street combination is more complex, see Figure 4. Again, if the "location is found", the location should be given. Otherwise, the algorithm first checks if town and/or street match exactly with strings in the database. Depending on whether or not town and street match, the algorithm decides which three locations should be returned to the user, using the trigrams Dice's measure. We go into detail about this flowchart, to get a good feeling on the algorithm.

If, for example, the town matches with a town string in the database, but the street string does not match with a street string in the database, we want to have the following locations given as output:

- The best two streets in town
- The best town having the best street

The reason for giving two streets in the town is that we want to exploit the fact that we have a match on town, which makes it very likely the user targets a location in the town. However, if the town is accidentally not the one targeted, we want to use the information we have from the street string to give a reasonable suggestion (i.e., by giving the best town having the best street).

Another possibility is that both street and town match exactly with a string in the database from the same type, but that the strings do not point to the same location. In that case we want to exploit both the match on town and the match on street. We give slightly more value to the street match, as it might be that the user links the wrong town to the street. This leads to the following output:

- The best two towns having the street
- The best street in the town, or, if only one town with the street: the best two streets in the town

When both town and street cannot be matched with a string of the same type, the algorithm returns "best street in best town", "best town having the best street", but also the best location, which is not given as output yet, based on an average similarity value. This average similarity value is an average of the trigrams Dice's score for the town string and the trigrams Dice's score for the street string of the location. The average similarity value is used to make sure we do not miss the targeted location when it has a high similarity value both on town and street, but is not valued as most similar in at least one of the two categories.

For what to do when no locations are left after the preprocessing step, we refer to Subsection 5.1.7. It is not true that "No location found" needs to be returned in all cases, because it may be that one of the two strings provides valuable information about the targeted location. If the other string does not provide valuable information, it could still lead to an empty set of possible targeted locations (see for details the preprocessing section; Section 5.2).

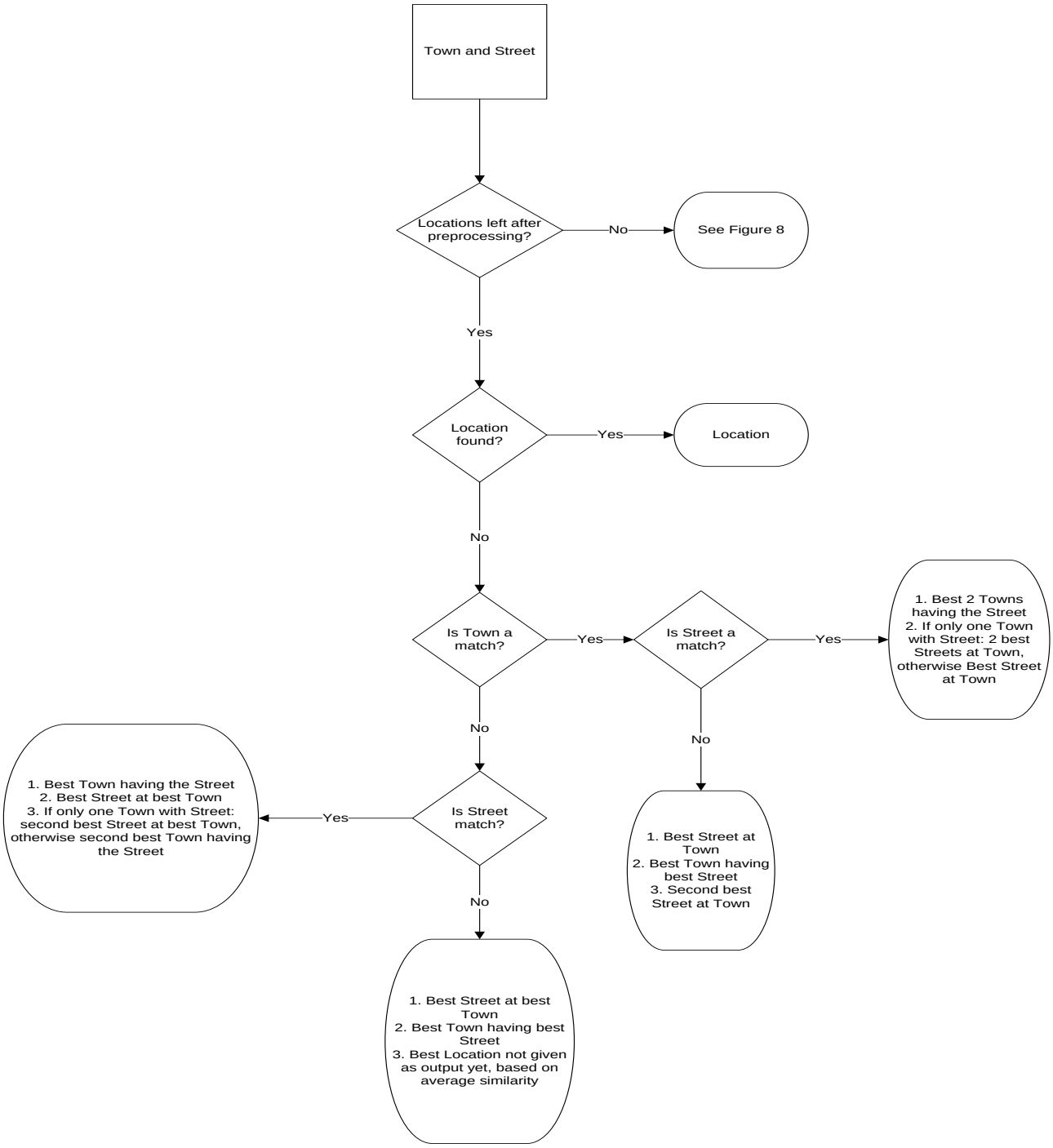

Figure 4: Town and street

5.1.4 Town and zip code

The next flowchart, Figure 5, is the representation of the output of the algorithm when a town and zip code string are given as input.

A first important note is that it might be necessary to slightly adjust the algorithm, depending on the level of detail a zip code gives in the considered country. For example, in the Netherlands every street consists of one or more zip codes. In Belgium, several towns may share the same zip code. Thus, the Belgium zip code is less specific in determining the targeted location than a zip code in the Netherlands. The flowchart is presented with the assumption that the zip code is the finest level of detail. The necessary adjustments if this is not the case, are however often obvious.

If in The Netherlands both the town and zip code are matches, but no location is found, the output should be as follows:

- Zip code
- 2 best zip codes in town

In the same situation in Belgium the output should be as follows, as intuitively clear:

- Town
- 2 best towns with zip code

No matter the level of detail a zip code represents, we make little use of a non-matching zip code, as it will be hard to match with the targeted zip code. For example, if the town matches, but the zip code cannot be matched, the output is:

- 3 best zip codes in town (or, just the town in case a town has only one zip code)

The only case where we want to find the "best" zip code if it does not match, is when also the best town is not matching. If the best zip code(s) point to more than one town, the best town should be selected out of the set of towns corresponding to the zip code(s). The algorithm returns the following locations to the user:

- 2 best zip codes at best town (or, if a town has only one zip code: 2 best towns)
- Best town with best zip code

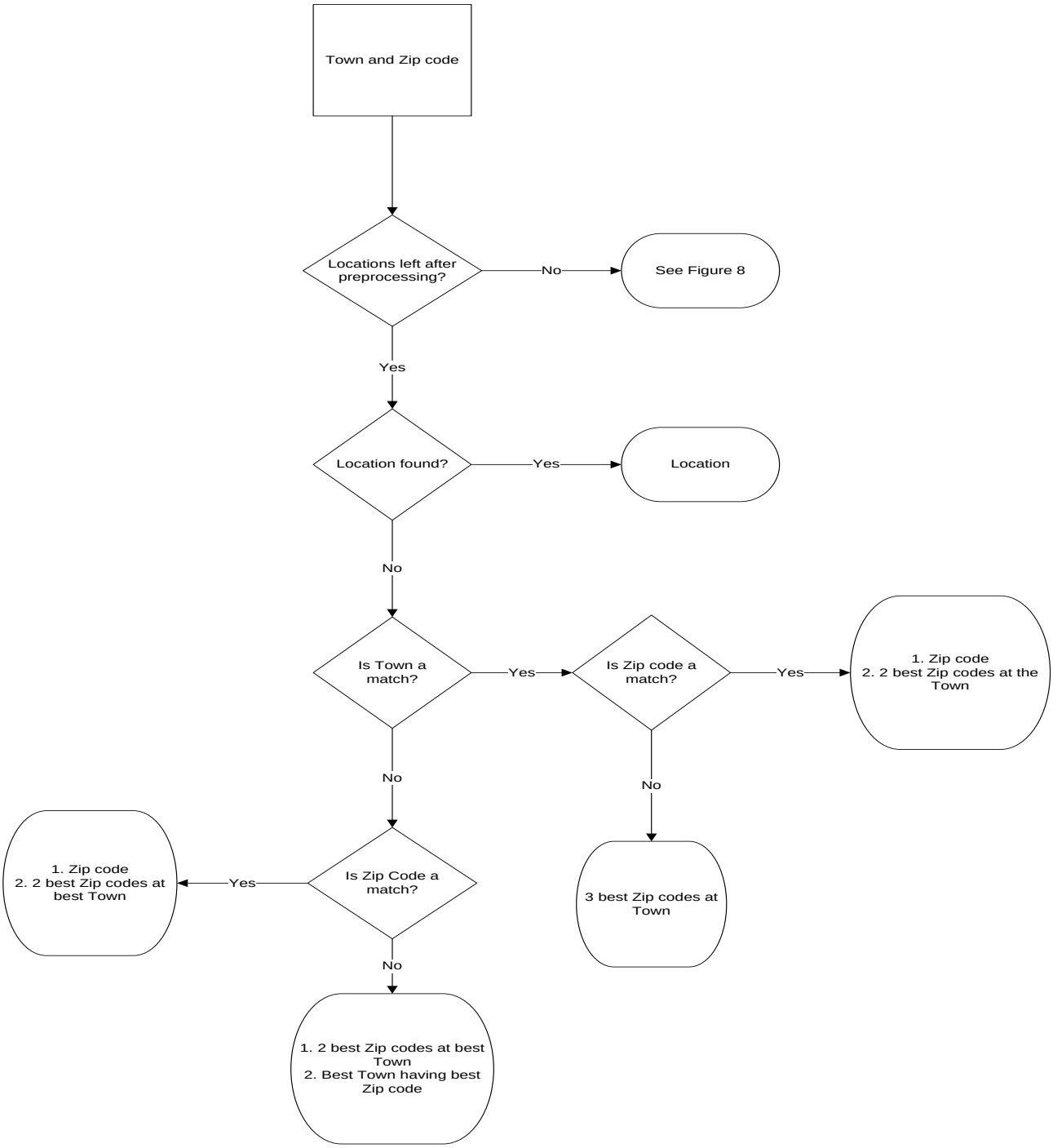

Figure 5: Town and zip code

5.1.5 Street and zip code

The flowchart when a street and zip code string are given as input is equivalent to the flowchart for town and zip code. It is build on the same ideas. See Figure 6.

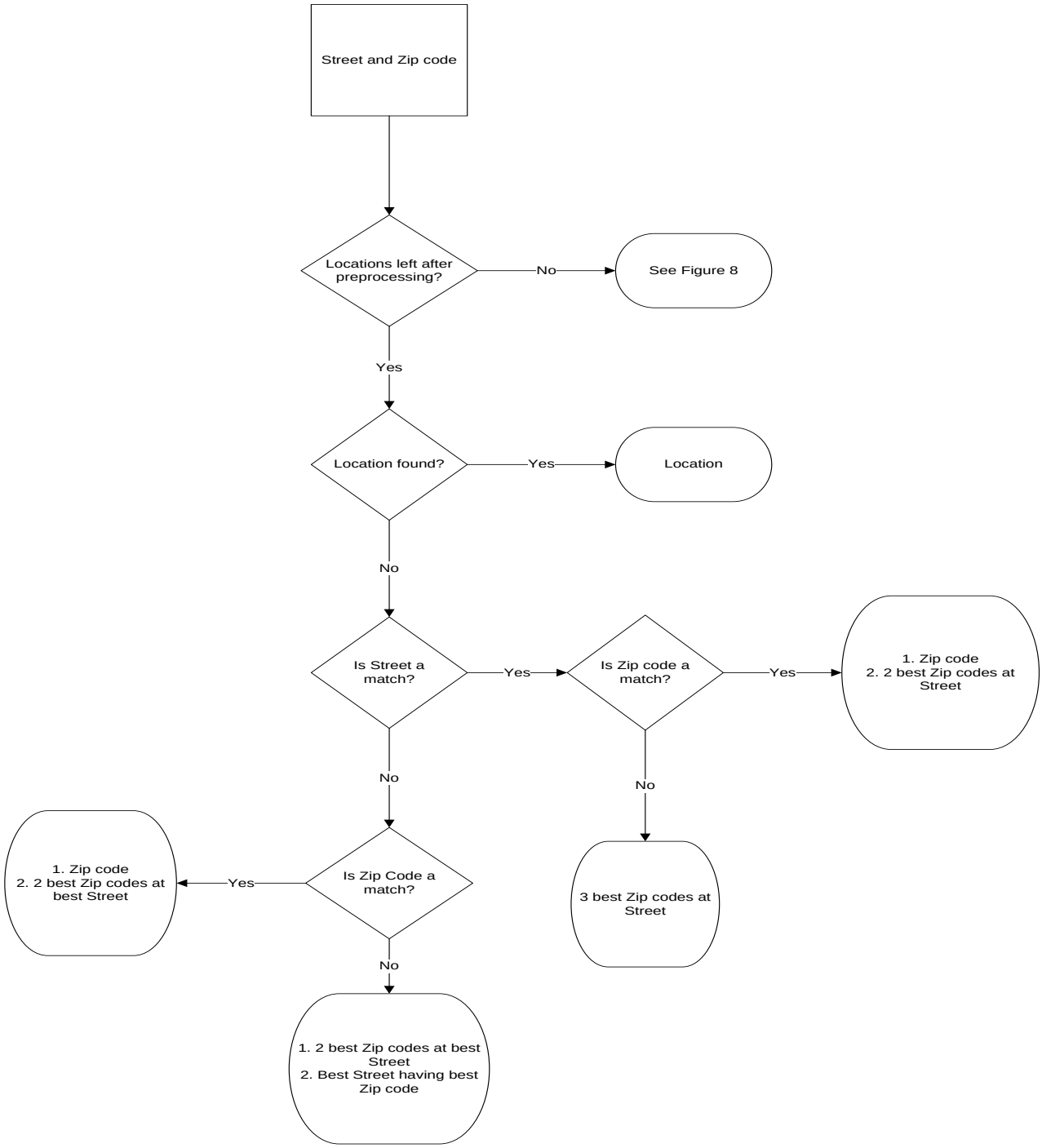

Figure 6: Street and zip code

5.1.6 Town, street and zip code

The flowchart representing the algorithm when all three input fields are used, is given in Figure 7. We go through the parts that need some explanation. The main ideas are still that we want to exploit exact matching strings and give little value to a non-matching zip code.

If two out of three strings point to the same location, it is assumed the location is the targeted location. It may be that, if all three strings are matching, the strings point to two different locations (e.g., town and street point to a different location than town and zip code). Both locations are returned to the user in that case.

Note that the output given when the zip code is not a match, is very similar to the output when only town and street are given as input. In all cases the zip code is not used if it does not match, because we expect to be able to find the targeted location using the town and street string.

When a matching zip code points to more than one street and or town, the location that will be given is the one with the highest average similarity value based on the street and town string.

In case a matching zip code points to more than one street and/or town, the location that is returned is the one out of the set of matching zip codes, with the highest average similarity value on its street and town string.

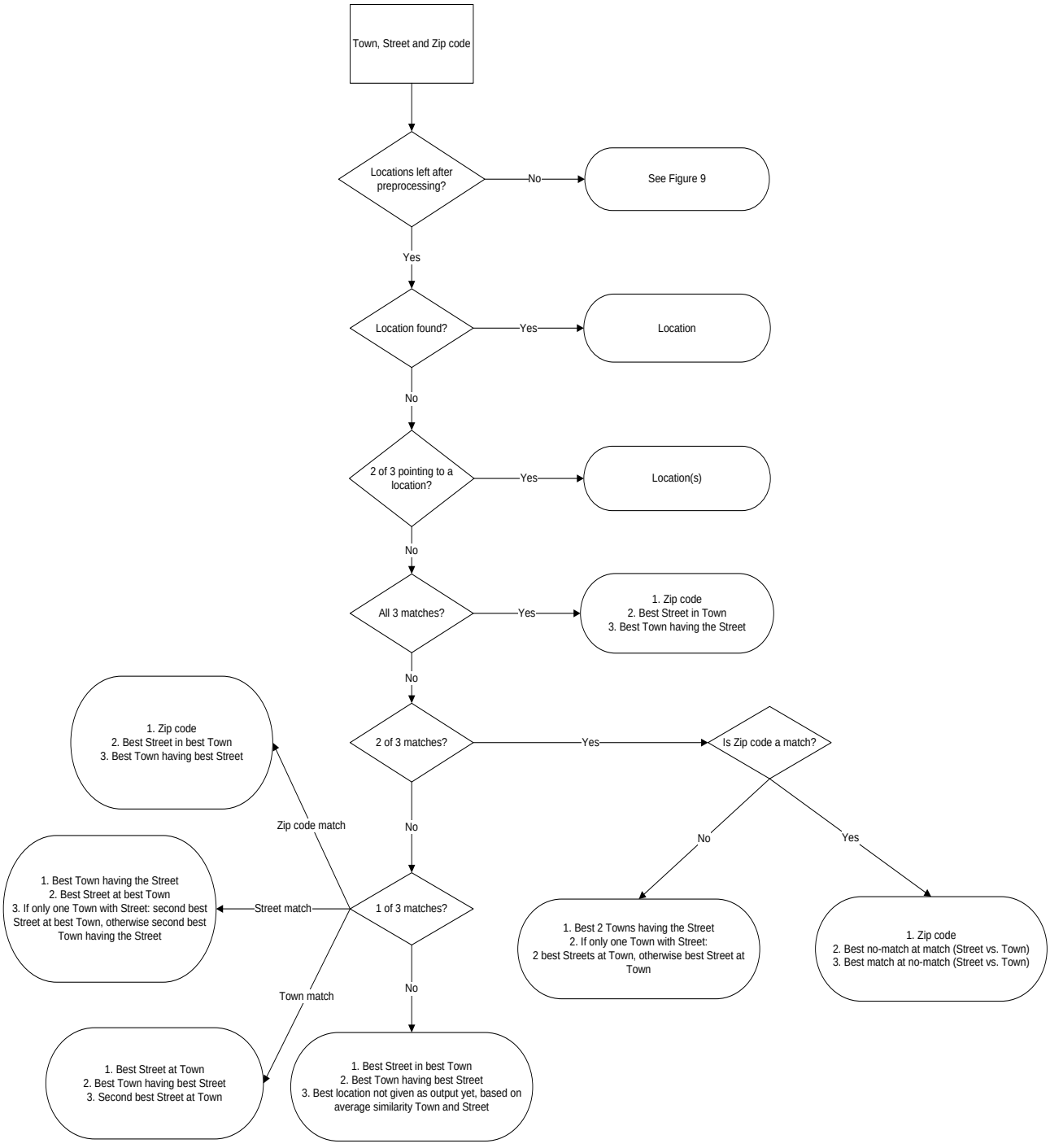

Figure 7: Town, street and zip code

5.1.7 No locations left after indexing

The flowcharts representing the algorithm when no locations are left after the preprocessing procedure, are presented in this subsection. We repeat that the explanation of the preprocessing procedure can be found in Section 5.2.

In Figure 8, the flowcharts are given for all combinations when two out of three input fields are used. In case the zip code is used together with the town field, or the street field, we only consider the zip code. This is because the preprocessing is performed only on the town or street, which means that none of the locations in the database has two trigrams in common with the town or street input string (hence, there is probably rubbish in the town or street field).

If the street and town field are used (but no zip code), the preprocessing procedure takes the intersection of the candidate sets resulting from the separate preprocessing of towns and streets. If no locations are left after taking the intersection, it depends on why this intersection set is empty. If one of the two sets is empty, we focus on the other set (e.g., the algorithm gives "3 best streets in town" if the street set is empty). If both sets are empty: "No location found" is returned. Otherwise, if both sets consist of locations, but the intersection is empty: we try to find the best locations from both sets.

In figure 9, the flowchart is given for the case in which all three input fields are used. Always the (best or exact matching) zip code should be returned to the user. The other locations given as output, are the same as in the case when only street and town are given as input and no locations remain after preprocessing. If both the candidate sets are empty and the zip code is not an exact match, the second and third best zip code are given.

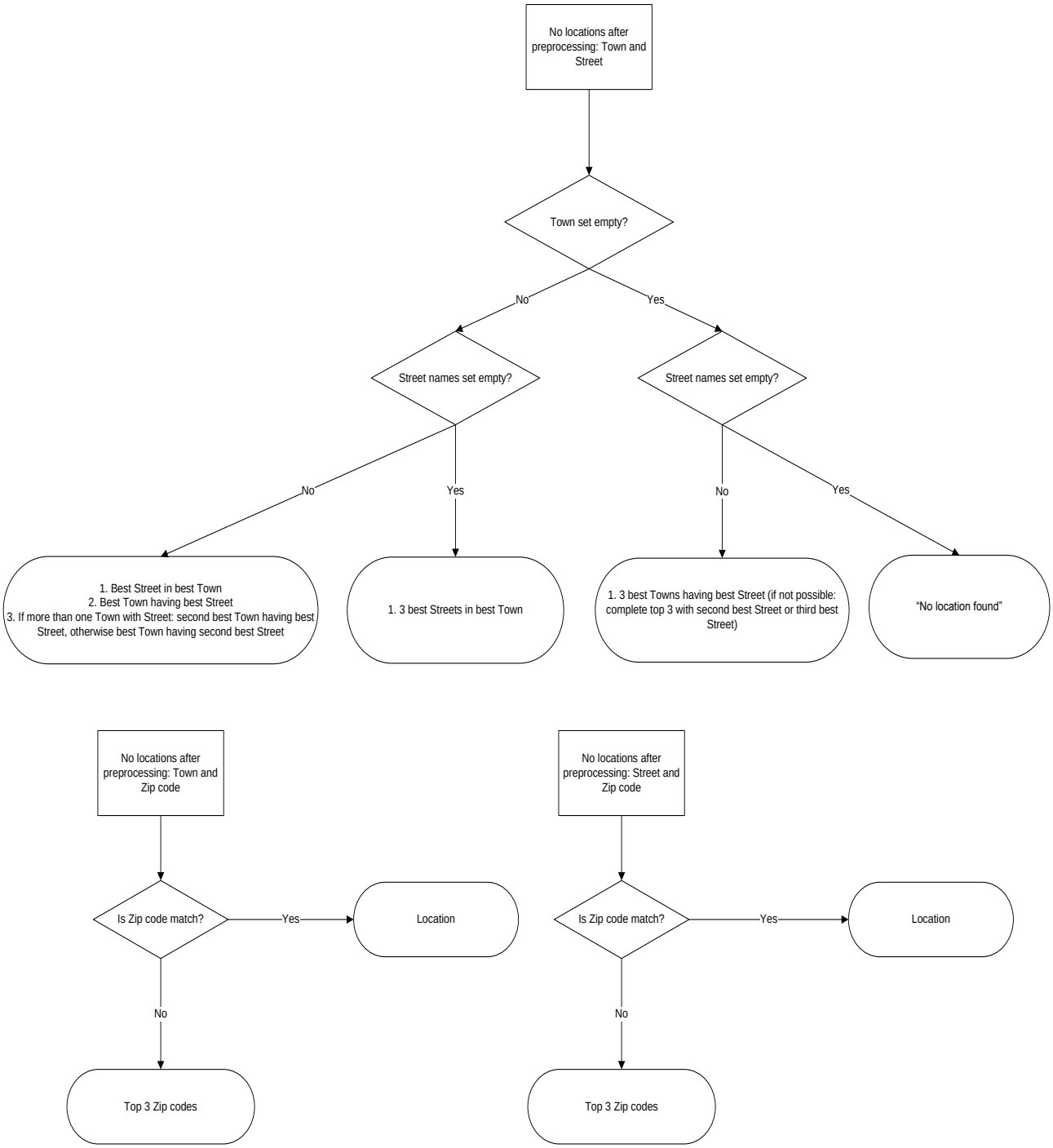

Figure 8: No locations left after preprocessing I

Figure 9: No locations left after preprocessing II

5.2 Preprocessing: database indexing with padded trigrams

In this section the preprocessing procedure is discussed. First, we explain why a preprocessing step is needed and give a general description of what it will do. After that, we go through the preprocessing algorithm in more detail and present some results.

For the matching algorithm to be useful in practice, it should return the location(s) to the user in a very short period of time. How long this period may be is not strictly defined, but ideally the user must not have the feeling he should wait. We therefore assume that it is unacceptable if the algorithm takes more than one second before the locations are presented to the user. However, the faster the calculations are done, the better.

If the trigrams Dice's similarity values need to be calculated between an input string and many strings in the database, it will soon take too much time. For example, when calculating the similarity value between an input string of regular size and all street names in the Netherlands, the computation time will already be at least a couple of seconds.

A common way to speed up the search, see for example [19], is to filter out strings that differ so much from the input strings, that the corresponding locations can not be targeted by the user. A string is considered not to be a candidate to be targeted, if it does not share at least a couple of padded q-grams with the input string (note the similarity with the trigrams Dice's measure, which calculates a similarity value based on the shared number of padded 3-grams). The padded q-grams of which the strings in the database consist are defined once, before any requests are given by a user. The storage of the padded q-grams and the link with the strings having the padded q-gram, is called *indexing*. When referring to q-grams in the rest of this chapter, we mean case-insensitive padded q-grams.

The idea of this filtering technique, is that for almost all inconsistencies that could occur between the input string and the targeted string in the database, the strings will share a number of q-grams. On the other hand, most of the dissimilar strings will share at most a few q-grams with the input string. By setting a threshold value T as the minimum number of q-grams in common needed to label the string as a candidate match, most dissimilar strings will be filtered out, while the targeted string will be kept as one of the candidates. The trigrams Dice's measure will be applied only on the set of candidate strings. Because the set of q-grams for all strings in the database needs to be created once beforehand, it does not increase the running time of the algorithm. The question if two strings share at least T q-grams can be answered very quickly, which is explained below (see Subsection 5.2.1).

We have chosen to implement padded trigrams with threshold value T equal to two. The reason for this is that even when a very short string of length three is manipulated due to an insertion, deletion or substitution, the number of common trigrams is still at least two. We consider this as the most severe error that can be made, while the input string can still be connected to the targeted string in the database. An example to make this more clear:

Assume there exists a place "Abc". Its 5 trigrams are:
(__a), (_ab), (abc), (bc_) and (c__). If one of the following errors occurs, there will still be two or more shared trigrams:

- Insertion: "Azbc" has the trigrams (__a), (bc_) and (c__) in common with "Abc".
- Deletion: "Ab" has the trigrams (__a) and (_ab) in common with "Abc".
- Substitution: "Azc" has the trigrams (__a) and (c__) in common with "Abc".

Other errors that are made on a regular basis will also lead to at least two shared padded trigrams:

- Due to the fixed number of trigrams influenced by typographical errors (insertion removes two, deletion and substitution three, and transpositions four common trigrams), longer strings would all have at least two trigrams in common when facing one of these error types.
- The number of common trigrams will not drop below two, if a multiple word string is swapped, or if a substring is added.
- It is very likely that at least two trigrams are shared if words are omitted or abbreviated, as it is obvious omissions and abbreviations will mostly occur in long strings.

Because of these reasons and the assumption that larger errors do not necessarily have to be corrected by the algorithm (consider for example the transposition "Bac" given as input for "Abc"), we used preprocessing with trigrams and the threshold $T = 2$.

We did not consider having a variable threshold value, or a threshold value higher than two, because it turned out that the reduction in computation time was very large, while having a higher threshold would mean a greater risk of deleting the targeted string from the candidate set.

We have chosen not to use a preprocessing step for zip codes. The zip code algorithm should be fast enough to do the calculations within a second, if efficiently implemented. See for implementation details Section 5.3 (also implementation details about the preprocessing method and how to

calculate trigrams Dice's efficiently are given).

If the user gives a town or street string as input, but not both, the set of candidate locations is the set after applying the preprocessing step. If both fields are filled, the final candidate set is an intersection of both candidate sets (i.e., the final set of candidate locations consists only of the locations that are in the candidate sets resulting from both preprocessing towns and preprocessing street names). If an input string is an exact match with a string from the same type in the database, but **all** locations corresponding to that string are not in the intersection of both sets, the locations are added to the final candidate set afterwards. This is to make sure that when a street is erroneously assigned to the wrong town, the location can still be found. For example, we do not want to skip "Vonderweg" in "Eindhoven" from the candidate set, if the user believes the "Vonderweg" is located in the town close by: "Boxtel". The other reason is that we do also not want to remove all locations from the town if the town matches exactly. In continuation of the example: if no street in "Boxtel" has at least two trigrams in common with "Vonderweg", we add all locations in Boxtel to the candidate set.

In the next subsection, the method is explained that quickly selects the strings from the database that have at least two trigrams in common with the input string.

5.2.1 MergeSkip algorithm

Assume we want to select strings from the database that have at least 2 trigrams in common with the input string and the strings are of the type town (although the method could be used for all string types). In this subsection it is described how to find the "candidate" strings efficiently.

First, a map $M[trigram, idList]$ is created. The keys are the trigrams from the town strings in the database. The corresponding value is a list of string ids, having the key trigram at least once. Ids of strings having the trigram twice should also be put into a different list (e.g, if the trigram "abc" occurs twice in a string, the id of the string should be put as value corresponding to the key "abc2"). The same holds for trigrams that occur more than two times in a string (e.g., if a trigram is three times in a string, the string is put in three different lists; also the key "abc3" is created in that case). All lists of ids are sorted in ascending order. M is created once, before any input strings are given.

When the user fills in the town input field, we need to find in M the ids that appear at least $T = 2$ times on the lists corresponding to the trigrams of the input string. Simply scanning the lists and keeping track of all ids with counters may take too long and as a consequence the preprocessing

step loses its purpose of reducing running time. To find the strings efficiently we implemented the MergeSkip algorithm as presented in [19].

The idea of the MergeSkip algorithm is to skip records that can not appear at least two times on the lists. Before we go to a formal description of the algorithm, a simple example to illustrate the idea:

Assume that the input string consists of three trigrams and the three lists corresponding to the trigrams consist of the following ids:

- $\begin{bmatrix} 1 & 2 & 3 & 4 & \dots & 19 & \dots & 100 \end{bmatrix}$
- $\begin{bmatrix} 20 & 21 & 22 & 23 & \dots & 100 \end{bmatrix}$
- $\begin{bmatrix} 30 & 31 & 32 & 33 & \dots & 200 \end{bmatrix}$

Consider the smallest id on every list: 1, 20 and 30. Because the second smallest number of those three numbers is 20, all ids on the lists that are smaller than 20 can never occur two or more times on the lists. Hence, we can focus only on the ids 20 and larger and neglect ids 1 to 19 on list one. Consequently, the strings with the ids 1 to 19 do not have two trigrams in common with the input string and should not be selected as possible targeted string.

A formal description of the MergeSkip algorithm is given in algorithm 3.

```
Input: Input string and the map M, from trigram to list of string ids in the database
       having the trigram
Output: The set C of strings having at least two trigrams in common with the input string
Select from M the in ascending order sorted lists of ids, corresponding to the trigrams of the
input string (select only the lists from trigrams that have any appearances in the database);
Put the first element of each list into a multiset H;
while H is not empty do
    if the smallest element s in H is T = 2 or more times in H then
        Add the string corresponding to the id s to a set of candidates, C;
        Add the next element of each list on which s is present, for which the end of the list is
        not reached, to H;
        Delete all elements equal to s from H;
    else
        Find the next element of the list on which s is present, that is larger than or equal to
        the second smallest element in H;
        if such an element exists then
            Add the element to H and remove the smaller elements (if any) from all selected
            lists (i.e., lists corresponding to the trigrams of the input string);
        end
        Delete s from H;
    end
end
```

Algorithm 3: Formal description of the MergeSkip algorithm

Note that if s appears only once at the list, the situation is the same as in the introductory example, in which at the start s is equal to one. The string corresponding to s does not have to be added to the set of candidates and we can neglect all ids smaller than the second smallest id in H (in the example: neglect every id smaller than 20 and remove id 1 from H). If, for example, the trigram "abc" occurs twice in the input string, the lists corresponding to the keys "abc" and "abc2" are selected, if they exist. By doing this it becomes possible to distinguish between strings that have the trigram only once and strings that have the trigram at least twice. If we would select the same list of ids corresponding to "abc" twice, the strings that have "abc" only once would appear twice on the lists and mistakenly be selected as candidate.

The trigrams Dice's similarity measure only needs to rank the input string to towns in the set C, such that the best match(es) can be returned to the user.

The running time of the MergeSkip algorithm is almost negligible. In [19], using padded trigrams and $T = 3$, the candidates were selected with the MergeSkip algorithm from a database with 1,200,000 actor names in around 10 milliseconds. Examples to compare: in The Netherlands, there are only 2,500 towns and 200,000 streets and in Germany, there are around 2,000 towns and 1,100,000 streets. Although street names could on average be longer than personal names, the average running time will certainly not be much more than 10 milliseconds, as we have chosen to set $T = 2$. The implementation of the MergeSkip algorithm in this thesis is not optimized with respect to speed, because the sorted lists do not consist of ids, but of objects containing information about the location (although the sorting is performed on the corresponding ids). Nevertheless, even for this implementation the running time of the MergeSkip algorithm for a street input string in The Netherlands is in general within 250 milliseconds.

The percentage of candidate strings left after the MergeSkip algorithm is applied, is given later in the section. First, we give a description of the adjustment of the preprocessing method, to be able to apply it usefully on street names.

5.2.2 Preprocessing street names by removing common suffixes and prefixes

Preprocessing with trigrams and selecting the right candidate string with the MergeSkip algorithm works fine for names corresponding to strings of the input type town. However, it may not be very useful for street strings. Consider for example the street "Main Street". The trigrams corresponding to the substring "street" will lead to a candidate set of strings consisting of at least all street names having the substring "street". In The Netherlands the synonym for "street" is "straat", and from all street names in The Netherlands, 32.9% ends with "straat". The substring "straat" does not give much information about the street name, but the preprocessing procedure takes more time, leaving much more irrelevant candidate strings for the trigrams Dice's similarity measure to judge, which on his turn will also consume more time.

A good way to overcome this, would be to remove commonly used prefixes and suffixes such as "street" or "road" and search for two common trigrams in the rest of the string. Described here is a method to find those common pre- and suffixes. How this should be done depends on the country considered, but the method used in this thesis seems a promising framework.

The idea of the method to find common pre- and suffixes, is to find substrings of street names that occur often at the start and/or at the end of a street name. This can be done by counting all substrings longer than length x and check which substring occurs most often. If the substring is part of another substring that occurs "almost" as much, the most frequently encountered substring

is often a part of the longer substring. If no such a longer substring exists and the substring is a very large percentage of its occurrences a pre- or suffix, it is likely that the substring is a common pre- or suffix such as street or road, which we would like to remove before indexing. In algorithm 4, the procedure is presented formally.

Input: List of all street names in the country
Output: A list of common pre- and suffixes of street names
Let S be the list of street names in the country;
Set the maximum number n of common pre- or suffixes to remove;
Set the parameter $\rho > 1$ (ρ defines the threshold of the drop in frequency of occurrence of a substring);
Set the parameter $x > 0$ (x is the minimum length of a common substring minus one);
Let $s0$ be the most encountered substring in S and $f0$ its frequency of occurring;
Let $s1$ be the most encountered substring in S from which $s0$ is a part, with frequency $f1$;
while *number of removed common pre- or suffixes* $< n$ **do**
    if $\frac{f0}{f1} > \rho$ **then**
        Count and store the number of times $s0$ is a prefix and the number of times $s0$ is a suffix;
        if $s0$ *is more often a prefix than a suffix* **then**
            Remove all strings from S starting with $s0$;
        else
            Remove all strings from S ending with $s0$;
        end
        $n = n + 1$;
        Let $s0$ be the most encountered substring in S and $f0$ its frequency of occurring;
        Let $s1$ be the most encountered substring in S from which $s0$ is a part, with frequency $f1$;
    else
        Set $s0 = s1$ and $f0 = f1$;
        Let $s1$ be the most encountered substring in S from which $s0$ is a part, with frequency $f1$;
    end
end

Algorithm 4: Algorithm to find common pre- and suffixes

The reason to return to set $s0 = s1$ and $f0 = f1$, when $\frac{f0}{f1} <= \rho$, is that in that case $s0$ is often a substring of $s1$. We then want to check if $s1$ is often a substring of another string and repeat the

procedure.

At the end, the substrings that are often used as prefix or suffix can be detected from the stored frequency values. Note that the goal is to not left too many irrelevant candidates, for which the trigrams Dice's similarity measure should calculate the degree of similarity with the input string. As a consequence, we are free in choosing which substrings to remove. The algorithm requires a few parameters to be set manually, but when applying the algorithm on street names in a specific country, the programmer can easily use several settings, probably leading to slightly different lists of common pre- and suffixes, and decide at the end which pre- and suffixes will be removed. Running the algorithm several times with manual tuning is possible because the common pre- and suffixes need to be defined only once (before the algorithm will run).

A complication may be that a frequent letter combination may appear often in the middle of the string and thus end up high in the frequency list. This can be easily solved, by choosing to neglect the substring in the frequency list.

For street names in The Netherlands we knew beforehand that the shortest commonly used pre- or suffix has length 3, so x was set to 2. ρ was set to 1.5. This means that the number of times the substring occurs as pre- and suffix is stored, if the frequency of occurrence of the substring dropped more than 50% in comparison with the frequency of occurrence of another substring, of which the substring is a part. The maximum number of pre- and suffixes generated was set to 9. As said before, several settings of the parameters can be tried is desirable. The results turned out to be very promising, as the pre- and suffixes that were expected are found by the algorithm. The results are shown in Table 17.

Substring	% in street names	% as prefix	% as suffix	% in middle of string
“straat”	32.90%	0.07%	99.26%	0.67%
“weg”	14.72%	0.37%	97.40%	2.23%
“laan”	10.65%	1.87%	96.65%	1.48%
“de ”	5.95%	60.40%	0%	39.60%
“plein”	2.21%	2.07%	98.17%	0%
“hof”	3.50%	7.64%	78.07%	14.29%
“dijk”	2.68%	4.79%	80.05%	15.16%
“ste”	6.11%	9.51%	0.05%	90.44%
“ste”	6.11%	0%	0.56%	99.44%

Table 17: Common pre- and suffixes street strings in the Netherlands

The main conclusion drawn from the percentages in Table 17 is that “straat”, “weg”, “laan”, “plein”, “hof” and “dijk” are all common suffixes. These words often occur in the strings (varying from 2.21% for “plein” to 32.90% for “straat”) and are mostly used as suffix. Removing these suffixes before indexation with trigrams will be useful. The substring “de ” is the only common prefix. In total we found seven common pre- and suffixes. The substring “ste” is the majority of its occurrences in the middle of the string. After removing the “ste” prefixes (as “ste” is more often used as prefix than as suffix), the substring is still on top of the list. If more than seven substrings commonly used as pre- or suffix need to be found, the substring “ste” can be neglected in the analysis and the procedure should be applied on the second most frequently occurring substring. Again, there is a lot of freedom in tuning the algorithm. A last note to the table: the reason why the total percentage the string “plein” is used as pre- or suffix exceeds 100%, is that “Plein” is sometimes a street name itself.

In this thesis, when creating the map M of the street trigrams to list of string ids, the commonly used pre- and suffixes are removed in a hierarchical way. For example, for a street string in The Netherlands, we first checked whether the string ends with “straat”. If so, “straat” is removed and the padded trigrams of the rest of the string are added to M. If not, it is checked whether the string ends with “weg”. If so: remove “weg”, find the trigrams, etc. If the size of the string would be less than 2 if the pre- or suffix is removed: no pre- or suffix is removed and the trigrams are created from the original string. When an input location is given by a user, the same procedure of removing the most common pre- or suffix is applied on the input street string.

5.2.3 Results of preprocessing

To get an idea about the percentage of locations the preprocessing method filters out, see Table 18. In the table the results are given for ten, randomly picked from the dataset, addresses in The Netherlands consisting of a town and street string. A location is in this case defined on the detail level of street:

% removed places	% removed streets	% removed locations	# locations left
93.31%	99.43%	99.93%	143
96.29%	96.43%	99.73%	542
95.55%	91.87%	99.55%	920
95.23%	95.60%	99.79%	424
95.39%	94.18%	99.69%	634
94.78%	86.87%	98.76%	2530
94.98%	95.48%	99.64%	728
91.07%	95.67%	99.63%	763
94.08%	98.98%	99.93%	141
95.19%	97.32%	99.93%	138

Table 18: Locations filtered using preprocessing method

As we see in Table 18, the number of strings that have to be compared using a similarity measure is reduced enormously after preprocessing (only 0.07% to 1.24% of the locations are left over). The filtering step can be performed in little time and the probability is very high we keep the targeted location in the candidate set. This makes the preprocessing a valuable method to speed up the search, almost without losing any accuracy.

5.2.4 Complexity analysis of preprocessing

In this subsection, we determine the computational complexity of the preprocessing algorithm. It is interesting to investigate what determines the time the preprocessing algorithm takes in the worst case, as the main goal is to reduce the computation time of the matching algorithm itself. As we will see in Subsection 5.3.2, this will also be the total complexity of determining a best street or town out of a set of streets and towns, as it is possible to determine the trigrams Dice's similarity values already in the preprocessing phase. In the worst case, the preprocessing needs to be performed for both a street and town string, after which the intersection of both sets is taken. We refer here to the MergeSkip algorithm given in subsection 5.2.1. The removal of a common pre-

or suffix will not increase computational complexity.

The first step is to select the list of the trigrams corresponding to the input string in $O(|S| * log(|D_t|))$, with $|S|$ the size of the input string and $|D_t|$ the number of lists corresponding to all padded trigrams in the database. Because the number of possible characters is 27 (space + all character in the alphabet), the first step will be calculated in $O(|S|)$.

In the next steps of the algorithm, the computational complexity is bounded by inspecting all elements of the selected lists. The sum of all elements is denoted by $\sum_{tri \in A} |L_{tri}|$, with A the set of id-lists and L_{tri} the list of string ids corresponding to trigram tri. Consequently, these steps take $O(\sum_{tri \in S} |L_{tri}|)$ time. So, to determine the candidate sets for both streets and towns, we have the following time complexity: $O(|S^s| + |S^t| + \sum_{triS \in S} |L_{triS}| + \sum_{triT \in T} |L_{triT}|)$, with S the set of selected lists with string ids for streets and T the set of selected lists with string ids for towns.

In the worst case the intersection of both candidate sets of locations (for the town and street string) needs to be determined. This means we need to find out which string ids are on both lists. To accomplish this, the two lists are sorted and we loop over the lists. The computational complexity is then bounded by the sorting, which costs

$$O\left(\sum_{triS \in S} |L_{triS}| * log(\sum_{triS \in S} |L_{triS}|) + \sum_{triT \in T} |L_{triT}| * log(\sum_{triT \in T} |L_{triT}|)\right).$$

This makes the total complexity of the preprocessing algorithm:

$$O\left(|S^s| + |S^t| + \sum_{triS \in S} |L_{triS}| * log(\sum_{triS \in S} |L_{triS}|) + \sum_{triT \in T} |L_{triT}| * log(\sum_{triT \in T} |L_{triT}|)\right).$$

Thus, the time complexity of the preprocessing is linear in the size of the input string and sensitive for the size of the lists of strings corresponding to the trigrams. Consequently, the computational complexity depends mostly on the size of the lists, as the input strings will in general be of limited size. Note that this complexity analysis shows that it is useful to remove common pre- and suffixes in streets, because the size of the lists of string ids will be very large for the trigrams in a common pre- or suffix.

5.3 Details of efficient implementation

Most of the data structures used in the algorithm are quite obvious. We use all sorts of maps that link together the towns, streets and zip codes (examples are a map from town to street and a

map from street to all towns having the street). Also an array of all towns, an array of all streets and an array of all zip codes are used.

The architecture of the program is already visualized by the flowcharts: an input location is presented to the algorithm and depending on which input fields are used, the corresponding flowchart represents the continuation of the algorithm. In most of the flowcharts, first the preprocessing method is applied. In the leaves of the flowcharts, the trigrams Dice's measure and zip code algorithm can be used to determine the final output.

In this section, the most remarkable data structures we used are represented. Next to this, we explain how we implemented the trigrams Dice's measure and the zip code algorithm, to obtain an efficient implementation.

The algorithm is implemented in C++ (Microsoft Visual Studio 2010 Professional) combined with Qt 4.8.2, on a 32-bit operating system. From the Qt application-framework we mainly used functions developed to apply on strings. The experiments are ran on an Intel Core 2 Duo processor with 4.00 GB RAM. All operations of the algorithm are performed in the memory of the computer: the database of locations is read from a text file and the output of the algorithm is written to a text file, after which the presented suggestions are checked manually.

5.3.1 Data structures

In the preprocessing procedure on street strings, we use a list with for every street in the country an object consisting of: the street name, the corresponding town and a list of corresponding zip codes. If both a town and a street string are given as input, the intersection of the candidate set of town strings and the candidate set of street strings can be applied on this set. The resulting locations are then used in the rest of the flowchart.

Every town, street, and zip code string, should in any of the maps or lists be accompanied with a number, that is used as tie-breaker according to the frequency of occurring of the string in the database (e.g., by storing as pair using the data-structures (string, integer)).

The data structures used in the MergeSkip algorithm (Subsection 5.2.1) are trivial. We use a multiset to store the currently investigated ids. The multiset sorts the ids in ascending order by default, which makes it easy to find the smallest id in the set and how often it occurs. Ids that occur on the lists of ids often enough, are put into a "candidate list". The link from id back to the string is simply made by the use of a map.

5.3.2 Efficient implementation of trigrams Dice's and zip code matching algorithm

Efficient use of trigrams Dice's

At the moment a string (town or street) is selected as a candidate string by the MergeSkip algorithm in the preprocessing phase, it is already known how many trigrams the string has in common with the input string. This is because the number of times the string id occurs in the multiset of string ids, represents the number of trigrams in common with the input string. If we calculate for each string in the database the number of padded trigrams it consists of beforehand, we can calculate the trigrams Dice's similarity value already when it is selected as a candidate. This will yield a great reduction in running time, compared with first completing the whole preprocessing procedure and then calculate the number of shared padded trigrams for all candidates. When we need to find the best strings (given the complete set of candidates, or only a subset), we only need to sort the strings on their scores.

Efficient use of zip code algorithm

To efficiently use the zip code algorithm (Subsection 4.2.3), we need to have a quick method to check if an exact match can be found and which strings have the longest common prefix (and the length of this common prefix).

As explained in the subsection where the general comments on the match algorithm are given, Subsection 5.1.1, the algorithm searches for an exact matching zip code, regardless the preprocessing of town and/or street strings. This indicates, that the search for an exact matching zip code is always done on the complete set of zip codes. A fast way to check if a string matches exact, is to create a set (i.e., a sorted list) of all zip codes in the country beforehand. In C++, the set function "find(*input zip*)" returns in logarithmic time with respect to the size of the set, if the input zip code can be matched with a zip code in the database.

Now assume no exact matching zip code can be found, but the algorithm needs to return the best zip code. This may be for all zip codes in the country, or for a subset of zip codes (for example the best zip code belonging to the best street or town). In the latter case, we create a set of the candidate zip codes. Although this has to be done in running time, the subset will likely be rather small and consequently not consume too much time. Regardless the set in which we have to find the best matching zip code(s), the fastest way to find the best zip code is to insert the input zip code into the set (the zip code will be put in the right place in the list of zip codes due to the sorting property of the set). With the Qt-function "Startswith(*substring*)" (a boolean returning true if a string starts with the substring), the size of the common prefix of the zip codes above and below the input zip code can be determined in a short amount of time (note that sorting makes sure at least one of those two zip codes has the longest common prefix with the input zip code). For both

the zip codes above and below the input zip code, this is done by first checking if the complete zip code agrees with the prefix of the input zip code. If so, the longest common prefix is found. Otherwise, check if the zip code minus the last character corresponds with the prefix of the input zip code, etc. By traversing over the list it can be checked which zip codes have the longest common prefix using the same procedure and, if necessary, on which zip codes Damerau-Levenshtein should be performed to determine the best matching zip code(s).

The Damerau-Levenshtein distance (see for the description Chapter 3) is calculated by building the dynamic programming matrix, in which we make use of the fact that only the last two columns of the matrix are needed to determine the values in the next column.

5.4 Test results and comparison of matching algorithm

In this section we compare the results of the matching algorithm developed by us, as described in the previous sections, with the results of the current matching algorithm used by CQM. The current matching algorithm is mainly based on the edit distance for town and street and a similar "longest common prefix"-type of measure for zip codes. For three different datasets, we checked how often the targeted string is returned by our algorithm and by CQM's current algorithm. The three datasets are used before when testing the similarity measures, but we did not have a different dataset to compare the two algorithms with. Nevertheless, the results should give a good indication on the performance. It is not completely true that the training datasets are also used for testing, because the matching algorithm produces a different type of results than the similarity measures when testing (namely, locations versus strings of only one type).

If it is impossible to manually match an input location with a location in the database, the input location is not taken into account in the analysis. Just as our algorithm, the current matching algorithm of CQM gives zero to three locations as suggestion. Therefore, we decided it is fair to compare the number of times the targeted string is given as suggestion, regardless the position of the suggestion on which the targeted location is given. The results are presented in Table 19. Dataset 1 has 1327 locations, dataset 2 1582 and dataset 3 336.

	% correct dataset 1	**% correct dataset 2**	**% correct dataset 3**
Matching algorithm thesis	97.36%	99.18%	99.70%
Matching algorithm CQM	80.60%	84.32%	95.18%

Table 19: % of targeted locations returned by algorithms

As Table 19 shows, the results of the matching algorithm are very promising. For all three datasets, there is a significant improvement in matching performance compared with the matching algorithm CQM currently uses. The percentage of correctly returned locations is for all sets above 97% and the difference with CQM's performance is respectively 16.76%, 14.86%, and 4.52%. T-tests show that the differences are clearly significant at a 0.05 critical alpha level, with p-values of less than 0.0001 for the first two sets and and p-value of 0.0002 for the third set.

6 Practical considerations

In this chapter we go through several different practical considerations. First, we discuss the idea of counting a trigram as matching to a certain degree, if letters are not exactly matching, but can be considered as similar (think of à versus a). Thereafter, we advise on several practical issues and problem extensions CQM faces. Every subject is covered in its own section. Also suggestions are given about the implementation (although the main focus is on presenting the concept, as there are often various ways to make sure the idea is covered by the algorithm).

6.1 Grouping characters

In the multilingual problem setting investigated in this thesis, some locations in the database will have a language specific character. Think for example of the Danish "æ" or the Polish "ł". It is unlikely the user will type this character, unless the user is from a country the character is often used in. Also more often used special characters, such as characters with an accent, will often be replaced in reality by the identical character without accent.

The quality of the matching algorithm definitely decreases when this is not taken into account. For example, the similarity value is underestimated when all trigrams having a "ł", are simply considered non-matching with the same trigrams having a normal letter "l".

There are several possibilities to incorporate this in the algorithm. One suggestion is to count the trigrams that mismatch on one character but would match on the "similar" character, for a half. We will not give a list here on which all "special" characters are presented, because this list can be made arbitrarily long. In practice, it would be a good idea to scan the complete database on non-regular characters. In the indexation step (in which the maps from trigram to list of string ids are created), a trigram with a special character should also be assigned to the list corresponding to the trigram with the regular character. In calculating the similarity value, this trigram (if shared with the input string) should be counted for a half.

6.2 Alternative spellings

A small extension of the problem is the incorporation of alternative spellings of a town. The extension follows directly from practice, as CQM knows for several towns in the database what the alternative spelling is.

In the presented matching algorithm, in which the trigrams Dice's similarity values are calculated already in the preprocessing phase, it is reasonable to introduce the location one time for each possible spelling of the town (i.e., consider the same location as different locations in the list of string ids). When the algorithm wants to output a location that is more than once in the list of string ids, it should be checked if the same location is not already presented to the user. In the implementation this can easily be done, by storing a number pointing to the id of the similar location in the object of information about this location (we assigned a number to every location, to which we here refer as id).

An example to clarify the idea: if the algorithm wants to give as output "the two best towns having the street", and the street is in "Den Haag" (alternative spelling of "Den Haag" is "'s-Gravenhage"), the algorithm needs to recognize that the street points to the same location for both town strings, by having the locations linked together. The location should be presented only once to the user.

6.3 Introduction of the community field

We now go through a bigger extension, namely the possibility for the user to use a community input field. Also this extension follows from reality, because for CQM's locations in the database also the community is given. The use of the community field can be incorporated in the matching algorithm in several degrees. Suggestions are given in this section. We use in all cases the observation that a community is basically a set of towns and hence, the set of locations in a community is the union of locations from the towns in the community. Some of the suggestions can be combined if desired.

Check if town string matches exact with community string

What will often occur in practice is that the string given in the town field is actually not the town, but the community. In one of the databases used for testing in this thesis, this occurred quite a few times. If the town string cannot be matched exactly with a town string in the database, it could be

checked if the string matches exactly with a community string. If this is the case, in the rest of the flowcharts it could be considered as an exact match on the town string, with the set of locations in the community used instead of the regular set of locations for a matching town string. So, if the algorithm outputs "best street in (exact matching) town", the streets considered are all streets in the community.

Use the community in the preprocessing procedure
We give two suggestions to use the community field in the preprocessing procedure.

The first suggestion is to add the locations from an exact matching community to the candidate set of locations, if the preprocessing procedure removes all locations in the community. This would for example be useful, in the case the targeted street string in the community is accidentally assumed to be located in a town, which is not in the community.

Another possibility is to simply use the community field in the preprocessing procedure to reduce the number of candidate locations. Just as the town and street strings, the padded trigrams can be created from the community strings and stored as keys in a map.

Use the community as full member in the flowcharts
The last (and obvious) suggestion is to use the community in the matching algorithm in the same degree as the other fields are used. Or, equivalently, to incorporate the community field in the flowcharts. We assume trigrams Dice's will be a good measure to match community strings.

For a few flowcharts it is clear how the algorithm needs to change the output it produces:

- Only the community: do the same as for other the other input types (top three or only exact matching).
- Community + street and/or zip code: use the same flowcharts as the ones with the town instead of community given as input, together with a street and/or zip code string. Instead of the locations in the (exact matching or best) town, the locations in the (exact matching or best) community are used.
- Community + town: return also best town in best community, if not given yet.

In all other flowcharts (note that in that case both the community and town are given), we only need to make adjustments if the (best) town is not located in the (best) community, because otherwise the community does not yield any new information. If the best town is indeed not located in the best community, we adjust the flowcharts as follows:

- Community + town + street. See Figure 4 on page 58. In all four parts of the flowchart, in which the algorithm uses the similarity measure to decide which locations are given as output, we also give as output the "best street in the community". This will replace the location that is given as second of a certain type (e.g. the second best from "2 best streets in town"), or instead of the average similarity. The reason is that we also want to return the best street in the best community, because the best town is not located in the best community. In this way we still make use of exact matching strings, but also use the information in all non-matching fields to find the targeted location, when the exact matching string does not not point to the targeted location.

- Community + town + zip code. See Figure 5 on page 60. Similar to the flowchart adjustment in the previous point, we can adjust the flowcharts in all parts by returning "best zip code in the community". It again replaces the location that is given as the second of a certain type (or, replaces the location given as third. This is when the town matches, but there is no-match on zip code).

- Community + street + town + zip code. See Figure 7 on page 64. If two, or three, out of four input fields point to the same location, we only return that location (or, those locations, because it may be more). In all other parts of the flowchart "best street at best community" replaces the third suggestion. The idea is again that we want to explore the locations in the best community. This can be done best by finding the best street in the community (as zip code matching is more difficult and the best town is not in the community).

  Most interesting is the choice to not make use of the match on the street string, in case zip code, town and street all match. The reason is, that we suspect that we should find the targeted location in the best community or in the best town, because otherwise they are both erroneously entered. On all other parts in the flowcharts, we think it is intuitive to search the best street in the best community.

6.4 Lacking community or zip code

A difficulty CQM faces, is the lack of the community and/or zip code for some locations in some countries.

Lacking zip code

The main idea to deal with lacking zip codes is actually very simple: just make the most use out of the zip codes that are in the database. If the best zip code needs to be returned, find the best zip codes from the zip codes that are available. If the best zip code in a street or town needs to be

returned, but no zip codes are available, then just pick an arbitrary location.

Lacking Community
A lacking community is different than a lacking zip code, in a sense that a zip code is often more specific than a community. This means that it is very likely each community will be presented in the database at least once. Hence, there will be no differences in finding the best community compared with the original case. The best town, street or zip code in a community needs to be returned from the locations for which the community is known (just as for zip codes, make use of the available data).

6.5 Uncertain town

In this section we present our suggestions on how to adapt the matching algorithm while having a specific problem setting, which CQM encounters in practice. In the database CQM uses, it is for many locations unknown to which town a location belongs. Instead of a single town string, a list of towns is given with for each town a number indicating the likeliness that the location belongs to the town. The maximum number of possible towns given per location is five (and often five towns are given as suggestion). The most likely town is assigned the value zero.

A small analysis of the dataset of locations in Germany showed that very often (49 out of 50 randomly picked locations) the town with the value zero actually is the correct town (we compared with other geographical data to determine this). However, we also found a location for which the actual town has the lowest likeliness value out of five suggested towns. In the likeliness values itself we did not find an exploitable structure.

Our suggestion is to use a similar matching approach as presented in the previous chapter of this thesis. To match town and street strings the trigrams Dice's measure is advised, as well as the zip code matching algorithm to match zip codes. We will present adjusted flowcharts and how to adapt the preprocessing procedure.

6.5.1 Adaption of flowcharts

The revised flowcharts are not presented itself in this chapter, but are given in appendix A, from Figure 10 to Figure 16. The flowcharts differ only in some of the leaves, which are visualized in bold. We go through the main ideas on which the matching algorithm is adjusted, but we do not go into the same level of detail as when explaining the regular flowcharts. For the adjustments, see the bold leaves in the flowcharts in the appendix. The adjustments of the flowcharts are based

on the observation that the most likely town is often the real town corresponding to the location. We attach the most value to the town that is rated as best and treat the other suggested towns as suggestions equally likely.

In general, for most of the occurrences of a town in a database, the town is not given as first suggestion for the location (note that this follows directly from the observation that often five towns are given as possible towns). For these locations it is not very likely the town corresponds to the location. On the other hand, for the locations the town is rated as the best town, the town will often be the actual town. If the best location in a certain town needs to be found, the idea is to focus on the set of locations for which the town is actually rated as most likely. If we would instead return the best location from all locations for which the town is given as suggestion, we will often find a location which is not in the town in reality. To do this, we create two sets for each town: a "0" set, with locations for which the town is valued as most likely, and an "all" set, with all locations for which the town is given as one of the possible towns (including the locations already in the "0" set).

For street names and zip codes (presumed that the zip code gives detailed information about the location), there will often only be a small number of corresponding suggested towns. Compared with finding the best location in a town, the probability is higher the actual town will be returned from that small list of towns. We will therefore focus on all suggested towns corresponding to a zip code or street. By implementing this properly in the flowcharts, we make it possible to return the targeted location, even when the correct town is not given as most likely town. Also for streets and zip codes we use "0" sets and "all" sets in the flowcharts. The "0" set consists of the most likely town(s) corresponding to the street or zip code and the "all" set consists of all suggested towns, including the most likely.

As said before, we do not go into too much detail about how the flowcharts are changed exactly. Instead, we give one example how the ideas described in the previous two paragraphs influence the flowcharts. Before we give the example, first note that the flowcharts for single string input do not have to be changed, because it is still possible to make a list of towns, streets and zip codes and define which are best from these lists. Besides, the flowchart representing the matching algorithm for the combination street/zip code does not have to be changed, because the information we have for those two string types is the same as in the original case.

A good example to illustrate how the flowcharts are adjusted is the following. Assume the input consists of a town and street string and that the town string matches with a string in the database, but the street string does not. The output given by the algorithm is as follows:

- 2 best streets in town in the "0" set (the "0" set is the set of locations for which the town is

given as first suggestion)

- Best town having best street in "all" set (the "all" set is here the set of all suggested towns corresponding to the street)

The algorithm exploits the matching town string by giving the two best streets in town. It investigates only the streets that are in the "0" set of the town, because it is likely that the locations in the "0" set are in the town in reality. By returning the best town having the best street in the "all" set, it becomes possible to return the targeted location when the actual town is not given as most likely town (provided that the best street actually is the targeted street).

6.5.2 Adaption of preprocessing procedure

The preprocessing procedure can be done similarly to the original problem. A location is left in the candidate set if at least one of the suggested towns shares more than two trigrams with the input string. The preprocessing procedure will consequently take more time, but it is very unlikely that it will become an issue in practice. In Section 5.2, we referred to a paper in which it is showed that it is possible to apply the preprocessing procedure on a large database in virtually no time. Multiplying the lengths of the lists of string ids by a factor up to five will not introduce a problem, considering the example in the paper.

7 Conclusions and Future work

The thesis can roughly be divided in two parts:

1. In Chapters 3 and 4 it is described how to find the best similarity measures, to match an input string representing a street or town with the targeted string from the same type in the database. "Best" is defined as a combination of matching quality and calculation speed. The latter is important, because of the practical requirement that the similarity measures need to be able to do the calculations fast enough, such that user should not have to wait on the outcome. The input string may differ from the targeted string due to several errors like misspellings, alternative spellings, abbreviations, different word orders, additions of a string and omission of a substring. It is assumed that it is known in which country the street or town is located and that the country is a European country. In this part also an algorithm is proposed to match an input string of the type zip code, because we determined that measuring the similarity between two zip codes, using a conventional similarity measure, is not appropriate. The tests are based on locations in The Netherlands.

2. In the second part (Chapter 5 and 6), an algorithm is proposed that tries to match an input location with the targeted location in the database. An input location consists of a

combination of up to three input strings from the types town, street and zip code. It is assumed that it is known to which type an input string belongs and that the country in which the targeted location should be, is known and European. Also this matching algorithm is tested on locations in the Netherlands.

In the first part, four datasets with locations as requested in reality are used, to determine the best similarity measures for street and town strings. We selected similarity measures from the available literature to compare with each other. We defined several error types (e.g. typographical errors, abbreviations, etc.) and determined the performance of the similarity measures on the different error types.

After comparing the performance of the measures on matching quality, the capability to return obvious matches and time complexity, it turned out that the trigrams Jaccard and trigrams Dice's similarity measures yield the best results, for both town and street strings. Both measures have a very good performance on basically all different error types, except for abbreviations. Trigrams Jaccard divides the number of shared padded trigrams (i.e., the number of padded trigrams both in the input string and the string in the database) by the number of padded trigrams in the longer string. Trigrams Dice's divides the number of shared padded trigrams by the average number of padded trigrams in both strings. Trigrams Jaccard should be used when the main inconsistency between the input string and the targeted string is a typographical error. If the majority of the errors lead to a large difference in the size of the input string and the size of the targeted string (e.g., due to abbreviations, substring additions and substring omissions), trigrams Dice's should be used.

The proposed zip code algorithm is based on the observation, that zip codes in almost all European countries have the property that the first few letters or numbers point to a region in the country and the last few letters or numbers to a specific part in the region. The two main operations the algorithm performs are the search for zip codes in the database having the longest common prefix with the input zip code, and the inconsistency matching between two zip codes with the Damerau-Levenshtein measure. The Damerau-Levenshtein measure is used, because we expect zip codes given as input will often differ from the targeted zip code due to a typographical error.

In the second part an algorithm is proposed that determines which output should be returned to the user, given an input location. The algorithm uses the trigrams Dice's measure to match street and town strings and the zip code matching algorithm to match zip codes. The main idea on which the "output algorithm" is developed, is that if a string of a certain type matches exactly with a string of the same type in the database, it is very likely the user really targets that string. Depending on which input fields are used and which strings match exactly with a string in the

database, the algorithm suggests up to three locations to the user.

To reduce the calculation time of the algorithm, a preprocessing method is presented. After the input strings are given by the user, the algorithm determines in very little time which locations in the dataset are candidates to be targeted. The percentage of locations left as possible targeted location is very often less than 1.5%, which significantly reduced the running time of algorithm.

A comparison of the matching algorithm proposed in this thesis with the current matching algorithm used by CQM, shows very promising results. For three different datasets, our matching algorithm performed significantly better. On average 12% more targeted locations are returned to the user than the current matching algorithm does. The percentages of correctly returned locations are 97.36%, 99.18% and 99.70%.

Also suggestions are given about practical problems that could be faced and about the extension of the problem in which also a community field can be used.

7.1 Future work

There are two very important subjects to investigate in the future:

- How will the algorithm perform on other countries in Europe and in the rest of the world, because the algorithm and measures are tested only on locations in the Netherlands. We do expect that the trigrams are distinctive enough to yield good matching results for all European languages and we think the matching algorithm will yield the expected positive results by making use of exact matching strings. However, we cannot prove it. Also useful is to test the algorithm that finds common pre- and suffixes for several other countries.
- Another obvious extension is to develop a matching algorithm for input locations that are given as one single string, maybe even without knowing in which country the targeted location is in. It should be noted that this problem is very different than the problem investigated in this thesis, mainly due to the fact that we first have to define to which type (e.g. zip code, street, town, etc.) a string belongs. Determining how to do this in a proper manner will probably be difficult without any training data and knowledge about foreign languages. The task to return a location fast enough will also be more challenging. We think that the concepts of the preprocessing step, the use of trigrams measures and the matching algorithm itself (with the main focus on finding exact matching strings), could be used in developing a solid framework for this extended problem.

A Flowcharts “Uncertain town”

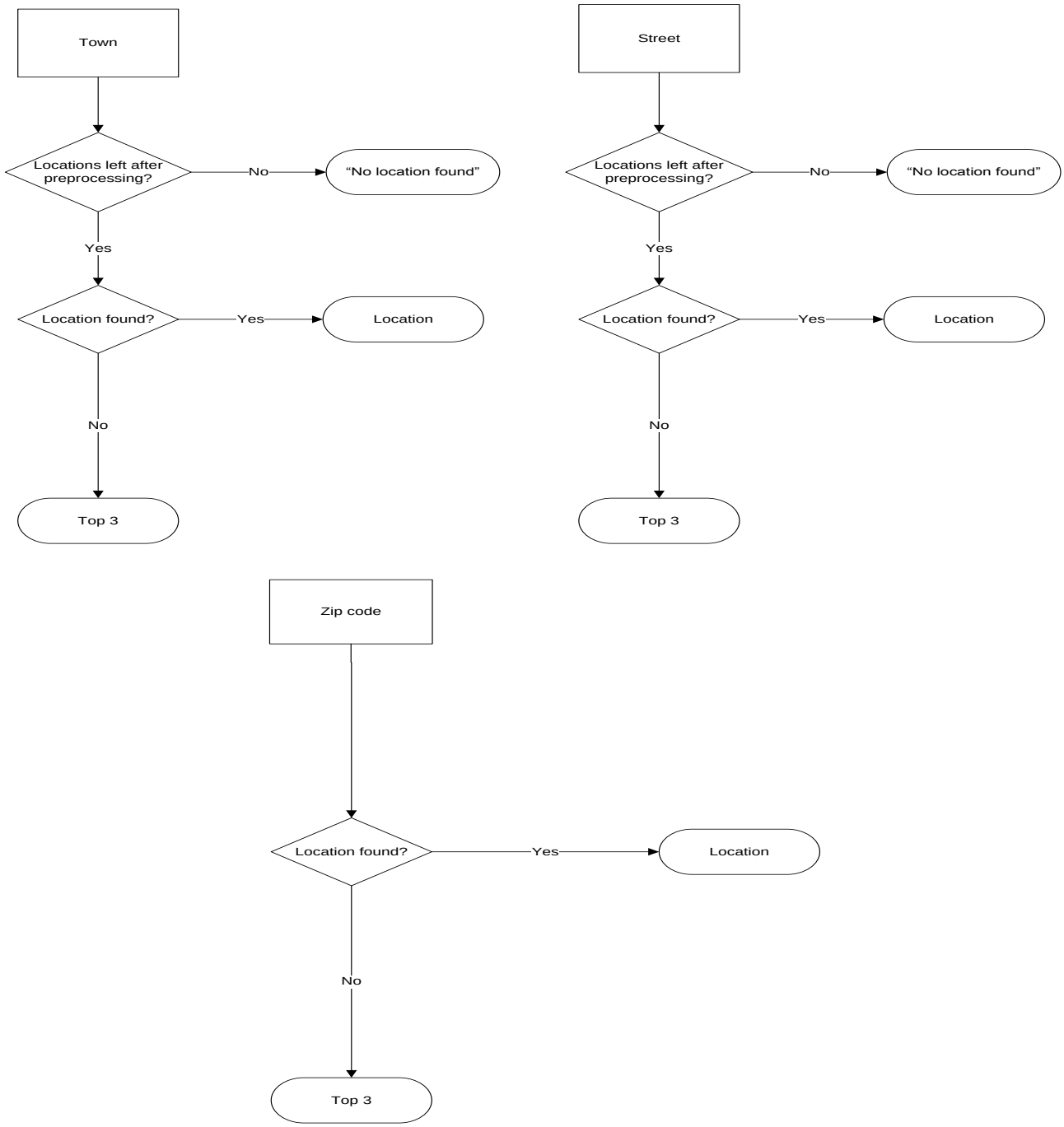

Figure 10: Uncertain town: town, street or zip code

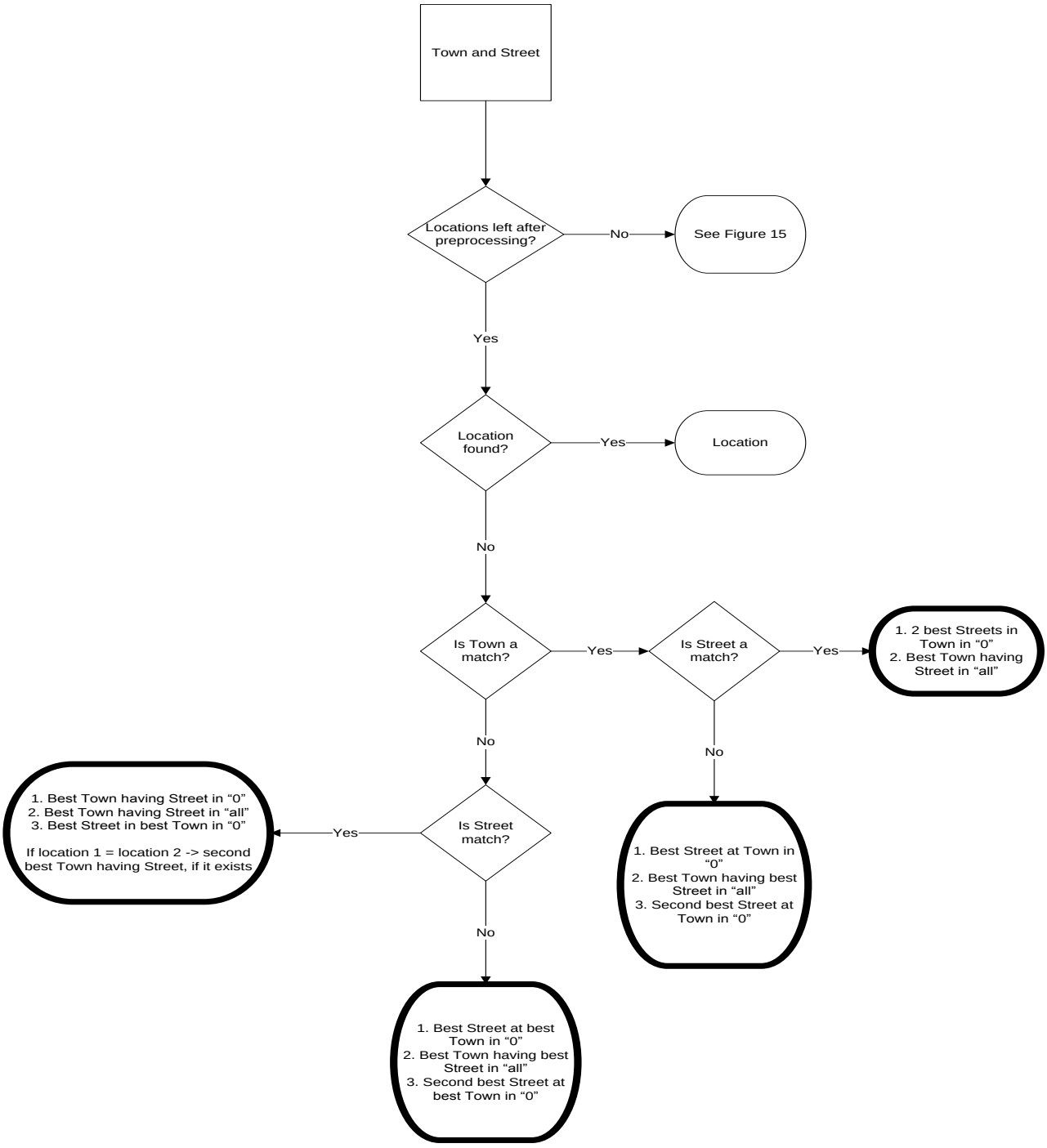

Figure 11: Uncertain town: town and street

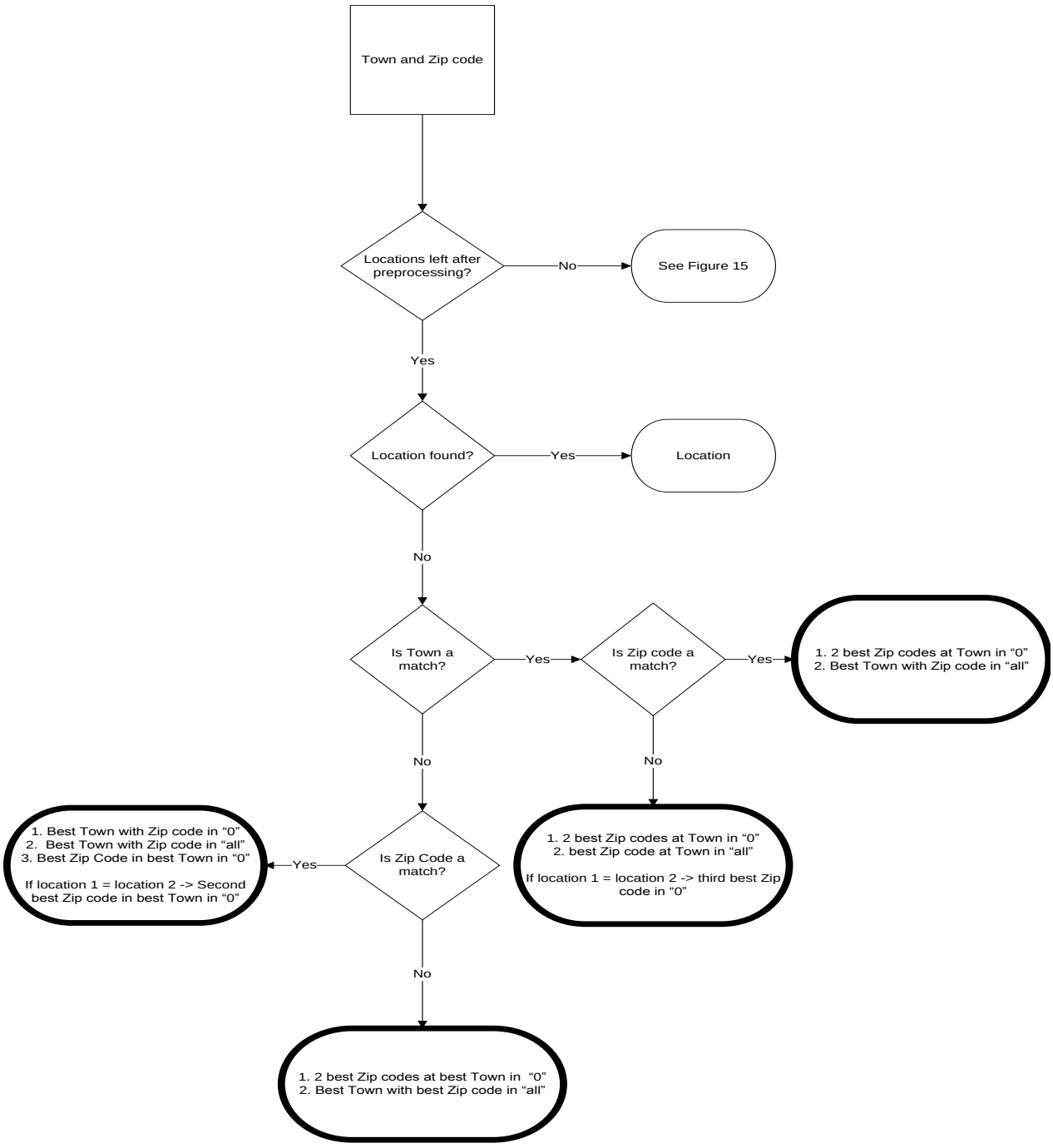

Figure 12: Uncertain town: town and zip code

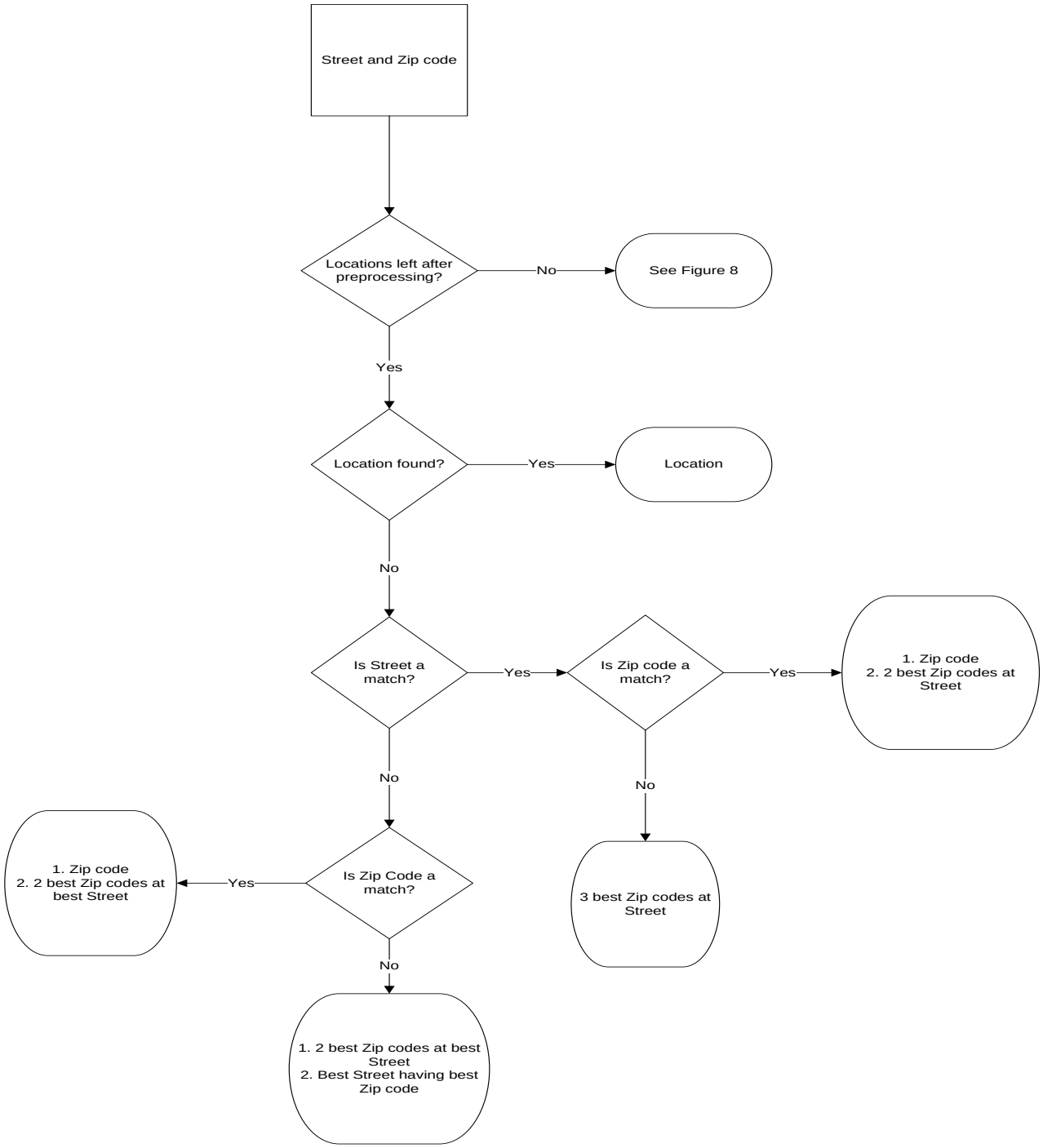

Figure 13: Uncertain town: street and zip code

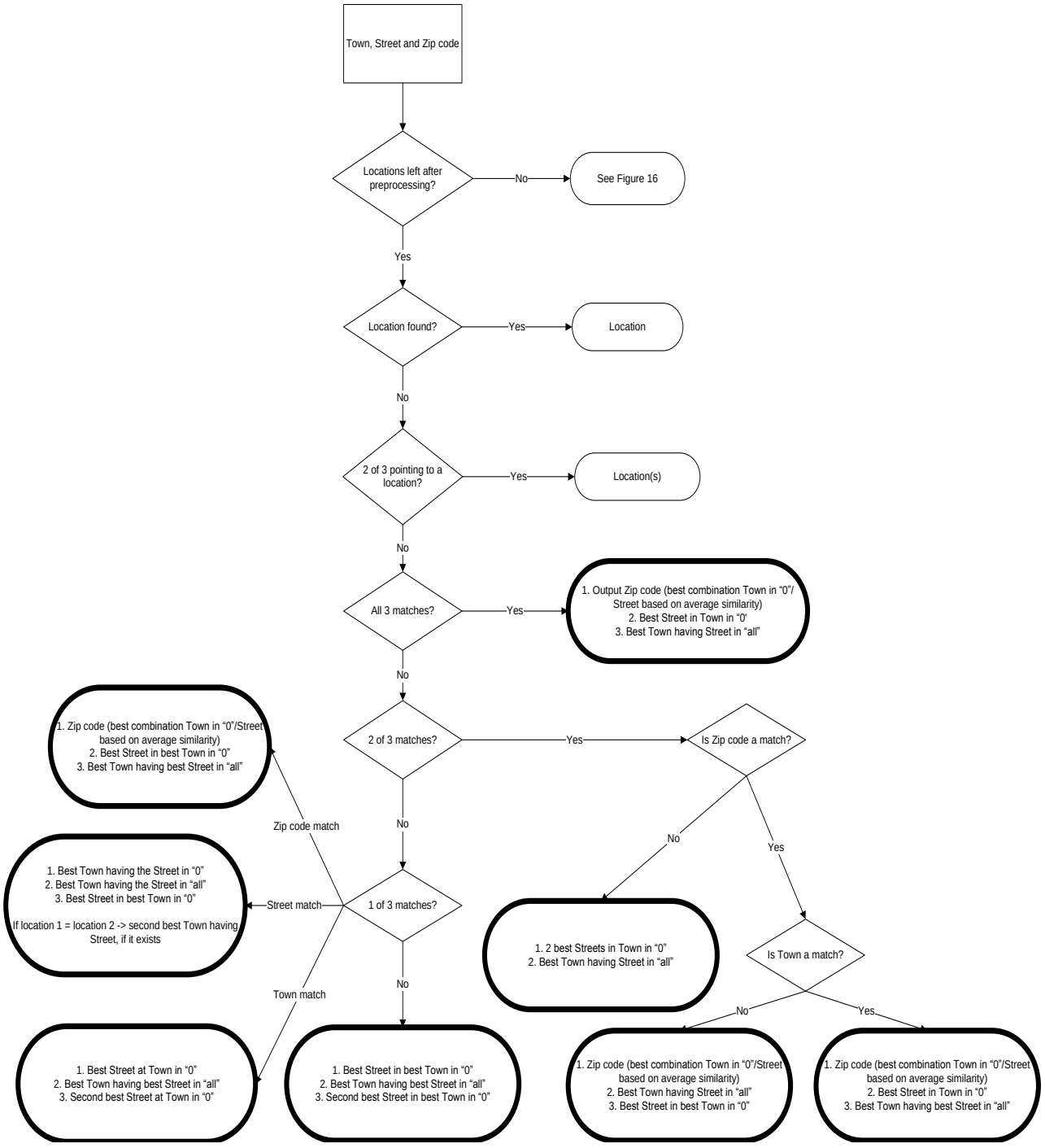

Figure 14: Uncertain town: town, street and zip code

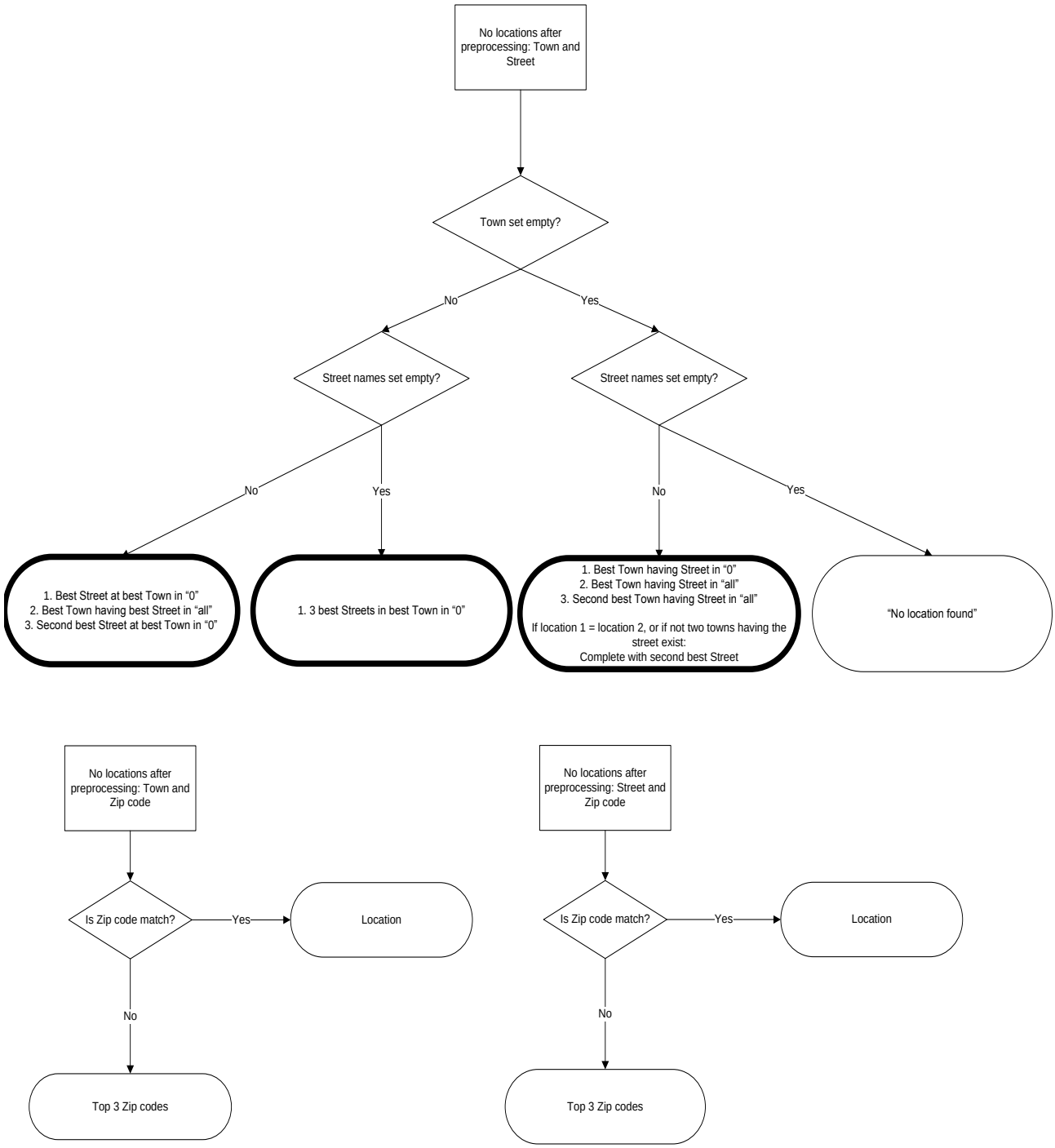

Figure 15: Uncertain town: No locations left after preprocessing I

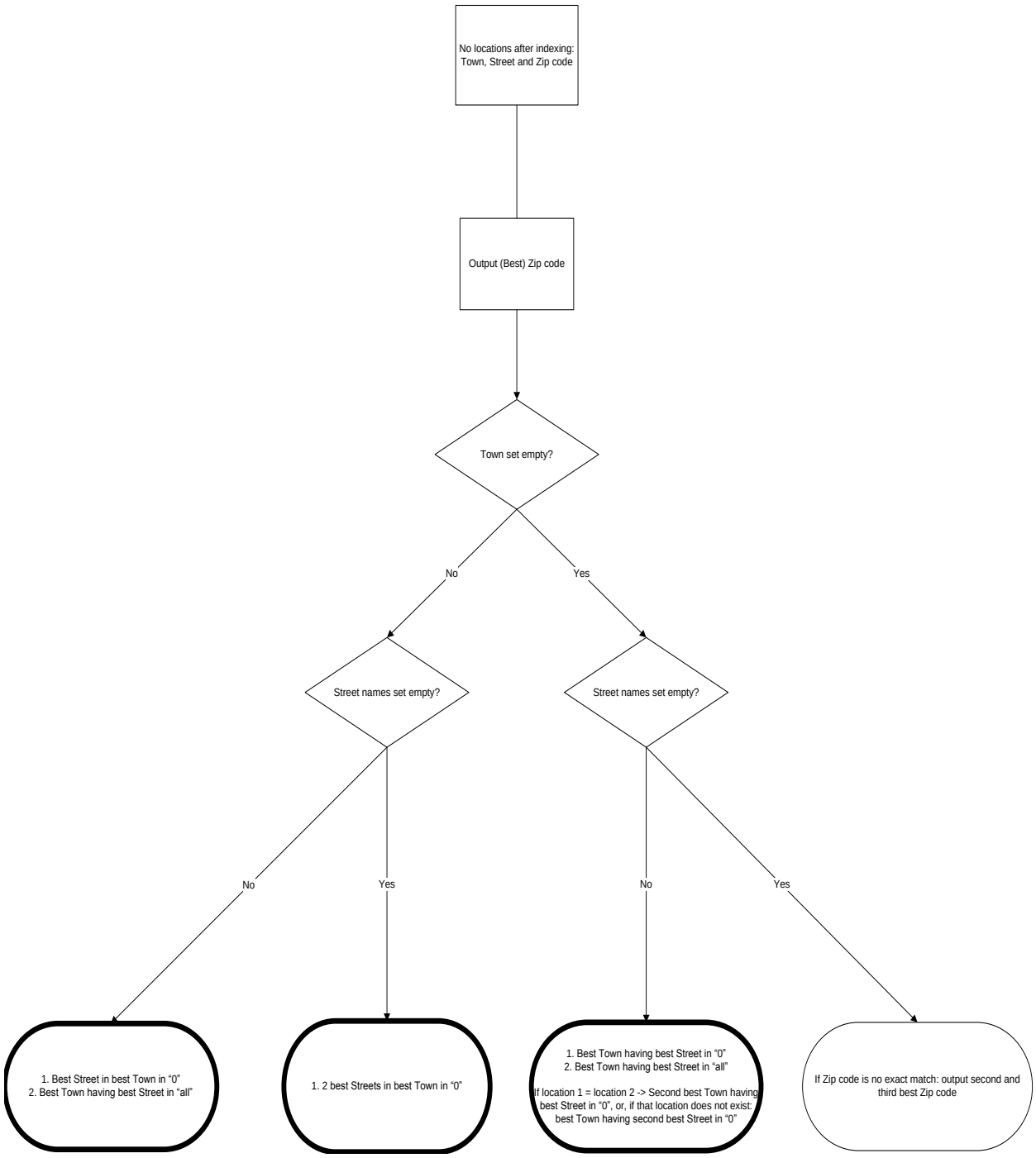

Figure 16: Uncertain town: No locations left after preprocessing II

References

[1] G. Bard. Spelling-error tolerant, order independent pass-phrases via the Damerau-Levenshtein string-edit distance metric. *Fifth Australasian Symposium on ACSW Frontiers*, 68:117–124, 2007.

[2] M. Bilenko, R. Mooney, W. Cohen, P. Ravikumar, and S. Fienberg. Adaptive name matching in information integration. *IEEE Intelligent Systems*, 18(5):16–23, 2003.

[3] J.P. Buckley, B.P. Buckles, and F.E. Petry. Processing noisy structured textual data using a fuzzy matching approach: application to postal address errors. *Soft Computing*, pages 195–205, 2000.

[4] P. Christen and T. Churches. A probabilistic deduplication, record linkage and geocoding system. *Proceedings of the Australian Research Council Health Data Mining Workshop*, 2005.

[5] P. Christen, T. Churches, and A. Willmore. A probabilistic geocoding system based on a national address file. *Proceedings of the 3rd Australasian Data Mining Conference, Cairns, Australia*, 2004.

[6] P.E. Christen. A comparison of personal name matching: Techniques and practical issues. http://astro.temple.edu/ joejupin/entitymatching/tr-cs-06-02.pdf. *Technical Report TR-CS-06-02, Australian National University*, 2006.

[7] T. Churches, P. Christen, K. Lim, and J.X. Zhu. Preparation of name and address data for record linkage using hidden Markov Models. *BMC Medical Informatics and Decision Making*, 2(9), December 2002.

[8] W. Cohen, P. Ravikumar, and S.E. Fienberg. A comparison of string distance metrics for name-matching tasks. *Proceedings of IJCAI-03 Workshop on Information Integration on the Web, IIWeb-03, AAAI*, 2003.

[9] A.K. Elmagarid, P.G. Ipeirotis, and V.S. Verykios. Duplicate record detection: A survey. *IEEE Transactions on Knowledge and Data Engeneering*, 19(1):1–16, January 2007.

[10] I.P. Fellegi and A.B. Sunter. A theory for record linkage. *Journal of the American Statistical Association*, 64(328):1183–1210, December 1969.

[11] D.W. Goldberg, J.P. Wilson, and C.A. Knoblock. From text to geographic coordinates: The current state of geocoding. *URISA Journal*, 19(1):33–46, 2007.

[12] S. Guha, N. Koudas, A. Marathe, and D. Srivastava. Merging the results of approximate match operations. *Proceedings of the 30th International Conference on Very Large Databases (VLDB 04)*, pages 636–647, 2004.

[13] D. Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology*. Cambridge University Press, 1997.

[14] G.R. Hjaltason and H. Samet. Index-driven similarity search in metric spaces. *ACM Transactions on Database Systems*, 28(4):517–580, December 2003.

[15] S. Jimenez, C. Becerra, A. Gelbukh, and F. Gonzalez. Generalized Mongue-Elkan method for approximate text string comparison. *CICLing '09*, pages 559–570, 2009.

[16] L. Jin, C. Li, and S. Mehrotra. Efficient record linkage in large data sets. *DASFAA*, pages 137–148, 2003.

[17] C.A. Davis Jr. and Emerson de Salles. Approximate string matching for geographic names and personal names. *IX Brazilian Symposium on GeoInformatics, Campos do Jordo, Brazil*, pages 49–60, November 2007.

[18] N. Koudas, A. Marathe, and D. Srivastava. Flexible string matching against large databases in practice. *Proc. 30th Intl Conf. Very Large Databases (VLDB 04)*, pages 1078–1086, 2004.

[19] Chen Li, Jiaheng Lu, and Yiming Lu. Efficient merging and filtering algorithms for approximate string searches. *Proceedings of International Conference of Data Engeneering '08*, pages 257–266, 2008.

[20] A.E. Monge and C.P. Elkan. The field matching problem: Algorithms and applications. *Proc. Second Intl Conf. Knowledge Discovery and Data Mining (KDD 96)*, pages 267–270, 1996.

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

[22] G. Navarro and R. Baeza-Yates. A practical q-gram index for text retrieval allowing errors. *CLEI Electronic Journal*, 1(2):1, 1998.

[23] L.R. Rabiner. A tutorial on hidden markov and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–285, February 1989.

[24] S. Tata and J.M. Patel. Estimating the selectivity of tf-idf based cosine similarity predicates. *SIGMOD Record*, 36(2):7–12, 2007.

[25] M.S. Waterman and T.F. Smith. Some biological sequence metrics. *Advances in mathematics*, 20(3):367–387, 1976.

[26] W.E. Winkler. Using the EM algorithm for weight computation in the Fellegi-Sunter model of record linkage. *Proceedings of the Section on Survey Research Methods, American Statistical Association*, pages 667–671, 1988.

[27] W.E. Winkler. The state of record linkage and current research problems. *Statistical Research Division, U.S. Census Bureau*, 1999.

[28] W.E. Winkler. Duplicate record detection: A survey. *Technical Report Statistical Research Report Series RRS2006/02, US Bureau of the Census, Washington D.C.*, 2006.

[29] W.E. Yancey. Evaluating string comparator performance for record linkage. *Technical Report Statistical Research Report Series RRS2005/05, US Bureau of the Census, Washington D.C.*, June 2005.