# Efficient Error-Correcting Geocoding *

Christian Jung, Daniel Karch, Sebastian Knopp, Dennis Luxen, and Peter Sanders
Karlsruhe Institute of Technology, 76128 Karlsruhe, Germany
{christian.jung,sebastian.knopp}, karch@math.tu-berlin.de {luxen,sanders}@kit.edu

February 16, 2011

**Abstract**

We study the problem of resolving a perhaps misspelled address of a location into geographic coordinates of latitude and longitude. Our data structure solves this problem within a few milliseconds even for misspelled and fragmentary queries. Compared to major geographic search engines such as Google or Bing we achieve results of significantly better quality.

## 1 Introduction

Geocoding of a location description is the process of transforming an address into a geographical coordinate. This process has been available in geographic information systems for quite some time [9] with applications for example for route planning, validating customer addresses [1], or surveillance and management of disease outbreaks like the yearly wave of influenza [19]. However, with its ubiquitous use in modern web services (e.g., [13, 5]), requirements have become more severe: Since most of these services are free, geocoding servers must handle huge streams of queries at very low cost. At the same time, users expect instantaneous answers. Finally, inputs will frequently be fragmentary, contain misspelled names or specify combinations of town and street that are inconsistent with the database. A service is likely to be more popular and useful if it tolerates such imprecisions.

While companies offering such services have naturally worked on this problem intensively, we are not aware of academic work that offers the required combination of low latency, high throughput, and error-correction for large address files. Our original aim was to make reasonable methods available to a cooperating company and to the academic community. However, to our own surprise it turned out that our approach achieves better solution quality than the market leaders at low costs so that it might also help the industry to improve their services.

In this paper we focus on the algorithmic aspects of the problem to map information about town and street to a database entry for the intended street. We do not consider the problem of resolving house numbers or producing highly accurate output coordinates. Although this has been a major focus of previous literature, e.g., [12], we view it as orthogonal to our problem since once town and street have been correctly identified, we are dealing with much more local data and hence much smaller data volumes. For example, searching an odd house number not present in the database can often be done by a binary search in a sequence of the known odd house numbers followed by an interpolation [11, 12]. On todays servers with many gigabytes of RAM, it might even be possible to precompute all estimated positions of house numbers.

The paper is structured as follows. Section 2 outlines the basic index data structure. Section 3 explains the query algorithm, while 4 develops on a rating function to rank the matches from the query. A variant of the algorithm for the case that the address is typed into a single field is explained in Section 5. Section 6 gives a data structure that resolves ambiguous town names both as a downstream computation step and in an online setting as real-time suggestions. Section 7 reports on an experimental evaluation. Last but not least Section 8 gives conclusions.

*Partially supported by DFG Grant 933/5-1.

### More Related Work

Geocoded data used to cost several dollars per 1 000 records in the mid-eighties [18] and didn't nearly provide the spatial accuracy of todays free services. At that time the use of geographic information systems was limited to professionals only that were aware of the difficulties and limitations of the geocoding process [24]. In [16], eight geocoders are compared. However, the evaluated Californian addresses are all given with high precision including city name, ZIP code, state and street name.

Improving the quality of geocoding using approximate string matching is considered desirable in [27] but then dismissed as too expensive.

Approximate string matching itself has been studied intensively, e.g. [23, 22] but most actually implemented methods match only a specific pair of strings which is infeasible for large databases. Although there is considerable theoretical work on text indices allowing approximate matching [22, 23], there are few implementations because these methods are complicated and require superlinear space. Moreover, most previous work is on finding all matches in a single large text. We will only require matching against full entries of a dictionary which is a slightly simpler problem and thus might also allow more efficient solutions. We build on one of the few available implementations [17], but still have to overcome the problem of superlinear space consumption.

Sengar et al. [25, 26] describe a system that is able to handle ill-formed queries to a certain extent. Their system does not require any country-specific rule set, but exploits the underlying geometric map data to produce a language independent representation of the data. This kind of abstraction is especially useful in areas of the world where formal address formats are non-existing, e.g., in India. However, it remains unclear how such systems scale to large databases.

## 2 Index Data Structure

Our basic input data are a set $T$ of *towns* and a set $S$ of *streets* that are fully defined by a name and a reference to a town. In this paper we use the term "town" both for a *district* of another place and for an independent place. To avoid confusion we will therefore use the term *city* for independent places even if they are small. Streets belonging to multiple towns are cut into pieces belonging to a single town. Districts additionally contain a reference to the city they belong to.

### 2.1 Inverted Indices and Town Lists

We view place and street names as (very short) documents containing a sequence of *tokens* separated by white space, commas or hyphens. Thus we can use methods known from full text search to support fast geocoding. In particular, we build two *inverted indices*, i.e. the town index maps tokens appearing in town names to the towns using that token in their name and the street index maps tokens appearing in street names to all streets containing this token.

In addition to the above inverted index, we precompute the set towns($s$) of town IDs containing a street with name $s$ and also a set towns($t$) of town IDs with name $t$. This translation to town IDs will enable us to quickly determine which combinations of town and street name correspond to actual addresses.

### 2.2 Ignoring Light Tokens

While indexing by token makes the index more convenient to use, it introduces a serious problem. A street query of the form "`New Hollywood Street`" will return *every* street that matches *any* of the tokens "`New`", "`Hollywood`", or "`Street`". In fact, about 25% of all street names match the token "`Street`", therefore we would get a really big candidate set. To avoid this problem, we adapt a concept which is often used in information retrieval and text mining [4, 28, 8]: The *inverse document frequency* of a token $c$ with respect

to a set $M$ of strings (town or street names in our case) is defined as

$$\mathrm{IDF}(c) := \log_2 \frac{\sum_{x \in M} |\,\mathrm{tokens}(x)|}{|\,\{x \in M : c \in \mathrm{tokens}(x)\}\,|}$$

where $\mathrm{tokens}(x)$ is defined as the set of tokens making up string $x$. Tokens that occur very often in the document (such as "`Street`") receive a lower IDF weight than those that appear only infrequently. Tokens that receive a high weight are more helpful in identifying the correct string, because they match fewer strings in the index.

We will use these observations to our advantage: When a user enters an address that they want to have geographically referenced, they may leave out parts of the address that they deem irrelevant, but they will probably enter those parts of the query that will non-ambiguously define what they are looking for. In our example, the user may leave out either "`New`" or "`Street`", but they most definitely won't leave out the token "`Hollywood`", which is also the token with the highest IDF weight among those three. If we expect the user to enter the most important part of an address, it is not necessary to have said address be referenced also by the remaining, unimportant tokens, i.e. we don't want to find the street "`New Hollywood Street`" by the token "`Street`", because we expect that the more descriptive token "`Hollywood`" will be entered anyway. Let

$$w_t^s := \frac{\mathrm{IDF}(t)}{\sum_{t' \in \mathrm{tokens}(s)} \mathrm{IDF}(t')}$$

be the *relative weight* of the token $t$ in the string $s$. In our example the relative weights might be 0.31 for "`New`", 0.6 for "` Hollywood`", and 0.09 for "`Street`", respectively. If we decide that the query must contain tokens that make up a fraction $\mu$ of the total weight, then we can ignore the lightest $k$ tokens, if the sum of their weights is not greater than $\mu$. E.g., for $\mu > 0.4$, we can ignore the tokens "`New`" and "` Street`". Note that this local definition of importance is very different from the stop words that are completely ignored in some inverted index data structures to save space. For example, suppose our database contains streets named "`Rhododendron Alley`" and "`Alley Street`" then it is likely that the index for "`Alley`" will contain "`Alley Street`" but not "`Rhododendron Alley`". This both save space, query time, and useless candidates.

### 2.3 Approximate Token Indices

We also build indices supporting approximate search on the sets of *tokens* appearing in town names and street names respectively. This design decision has two crucial advantages over the more obvious choice to have an index on the town and street names themselves. First, since the dictionaries are much smaller than the full data base, we can afford superlinear space to some extent. For example, our German input set contains 1.35 million streets but only 219 thousand distinct tokens for street names. Furthermore, token based indices can easily handle queries that drop part of the town or street name. For example, most users just type "`Frankfurt`" when they are looking for "`Frankfurt am Main`". The approximate index [17] for token set $M$ with maximum error $d_i$ can be queried with a string $q$ and returns a set $M_q \subseteq M$ of tokens that have edit distance (Levenshtein distance) at most $d_i$.

## 3 Multi-Field Search

We first concentrate on the case where a query consists of two strings typed into separate fields for town name and street name. Note that in this case it is easy and largely orthogonal to allow additional fields for house number or ZIP code. After normalization and tokenization (Section 3.1), we try three increasingly sophisticated ways to obtain sets $C_{\mathcal{T}}$ and $C_{\mathcal{S}}$ of town and street *candidates* respectively that allow an increasing number of errors (Section 3.2–3.4). After each of these attempts, we combine these candidate sets into consistent candidate addresses from $C_{\mathcal{T}} \times C_{\mathcal{S}}$. We stop as soon as we have found satisfying solutions. When at the end no reasonable solutions have been found, an empty result is returned.

### 3.1 Initialization Phase

The input strings are first scanned and transformed into a set $Q$ of tokens and normalized to lower-case. This is also the place where some culture-specific preprocessing can be done. In our German implementation, there is only one such specialty: The German words for "`Street`", "`Lane`", ... are sometimes used as a separate word and sometimes as a suffix and nobody really knows which version is correct in every case. Hence, compounds with these suffixes are broken into a normal form with separate tokens. This even works when the suffixes are misspelled or abbreviated.

### 3.2 Partially Exact Town Match

Following the successful principle of "make the common case fast", we use a simplified special treatment for the case of a *partially exact town match* where at least one sufficiently rare token of a town candidate is exactly matched. If this already yields a plausible result, we stop. For example, in the query "`Franfrt am Main, Römerberg`", "`Frankfurt`" was misspelled. But the token "`Main`" is an exact match and therefore we consider the set of towns that contain this token somewhere in their name before we look at all possible approximate matches. If we find a street similar to "`Römerberg`" in one of the candidates, we can stop.

### 3.3 Periphery Search

If we successfully identified partially exact town matches during the first phase, but could not match a street in these towns with a sufficient rating, we extend the scope of exact search to the *periphery*: If the input specifies a city, we try all its districts, if it specifies a district, we try the city it belongs to and all its districts.

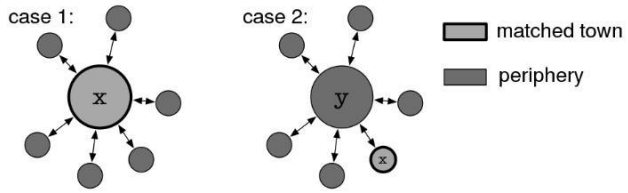


Figure 1: Two Cases of Periphery Search

If the town provided by the user is matched against a candidate $x$ which is subsequently corrected to a town $y$ in the periphery of $x$, we still calculate the rating for $x$, because the name of $y$ generally does not match anything in the query string and would lead to a low rating.

### 3.4 Approximate Search

When there are no or no good partially exact matches or when even periphery search does not find a good candidate, additional candidates are computed using the approximate indices for towns and streets. If a town candidate found specifies a district $x$ of a city $y$, we also add $y$ to the candidates. However, we do not do a full scale approximate periphery search because this could easily yield results that are hard to understand.

### 3.5 Finding Compatible Candidates

After partially exact matching, periphery search, or approximate search, that all treat towns and streets separately, we generate address candidates where town and street are compatible with each other. A pair

$(t, s) \in C_{\mathcal{T}} \times C_{\mathcal{S}}$ is compatible if a street with name $s$ is present in some town with name $t$, i.e., we have to compute the set

$$C_{\mathcal{T}\times\mathcal{S}} := \{(t, s) \in C_{\mathcal{T}} \times C_{\mathcal{S}} : \text{towns}(t) \cap \text{towns}(s) \neq \emptyset\} \quad .$$

There are various ways to do this more efficiently than the naive way of computing $|C_{\mathcal{T}}| \times |C_{\mathcal{S}}|$ set intersections. Appendix A gives one particularly efficient implementation.

# 4 Rating Candidates

After we have dismissed most of the search space, we are left with a hopefully small set of compatible address candidates $(t, s) \in \mathcal{T} \times \mathcal{S}$. These are then *rated*. The result is interpreted using two threshold values $\underline{\rho}$ and $\overline{\rho}$. Ratings below $\underline{\rho}$ are considered unsatisfactory. If all results are unsatisfactory, more extensive search is done (after partially exact matching or periphery search) or, when everything failed, an empty result is returned. In contrast, if a candidate with rating $\geq \overline{\rho}$ is found, the search returns successfully without further attempts at refined searching. Depending on the application we can then return the top ranked candidate or a list of good candidates.

To develop a rating heuristic, let us recall the different kinds of errors that we want to compensate for:

- Typing errors
- Missing or redundant tokens
- Inconsistent pairing of a street and a town.

Since periphery search and candidate filtering have already dealt with inconsistent candidates, we are left with the first two issues.

The first step on the way to a robust rating heuristic is to align the query to a candidate, i.e. find a good mapping from the tokens in the query to the tokens in the candidate (see Section 4.1). Based on this mapping, we then compute the actual rating.

The rating is computed separately for town and street by the same method and combined by the arithmetic mean afterwards. Hence, the following explanation details the town rating only. There is one small asymmetry however that we call *filter by edit distance*: Since there are usually less candidate towns than candidate streets, we first filter out candidates that are already unsatisfactory because they do not sufficiently well fit the town description in the query.

## 4.1 Matching the Query to a Candidate

To match the town tokens $q \in Q$ given by the users to the tokens of a candidate town name $c \in C$, we solve a *minimum weight perfect matching problem* on a bipartite graph. If $|Q| \leq |C|$ we add $|C| - |Q|$ *dummy* nodes to $Q$ and obtain the matching graph $G = (Q \cup C, Q \times C)$ where the weight of edge $(q, c)$ is the edit distance between $q$ and $c$ if $c$ is not a dummy node and 0 if $c$ is a dummy node. Edit distances take misspellings into account and dummy query nodes go a long way to model missing tokens in the query. Similarly, but perhaps less importantly, if $|Q| > |C|$ we add $|Q| - |C|$ *dummy* nodes to $C$. This matching problem can be solved in polynomial time (e.g. [2, 20]). Moreover, the considered graphs are very small so that solutions can be computed quickly. See Figure 2 for an illustration.

## 4.2 The Rating Function

Based on the matching, we calculate a rating for each candidate. The rating should take into account the following considerations:

- Each token that matches with at most $d$ errors should be awarded some points. It makes sense to choose $d$ larger than the error bound $d_i$ for the approximate index since space or index access time is no issue for the pairwise distance computations used for the rating function.

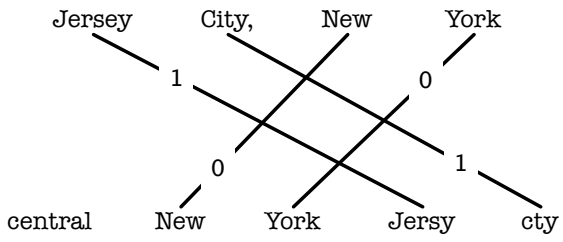


Figure 2: Candidate (top) is matched against query (bottom). Edge labels symbolize edit distances.

- Tokens that could not be matched with at most $d$ errors should not be awarded points and may even be punished.
- The user is more likely to omit information (either because they forget it or because they deem it unnecessary) than to over-specify the query. Therefore, tokens in the query that don't match anything in the candidate should be punished higher than candidate tokens that don't match anything in the query.
- The rating should be a real number in the interval $[0, 1]$, with one denoting a perfect match.
- The heuristic should be able to distinguish between tokens that are *important* and tokens that do not provide much information.

Rather than directly using the edit distance, we also want to take into account the lengths of compared words, since the rate of error that can be introduced into a word with a constant number of changes depends on its length. We therefore define *token similarity* between two tokens $q$ and $c$ as

$$\mathrm{sim}(q,c) := \begin{cases} 1 - \frac{\mathrm{editDistance}(q,c)}{|c|}, & \text{if editDistance}(q,c) \leq \mathrm{d} \\ 0, & \text{else} \end{cases}.$$

We normalize the error rate by the length of $c$ since candidates are entries that are actually present in our database.

In order to take the *importance* of a candidate token into account, we use its inverse document frequency (see Section 2).

Let $M$ denote the set of edges $(d, c)$ between query tokens and candidate tokens that were matched with edit distance $\leq d$. Let $U$ denote the set of unmatched query tokens, i.e., those tokens that could not be matched to any candidate token with at most $d$ errors. We can now define our rating function

$$\mathrm{rating}(Q,C) := \gamma\, \mathrm{rating}^Q(Q,C) + (1-\gamma)\, \mathrm{rating}^C(Q,C)$$

where

$$\mathrm{rating}^Q(Q,C) := \frac{\sum_{(q,c)\in M} (\mathrm{sim}(q,c))^\alpha\, \mathrm{IDF}(c)}{\sum_{(q,c)\in M} \mathrm{IDF}(c) + |U|\, \mathrm{IDF}_{avg}} \text{ and}$$

$$\mathrm{rating}^C(Q,C) := \sum_{(q,c)\in M} \mathrm{IDF}(c) / \sum_{c\in C} \mathrm{IDF}(c)$$

Where $\mathrm{IDF}_{avg}$ is the average over the IDF-values of all town tokens. The term $|M_c|\, \mathrm{IDF}_{avg}$ expresses that the unmatched queries should have matched somewhere but we have no idea where – so we use an average IDF-value. The parameter $\alpha$ is used to adjust how important it is to have similar matches. Notice that

rating$^Q$ is not influenced by the number of unmatched *candidate* tokens. This is why we compute a convex combination of rating$^Q$ with rating$^C$ which penalizes unmatched candidate tokens. The parameter $\gamma \in [0, 1]$ specifies the relative weight. Usually we want to give more weight to the matched parts of a query, therefore we choose $\gamma > 1/2$.

# 5 Single-Field Search

In the previous sections we focused on separate fields for town and street because this interface is more important for our cooperation partner, because multi-field search is easier to program, and one should expect that it reduces errors. From the users points of view, however, it is more convenient to enter a query into a single text field, with street and town in arbitrary order. Online services like Google Maps or Bing Maps have therefore adopted single-field search.

However, in order to compare multi-field search and single-field search and in order to compare our approach with internet services, we have also implemented a simple version of single-field search with an emphasis on quality.

Our solution is based on the plausible hypothesis that the token sequence resulting from a single-field query has the format $\mathsf{streetToken}^*\mathsf{townToken}^+$ or $\mathsf{townToken}^+\mathsf{streetToken}^*$, i.e., street and town tokens are contiguous and there is at least one token designating a town. We exhaustively try all $2m-1$ possible ways to split a token sequence of length $m$ and call a multi-field search for each of them. Refer to Section 8 for a discussion of possible optimizations.

For example, the query "`Oxford Street, London`" can be split as

| Town | Street |
|---|---|
| `Oxford` | `Street London` |
| `Oxford Street` | `London` |
| `Oxford Street London` | |
| `Street London` | `Oxford` |
| **London** | **Oxford Street** |

Only the last line would return a perfect rating as we might have hoped. This query is a bit lucky though since there is no "`London Street`" in "`Oxford`" which, if it existed, would also receive a perfect rating.

# 6 Neighborhood Graph

In most countries, city names are not unique. For example, Wikipedia knows 41 Springfields, 5 of them in Wisconsin. We present an efficient data structure here that can be used to find plausible nearby cities for disambiguation. Applications could either use the data structure to propose disambiguating places or they could match them to inputs of the user such as "`Springfield near Kansas City`".

What makes a city plausible? It should be large, and it should be close to the city it is supposed to disambiguate. Hence, we are facing a bicriteria optimization problem. A safe way to handle such a situation is to consider all cities that are not dominated by any other city. ($x$ dominates $y$ wrt city $t$ if it is both closer to $t$ than $y$ and larger than $y$.) This problem is known under several names: finding *Pareto optima*, *vector maxima* [21], or a *skyline* [7]. If the cities are sorted by size, it is easy to solve the problem with a full scan of all cities – outputting a city if it is closer than all previously inspected ones. However, looking at all cities may still be too slow.

We will encode the required information into a graph. Assume the cities are numbered 1 to $n$ by decreasing size. There is no need to store all towns, only those cities big enough to serve as a reference place. A convenient way to define the size of a city here is to just count the number of streets. Let $D_i := (\{1, \ldots, i\}, E_i)$ denote the Delaunay triangulation[1] [10] of the $i$ largest cities.

[1]Note that for convenience we work with Euclidean geometry here. It is an interesting problem to consider what happens when we do geometry on the surface of a sphere.

Then we will consider the *neighborhood graph* $D^*:= (\{1,\ldots,i\}, \cup_{i=1}^{n} E_i)$. Depending on the application, we may interpret $D^*$ as a directed acylic graph where all edges go from smaller to larger indices (downward) or vice versa (upward). The intuition here is that the Delaunay triangulation encodes a natural concept of proximity. Directing the edges *upward*, i.e., towards larger cities allows us to find such cities. Obviously, it is not enough to just consider the Delaunay triangulation $D_n$ of all $n$ cities since we would usually end up in a dead end of a medium sized cities whose neighbors are all smaller. The following theorem states that the union $D^*$ of $n$ Delaunay triangulation solves this problem very effectively:

**Theorem 1** *Consider any map position $(x, y)$. The Pareto optima with respect to city size and closeness to position $(x, y)$ forms a downward path from node 1 to the city closest to $(x, y)$. Moreover, this path can be found with the simple greedy algorithm depicted in Figure 3.*

**Proof 1** *By induction from 1 to $n$, analogous to the work of Birn et al. [6].*

Of course it is important how long it takes to construct $D^*$ and how much space it takes. If the size of a city is assumed to be independent on its position, the problem is the same as in *randomized incremental construction* of a Delaunay triangulation [14] – we get a linear number of edges in time $O(n \log n)$.

# 7 Experimental Evaluation

**Implementation** We have implemented the system described above in C++ making extensive use of the STL library. This probably leaves open significant further tuning opportunities. For example, we use a simple merging based STL algorithm for computing set intersections rather than a tuned code as used in full text indices. The tuning parameters have been chosen intuitively without an attempt at finding optimal values: We ignore light tokens comprising up to 40 % of the cumulative IDF of a name. The correction limit of the approximate dictionaries and pairwise edit distance computations is limited to $d = d_i = 2$ in order to keep space consumption low. Candidate matching use the Hungarian method [2] using the implementation by [3]. In the rating function, similarities between matched words are taken to the power $\alpha = 2$ and in the convex combination $\text{rating}^Q(Q, C) = \gamma\,\text{rating}^Q(Q, C) + (1-\gamma)\,\text{rating}^C(Q, C)$ we choose $\gamma = 3/4$. The threshold for a satisfactory rating is $\underline{\rho} = 1/2$ and a good rating starts at $\overline{\rho} = 4/5$. Experiments were performed on an Intel Core i7 920@2.67GHz with 12GB RAM on Linux 2.6.27 using a single core. The programs were compiled with GCC 4.3.2.

**Map Data** We use commercial data (from 2009) comprising all German street and town names. There are about 12 000 cities, 108 000 towns, 80 000 town names, 76 000 town name tokens, 1 350 000 streets, 560 000 of which contain the token "`Strasse`", 444 000 street names, and 269 000 street name tokens. A street name consists of 2.5 tokens on average, while town names consist of 1.1 tokens on average. The input data takes

```
Procedure ParetoOptima(q)
    u:= 1                                                  // start at top of hierarchy.
    repeat
        output u
        foreach neighbor v of u in D*
                with u < v in increasing order do
            if ||q − position(v)||_2 < ||q − position(u)||_2 then
                u:= v
                break for loop
    until no closer node found
```

Figure 3: Finding Pareto optimal cities in the neighborhood graph $D^*$.

about 30 MB space while our index data structures take about 327 MB. A more frugal assignment of space to various hash tables could reduce that to a bit more than 200 MB but we think that 327 MB is almost negligible for a server setting – even very energy-efficient 32-bit servers based on Atom, or ARM could be used.

## 7.1 Pseudorealistic Random Queries

For our experiments, we use a set of existing, *relevant* addresses $R$, and a set of non-existing, *irrelevant* addresses $I$. A relevant address is sampled by first choosing a random street name $s$ and then picking a random town from towns($s$). An irrelevant is composed of randomly chosen town and street names such that combination which accidentally occur in the database are rejected. Ideally, we would like to return correct results for relevant address queries and no result for irrelevant address queries. To generate a simple random query, it would be easiest to just insert, delete or substitute random characters in an existing address. The errors that are introduced this way, however, are unlikely to resemble the errors that a human would make while entering a query through a keyboard. We identify several sources of errors to generate input sets with more realistic errors.

Typing errors are very common and we distinguish

- swapped characters

  *Example:* "`Frankfurt`" → "`Frankfrut`"

- missing characters

  *Example:* "`Frankfurt`" → "`Franfurt`"

- superfluous or wrong characters, mostly closely located to the correct character on the keyboard (here in terms of the German QWERTZ-layout).

  *Example:* "`Frankfurt`" → "`Frankdfurt`" or "`Frankdurt`"

Depending on the respective language there are several sources of error that are phonetic.

- doubled characters where there should be a single character

  *Example:* "`Dublin`" → "`Dubblin`"

- single character where there should be two of the same

  *Example:* "`Cardiff`" → "`Cardif`"

- The SOUNDEX algorithm identifies classes of characters such that different characters from the same class differ only slightly in their pronunciation.

  *Example:* `z` ≡ `s`, "`Zaragoza`" → "`Saragosa`"

- Two consecutive vowels that occur in the same syllable are called a *diphthong*. In German, for example, several different diphthongs sound the same or similar:

  - `ei` ≡ `ey` ≡ `ay` ≡ `ai`
  - `eu` ≡ `äu` ≡ `oy` ≡ `oi`
  - ...

  *Example:* `Hoyerswerda` → `Heuerswerda`

In order to introduce $k$ errors into an address, we introduce $\lceil k/2 \rceil$ errors into the street string and $\lfloor k/2 \rfloor$ errors into the town string. To introduce an error, we first pick a random error class, then a random token, and then a random position. All distributions are uniform.

Here we report experiments on 1 000 relevant and 100 irrelevant addresses. We classify the results returned by the index as follows:

**Multi-field search**

| | relevant | | | irrelevant | | |
|---|---|---|---|---|---|---|
| **Errors** | **TP** | **FN** | **II** | **TN** | **FP** | **Time [ms]** |
| 0 | 1 000 | 0 | 0 | 93 | 7 | 3.02 |
| 1 | 989 | 10 | 1 | 95 | 5 | 2.75 |
| 2 | 988 | 11 | 1 | 94 | 6 | 2.44 |
| 3 | 928 | 66 | 6 | 94 | 6 | 2.40 |
| 4 | 854 | 140 | 6 | 99 | 1 | 1.79 |
| 5 | 557 | 431 | 12 | 97 | 3 | 1.59 |

**Single-field search**

| | relevant | | | irrelevant | | |
|---|---|---|---|---|---|---|
| **Errors** | **TP** | **FN** | **II** | **TN** | **FP** | **Time [ms]** |
| 0 | 1 000 | 0 | 0 | 52 | 48 | 26.07 |
| 1 | 989 | 10 | 1 | 63 | 37 | 23.33 |
| 2 | 986 | 13 | 1 | 74 | 26 | 19.72 |
| 3 | 927 | 66 | 7 | 75 | 25 | 18.44 |
| 4 | 856 | 125 | 19 | 80 | 20 | 16.69 |
| 5 | 560 | 414 | 26 | 86 | 14 | 14.31 |

Table 1: Matching rates and query times for random addresses – 1 000 relevant ones and 100 irrelevant ones.

- *True Positive (TP):* A relevant address that is correctly identified.
- *True Negative (TN):* An irrelevant address that does not return a result or a correct partial result (i.e. the correct town).
- *False Positive (FP):* An irrelevant address where the index does return a result.
- *False Negative (FN):* A relevant address where we don't find a result.
- *Incorrectly Identified (II):* A relevant address that returns an incorrect result, i.e. another relevant address.

The match rates, along with the query times, are shown in Table 1.

Multi-field search works extremely well, both with respect to result quality and query time. Only at five errors, when three errors are introduced into the street name, we see a sharp increase of false negative results. At this point, the approximate street index will often fail to find the right result because its error limit is set to $d_i = 2$. Interestingly, query times *decrease* with the number of errors. The reason is that we have to check a smaller number of candidates both in the approximate index and when rating candidates.

Single-field search for relevant addresses works almost as well as multi-field search. The only noticable difference is that a small fraction of the false negative results mutates into incorrectly identified results. For irrelevant addresses, our current implementation seems to be too aggressive though because it returns a significant number of false positives. On the first glance it looks paradoxical that we get a larger number of output errors when there are no spelling errors in the input. But the reason is simple: without spelling errors we obtain higher ratings for the generated candidates and thus it becomes more likely that the result is accepted. This indicates that the result quality could be improved by increasing the threshold for accepting a result. Single-field query times are an order of magnitude larger than for multi-field search. This is not surprising, since our current implementation naively factors a single-field search into several multi-field searches.

Figure 4 shows the distribution of query times for the same set of $5 \times 1\,100$ queries. 90% of all multi-field queries finish in less than 5 ms. The maximum query time observed is 106 ms. Note that for the server scenario we are considering, we want very low *average* query time to achieve high throughput and low cost.

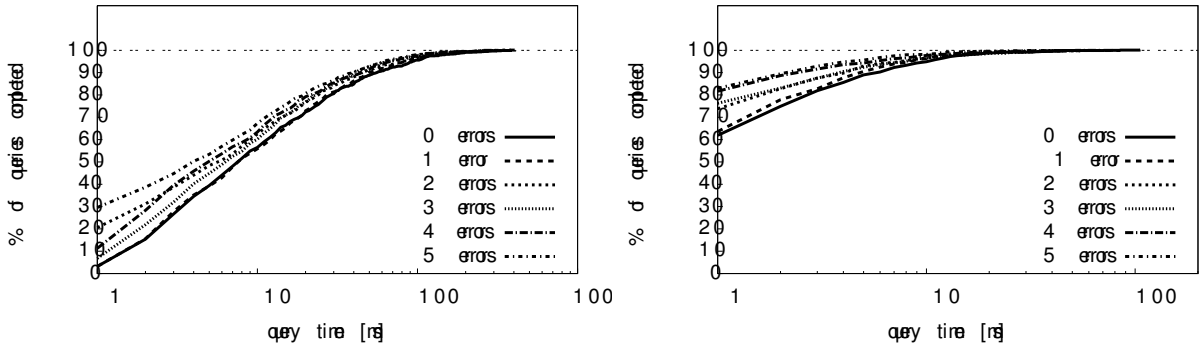


Figure 4: Query times of single-field (left) and multi-field (right) search for several numbers of distortions.

| $d$ | **Bing** | **Google** | **Ours** |
|---|---|---|---|
| 0 | 87 | 97 | 100 |
| 1 | 78 | 54 | 98 |
| 2 | 71 | 3 | 98 |
| 3 | 59 | 3 | 94 |
| 4 | 55 | 2 | 86 |

Table 2: number of resolved queries

Occasional slower queries are no problem as long as they do not lead to noticable delays for the user. 100 ms is well below the delays users are accustomed to experience due to network latencies anyway. Although single-field search is an order of magnitude slower on the average, this slow-down does not translate into a proportional increase of the slowest query times – we still remain below 406 ms. This indicates that the query times of the generated multi-field subqueries are not strongly correlated.

We also compared against existing geocoders. To do so, we took the first 100 relevant queries from the above query set and ran them against the publicly available APIs[2] of Bing and Google Maps. Table 2 reports on the number of true positive results. Google works very well for undistorted inputs – the three remaining errors could perhaps be attributed to differences in the input data. But already for a single error, the recognition rate drops to 54 % and completely collapses for $d \geq 2$. Bing already has significant deficits at $d = 2$ but fails more gracefully for distorted inputs. Still, the number of failed answers is an order of magnitude larger than for our system.

## 7.2 Real-world Queries

To get an impression of the quality of the results, we also did experiments with real-world input, i.e. actual queries that have been provided by users of an existing geocoder[3]. The test data consists of multi-field 1 383 queries that were logged from the the users of a cooperating company. The results were pre-classified by the company into five categories. We briefly describe the categories.

**Exact:** Queries where each token can be matched without errors to a token in the result. The differences allowed between query and result are those that are handled by a normalization phase.

*Example*: "`london tally road`" → "`London, Tally Road`"

**Partially Exact:** Queries where each token occurs in the result, but not each token of the result string occurs in the query.

[2]It should be noted that the subjective performance of Google with interactive use on Google-Maps is much better than over the API. In particular, the auto-completion mechanism works very well but is obviously not applicable to the API.

[3]Complete references will be given in the final version of paper.

| | | Bing | | | Google | | | Ours | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Class** | **#** | **s** | **w** | **n** | **s** | **w** | **n** | **s** | **w** | **n** |
| Exact | 100 | 81 | 5 | 24 | 100 | 0 | 0 | 100 | 0 | 0 |
| Partial | 99 | 77 | 20 | 2 | 96 | 2 | 1 | 99 | 0 | 0 |
| High | 81 | 60 | 16 | 5 | 65 | 10 | 6 | 77 | 2 | 2 |
| Medium | 34 | 7 | 1 | 26 | 16 | 11 | 7 | 27 | 6 | 1 |
| Low | 15 | 1 | 1 | 12 | 9 | 4 | 2 | 13 | 1 | 1 |

Table 3: The match rates of our geocoder on real-world queries.

*Example*: "`london tally`" → "`London, Tally Road`"

**High/Medium/Low:** Queries that contain errors. The labels High, Medium, and Low were assigned depending on the "confidence" of the old geocoder to have a correct interpretation of the input.

*Example*: "`Lodon; Tall Rd`" → "`London, Tally Road`" (e.g. classified as Medium )

We tested our address search with these queries checking correctness of our result manually. We classify the results into three categories:

**Strong Match:** A result that is unquestionably correct. In many cases the query was non-ambiguous and easy to verify.

**Weak Match:** A result that is not correct, but has successfully identified parts of the query, e.g. the town.

**No Match:** Either a result that is definitely incorrect, or no result at all.

In total, the test data consists of 1 383 queries, classified into 844 Exact, 357 Partially Exact, 125 High, 41 Medium, and 16 Low queries. Since they had to be verified by hand, we picked random samples of at most 100 queries per class. Also, we removed those that were not resolvable (e.g. for shopping centers, water parks etc. – our index does not contain points of interest). There remained 100 exact and 99 partially exact queries, as well as 81, 34 and 15 queries classified as high, medium and low. The results are shown in Table 3.

Our system outperforms the Google and Bing API in all categories. As with the real world inputs, Google is similarly good for (partially) exact queries but looses ground for errorneous inputs. For example, for category *High* our system returns correct results for all but 4 of the inputs whereas Google fails for 16 inputs – a factor four in the failure rate. An interesting difference to the random inputs is that now Google consistently outperforms Bing – also for the inputs with errors.

## 7.3 Parameters that affect the Query Time

We use several techniques to make sure that the number of candidates stays small and that we don't have to perform too many edit distance computations. To see if these techniques are necessary and how each of them affects the query time, we have performed a number of of experiments. The techniques are:

**Filter Incompatible Candidates (FIC):** As described in Section 3.5, we keep only those town and street candidates that are geographically compatible.

**Filter by Edit Distance (FED):** As described in Section 4 we have an additional filtering stage before full rating evaluation the drops candidates that can already eliminated because the town names are an unsatisfactory match.

**Ignore Light Tokens (ILT):** As described in Section 2, we can ignore some tokens during the construction of the index due to their weight in comparison to the other tokens. E.g. the candidate "`New Hollywood Street`" will be represented only by "`Hollywood`", because the other two tokens occur so frequently in the dictionary that they would not be of much help to distinguish this candidate from others.

As we can see in Table 4, disabling ILT absolutely destroys the performance. To see why this is so, consider the most frequent token in the street dictionary, "`street`". If we randomly choose any street, the probability that it contains the token "`street`" is about 1/3. Without ILT, *any* query that contains the token "`street`" will return *all* candidates that contain this token. Hence, if we randomly choose a candidate

| ILT | FIC | FED | Query Time [ms] |
|---|---|---|---|
| × | × | × | 570.00 |
| × | × | ✓ | 566.00 |
| × | ✓ | × | 199.00 |
| × | ✓ | ✓ | 126.00 |
| ✓ | × | × | 10.68 |
| ✓ | × | ✓ | 10.58 |
| ✓ | ✓ | × | 3.45 |
| ✓ | ✓ | ✓ | 2.09 |

Table 4: The effect of the features ILT, FIC and FED on the lookup time.

and query the index with this candidate, we can expect a candidate set that contains at least 1/9 of all streets, which is almost 50 000 candidates for our data. In our experiments the actual number of candidates was even bigger, 67 000 candidates on average. The query time drops by a factor of almost 100 when we enable ILT. FIC gives us another boost of factor 3 and FED makes a difference only when used in conjunction with FIC. None of these features has a noteworthy effect on the memory requirements of the index, therefore all of them should be enabled by default.

### 7.4 Neighborhood Graph

We have build a neighborhood graph for all 12 379 *cities* in our input, resulting in 71 896 edges, i.e., there are less than 6 edges per node which is similar to what we would expect for random city sizes. About 290 nodes are reachable from a city on the average, saving a factor $> 40$ compared to a full scan of the city table, even if we do not fully use Theorem 1.

We also performed a small user study to test our model. We randomly picked 48 towns from a map of Germany and asked a non-expert in the field of geography or computer science to attribute a larger reference town to the one we picked. In 46 of these cases, a Pareto-optimal reference town was chosen.

## 8 Conclusions and Future Work

We presented algorithms and data structures for error-correcting geocoding that yield instantaneous answers at costs negligible compared to the overheads for displaying maps answers, etc. over the internet. Perhaps most surprising is that we still have a high match rate with as much as four errors and this is much better than current web geocoders. Another pleasant surprise was that our combination of powerful data structures and general rating functions yields a considerably simpler solution than several rule based industrial solutions we have heard about.

Although we are already quite fast, we still have significant tuning opportunities. In particular, it will be relatively easy to further speed up single-field search. For example, we currently do not even cache accesses to the approximate dictionary or results edit distance computations. More generally, faster algorithms for edit distance computation could be tried [15].

Although our experiments have so far focused on commercial German road data, we believe that they are easy to adapt to other Western industrial countries. In particular, these countries have similar address systems and language conventions. Points of interest (POI) like gas stations or restaurants can be incorporated by treating them like a street, town or house number depending on the context. Finding street intersections is relatively easy once we have geocoded the two intersecting streets.

The basic ingredients – fast approximate dictionary search, token matching and scoring functions might also help in other settings like in countries with more complicated addresses or less structured reference data. However in that case we should expect more errors, longer query times, and the need for further heuristics.

Our concept of neighborhood graph is a promising approach to disambiguate queries involving frequent town names. One interesting aspect is that the set of nodes reachable through the neighborhood graph $D^*$ may contain further "interesting" nodes. In particular, we may want to restrict the set of reported cities to those in the same "region" (e.g., county, state, or nation) as the query town. Since Delaunay triangulations encode neighbors "in all directions", we suspect that with appropriately defined "locally well behaved region shapes" the desired output still consists of nodes reachable in $D^*$.

# References


[1] C. C. A. Tang. *ArcGIS9 Geocoding Rule Base Developer Guide*. ESRI, 2003.

[2] R. Ahuja, T. Magnanti, and J. Orlin. *Network Flows : Theory and Algorithms*. Prentice-Hall, 1993.

[3] An implementation of the Kuhn-Munkres Assignment Algorithm. http://code.google.com/p/hungarianassignment/.

[4] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. ACM Press / Addison-Wesley, 1999.

[5] Bing Maps. http://maps.bing.com.

[6] M. Birn, M. Holtgrewe, P. Sanders, and J. Singler. Simple and fast nearest neighbor search. In *Proceedings of the Twelfth Workshop on Algorithm Engineering and Experiments, ALENEX 2010*, pages 43–54, 2010.

[7] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *17th International Conference on Data Engineering (ICDE' 01)*. IEEE, 2001.

[8] S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani. Robust and efficient fuzzy match for online data cleaning. In *ACM SIGMOD International Conference on Management of data*, 2003.

[9] D. Cooke. *The History of Geographic Information Systems: Perspectives from the Pioneers.* Prentice Hall, 1998.

[10] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry Algorithms and Applications*. Springer-Verlag, 2nd edition, 2000.

[11] W. J. Drummond. Address matching: GIS technology for mapping human activity patterns, 1995.

[12] D. W. Goldberg, J. P. Wilson, and C. A. Knoblock. From text to geographic coordinates: The current state of geocoding. *Journal of the Urban and Regional Information Systems Association*, 19, 2007.

[13] Google Maps. http://maps.google.com.

[14] L. J. Guibas, D. E. Knuth, and M. Sharir. Randomized incremental construction of Delaunay and Voronoi diagrams. *Algorithmica*, 7(4):381–413, 1992.

[15] H. Hyyrö, K. Fredriksson, and G. Navarro. Increased bit-parallelism for approximate string matching. *ACM Journal of Experimental Algorithmics*, 10, 2005.

[16] J. P. W. J. N. Swift, D. W. Goldberg. Geocoding best practices: Review of eight commonly used geocoding systems. Technical report, University of Southern California GIS Research Laboratory, 2008.

[17] D. Karch, D. Luxen, and P. Sanders. Improved fast similarity search in dictionaries. In E. Chavez and S. Lonardi, editors, *String Processing and Information Retrieval*, volume 6393 of *Lecture Notes in Computer Science*, pages 173–178. Springer Berlin / Heidelberg, 2010.

[18] N. Krieger. Overcoming the absence of socioeconomic data in medical records: validation and application of a census-based methodology. *Am J Public Health*, 82, 1992.

[19] N. Krieger, J. T. Chen, P. D. Waterman, M. J. Soobader, S. V. Subramanian, and R. Carson. Geocoding and monitoring of us socioeconomic inequalities in mortality and cancer incidence: does the choice of area-based measure and geographic level matter?: the public health disparities geocoding project. *Am J Epidemiol*, 156, Sep 2002.

[20] H. W. Kuhn. The Hungarian method for the assignment problem. *Naval Research Logistic Quarterly*, 2, 1955.

[21] H. T. Kung, F. Luccio, and F. P. Preparata. On finding the maxima of a set of vectors. *J. ACM*, 22(4):469–476, 1975.

[22] G. Navarro, R. Baeza-yates, E. Sutinen, and J. Tarhio. Indexing methods for approximate string matching. *IEEE Data Engineering Bulletin*, 24:2001, 2000.

[23] G. Navarro and E. Chávez. A metric index for approximate string matching. *Theor. Comput. Sci.*, 352(1):266–279, 2006.

[24] D. Roongpiboonsopit and H. A. Karimi. Comparative evaluation and analysis of online geocoding services. *International Journal of Geographical Information Science*, 24, 2010.

[25] V. Sengar, T. Joshi, J. Joy, S. Prakash, and K. Toyama. Robust location search from text queries. In *GIS '07: Proceedings of the 15th annual ACM international symposium on Advances in geographic information systems*, 2007.

[26] V. Sengar, T. Joshi, J. M. Joy, and S. Prakash. Building a global location search service. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM, 2008.

[27] W. Winkler. Approximate string comparator search strategies for very large administrative lists. In *Proceedings of the Section on Survey Research Methods*, pages 4595–4602, 2004.

[28] H. C. Wu, R. W. P. Luk, K. F. Wong, and K. L. Kwok. Interpreting tf-idf term weights as making relevance decisions. *ACM Trans. Inf. Syst.*, 26(3):1–37, 2008.

# A Fast Computation of $C_{\mathcal{T}\times\mathcal{S}}$

We describe this algorithm in an appendix since it is an application of folklore tricks in maintaining sets of small integers and thus not a really new result. On the other hand, the applications of these tricks may not be obvious in this case so that a description is sensible. We first compute the union towns($C_{\mathcal{S}}$) of all sets towns($s$) for $s \in C_{\mathcal{S}}$. towns($C_{\mathcal{S}}$) is represented as an array $A$ with one entry for each town ID.[4] While doing this, we annotate each entry of towns($C_{\mathcal{S}}$) with a list $\ell$ of pointers to the street names that enter this town into the union. Then, for each $t \in C_{\mathcal{T}}$ and each $x$ in towns($t$) we remember $\{(t, s) : s \in A[x].\ell\}$ as compatible candidates. The algorithm is linear in the size of the inspected lists towns($t$), towns($s$), and the output size.

[4] A is initialized only once during program startup. Afterwards it suffices to clean entries that have actually been used. To facilitate this, the used town IDs are stored in a stack.