Engineering Efficient Error-Correcting Geocoding [*]

Christian Jung
PTV AG
Stumpfstraße 1
76131 Karlsruhe, Germany
christian.jung@ptv.de

Daniel Karch
TU Berlin
Straße des 17. Juni 135
10623 Berlin, Germany
karch@math.tu-berlin.de

Sebastian Knopp
PTV AG
Stumpfstraße 1
76131 Karlsruhe, Germany
Sebastian.Knopp@ptv.de

Dennis Luxen
Karlsruhe Institute of Technology
Kaiserstraße 12
76131 Karlsruhe, Germany
luxen@kit.edu

Peter Sanders
Karlsruhe Institute of Technology
Kaiserstraße 12
76131 Karlsruhe, Germany
sanders@kit.edu

ABSTRACT

We study the problem of resolving a perhaps misspelled address of a location into geographic coordinates of latitude and longitude. Our solution does not require any prefixed rule set and is able to recover even heavily misspelled and fragmentary queries within a few milliseconds.

Categories and Subject Descriptors

H.3.1 [**Content Analysis and Indexing**]; H.3.3 [**Information Search and Retrieval**]

General Terms

Algorithms, Performance, Experimentation

Keywords

Approximate string indexing, Data Structures, Geocoding, Algorithm Engineering

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, e.g [3]. However, with its ubiquitous use in modern web services (e.g., [4, 2]), requirements have become more severe: Since most of these services are free, geocoding servers must handle huge streams of queries at very low cost. Also, users expect instantaneous answers. Inputs will frequently be fragmentary, contain misspelled names or specify combinations of town and street that are inconsistent with the database. We focus on the algorithmic and algorithm engineering aspects of the problem to map information about town and street to a data base entry for the intended street.

[*]Partially supported by DFG grant SA 933/5-1.

2. INDEX DATA STRUCTURE

Our input data are a set T of *towns* and a set S of *streets* that are defined by a name and a reference to a town. We use the term "town" both for a *district* of another place and for an independent place. Also, we use the term *city* for independent places even if they are small. Streets belonging to multiple towns are cut into respective pieces. Districts contain a reference to the city they belong to.

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. 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. We compute 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 lets us efficiently check which combinations of town and street name correspond to actual addresses.

Indexing tokens makes the index easier to use, but leads to a problem. A query of the form "`New Georgia Street`" will return *every* street that matches *any* of the tokens "`New`", "`Georgia`", or "`Street`". As remedy, we apply a concept from information retrieval and text mining [1, 7], called *inverse document frequency* (IDF). Tokens that occur often receive a lower IDF weight than those that appear only infrequently. Tokens with a high weight are more helpful to identify the correct string, because they match fewer strings in the index.

We will use this 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. 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.

We build indices supporting approximate search on the sets of *tokens* appearing in town names and street names respectively. Since dictionaries are much smaller than the full data base, we can afford super-linear space to some extent. Token based indices can easily handle queries that drop part of the town or street name. A recently published approximate string index [5] offers a convenient trade-off between speed and memory consumption. An approximate index 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 (Levenshtein) distance at most d_i.

We first concentrate on the case where a query consists of two strings typed into separate fields for town name and street name. After normalization and tokenization, 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. 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. Otherwise, an empty result is returned.

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 yields a plausible result, we stop.

If we successfully identify 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. If the town 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. The name of y generally does not match anything in the query string and would lead to a low rating.

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 yield hard to understand results.

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 a town with name t, i.e. we compute:

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

3. RATING CANDIDATES

After having 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 unsatisfactory. If all results are unsatisfactory, more extensive search is done (after partially exact matching or periphery search) or, when everything fails, an empty result is returned. If a candidate with rating $\geq \overline{\rho}$ is found, the search returns successfully without further attempts at refining.

Let us recall the different kinds of errors that we want to compensate for: *typing errors*, *missing or redundant tokens*, and *inconsistent pairing* of a street and a town. 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. The rating is computed separately for town and street by the same method and combined by the arithmetic mean afterwards. There is one small asymmetry however that we call *filter by edit distance*: Since there are usually less candidate towns than candidate streets, we prune candidates that are already unsatisfactory because they do not sufficiently well fit the town description from the query.

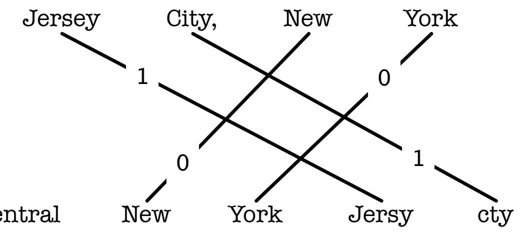

Figure 1: Candidate (top) is matched against query (bottom). Edge labels symbolize edit distances.

To match the town tokens $q \in Q$ from the input to 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 model missing tokens in the query. Similarly, if $|Q| > |C|$ we add $|Q| - |C|$ *dummy* nodes to C. This matching problem can be solved in polynomial time [6]. The considered graphs are tiny and processing is neglectable.

Each token that matches with at most d errors should be awarded some points in a rating. 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. Tokens that could not be matched with at most d errors should not be awarded points and may even be punished. Users are 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 do not match anything in the candidate should be punished higher than candidate tokens that do not 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 define *token similarity* between two tokens q and c as:

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

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 IDF. M is the set of edges (d, c) between tokens and candidate tokens that were matched with edit distance $\leq d$. U is 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 use the rating function:

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

where

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

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

IDF_{avg} is averaged over IDF values of all town tokens. The term $|M_c|\,\text{IDF}_{avg}$ expresses that unmatched queries should have matched somewhere but we have no idea where – so we use an average value. Parameter α is used to adjust how important it is to have similar matches. 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. To give more weight to the matched parts of a query, set $\gamma > 1/2$.

We focused on separate fields for town and street, because multi-field search is easier to program, and one expects that it reduces errors. From a users points of view, however, it is more convenient to enter a query into a single field, with street and town in arbitrary order.

In order to compare multi-field search and single-field search and in order to compare our approach with Internet services, we have 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 try all $2m - 1$ possible ways to split a token sequence of length m and call a multi-field search each time.

4. EXPERIMENTAL EVALUATION

We implemented the system described above in C++ making extensive use of the STL. Experiments were done on a single core of an Intel Core i7 920, running at 2.67 GHz with 12GB RAM on Linux kernel 2.6.27. Source was compiled with GCC 4.3.2.

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 threshold of the approximate dictionaries and pairwise edit distance computations is limited to $d = d_i = 2$ in order to keep space consumption low. 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 $\overline{\rho} = 4/5$.

The commercial data (from 2009) comprises 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, 444 000 street names, and 269 000 street name tokens. A street name consists of 2.5 tokens, while town names have 1.1 tokens on average. The input data takes about 30 MB (uncompressed) space while our index data structures take about 327 MB.

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 address is composed of randomly chosen town and street names such that combinations which accidentally occur in the database are rejected. We want to return correct results for relevant and no result for irrelevant address queries. To generate a random query, it would be easy to just insert, delete or substitute random characters in an existing address. Those errors, 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 common and we distinguish *swapped characters*, *missing characters*, and *superfluous or wrong characters*, *doubled characters*, where there should be a single character, or *a single character*, where there should be two of the same. The SOUNDEX algorithm identifies classes of characters such that different characters from the same class differ only slightly in their pronunciation. Depending on the respective language there are several sources of error that are phonetic. Two consecutive vowels that occur in the same syllable are called a *diphthong*. In German, several different *diphthongs* sound the same or similar.

To get k errors, we introduce $\lceil k/2 \rceil$ errors into the street string and $\lfloor k/2 \rfloor$ errors into the town string. A random error class is picked, then a random token, and then a random position. All distributions are uniform. We classify the results by: *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 do not find a result, and *Incorrectly Identified (II):* A relevant address that returns an incorrect result, i.e. another relevant address. Table 1 gives match rates on 1 000 relevant and 100 irrelevant addresses, along with query times. Multi-field search works extremely well. Only at five errors, we see a sharp increase of false negative results. At this point, the approximate street index will 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, because a smaller number of candidates is checked both in the approximate index and when rating candidates.

Single-field search for relevant addresses works almost as well as multi-field search. The notable difference is that a small fraction of false negative results mutates into incorrectly identified results. Our implementation seems to be too aggressive here. It is paradoxical that we get a larger number of output errors when there are no spelling errors in the input. The reason is simple. Without spelling errors we obtain higher ratings for the generated candidates and it becomes more likely that the result is accepted. This indicates that the result quality could be improved by increasing the acceptance threshold. Single-field query times are an order of magnitude larger, since the implementation naively factors a single-field search into several multi-field searches.

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

90% of all multi-field queries finish in less than 5 ms. The maximum is 106 ms. For our server scenario, we want very low *average* query time to achieve high throughput and low cost. Occasional slow queries are no problem as long as they do not lead to noticeable delays for the user. 100 ms is in the order of the delays users experience due to network latencies. 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 compare against existing geocoders and take the first 100 relevant queries from the above query set and run them against the available APIs of Bing, Yahoo and Google Maps. The results of Yahoo are very sensitive to the ordering of street and town. Therefore we fed both orderings to the API. It should also be noted that the subjective performance of Google with interactive use on the web is much better. Google works very well for undistorted inputs, but already for a single error, the recognition rate drops to 54 % and collapses for $d \geq 2$. Bing already has significant deficits at $d = 2$, but fails for distorted inputs. The number of failed answers is an order of magnitude worse than for our system.

We also experimented with real-world input. 1 383 queries have been provided by users of an existing geocoder running at PTV AG. We checked the correctness of our results manually. Our system outperforms the Google, Yahoo! and Bing API. Google is similarly good for (partially) exact queries but looses ground for errorneous inputs. A difference to the random inputs is that now Google consistently outperforms Bing – also for the inputs with errors. Yahoo is sometimes better than Bing.

We use several heuristics to make sure that the number of candidates stays small and that we do not have to perform too many edit distance computations. Tests show each of them affects the query time. The techniques are: *Filter Incompatible Candidates (FIC):* We keep only those town and street candidates that are geographically compatible. *Filter by Edit Distance (FED):* As described in Section 3 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. candidate "`New Georgia Street`" will be represented by "`Georgia`", because the other tokens occur so frequently in the dictionary.

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

Table 2: Effect of ILT, FIC and FED.

Table 2 reports that disabling ILT destroys performance. FED is always enabled since it only has an impact together with FIC. Without ILT, *any* query that contains the token "`street`" will return *all* candidates that contain this token. Query time drops by a factor of almost 100 with ILT enabled. FIC gives another boost of factor 3. None of these features has a noteworthy effect on the memory requirements of the index, therefore all of them are enabled by default.

5. FUTURE WORK

Developing further techniques especially suited for single-field search has a high priority. The experiments used commercial German map data, and it should be easy to adapt to other Western countries. 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.

Points of interest can be included by seeing them as streets, towns or house numbers in a separate index depending on the context. Finding street intersections is easy once we have geocoded the two intersecting streets. We did not cover the problem of ambiguities in detail, e.g. "Berlin, Germany" vs. "Berlin, Ohio" vs. "Berlin, Connecticut" or with omitted information, e.g. "Times Square" without "New York City". This problem can be tackled by sorting the town tokens to which the street tokens point to by some *importance*.

6. REFERENCES

[1] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval.* Addison-Wesley, 1999.

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

[3] 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.

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

[5] D. Karch, D. Luxen, and P. Sanders. Improved fast similarity search in dictionaries. In *Proc. of SPIRE'10*, LNCS. Springer, 2010.

[6] J. Munkres. Algorithms for the assignment and transportation problems. *Journ. SIAM*, 5(1), 1957.

[7] 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.