An efficient algorithm for fault-tolerant geocoding

Diploma thesis
of

Daniel Karch

at the
Institute for Theoretical Computer Science, Algorithmics II

Referee: Prof. Dr. Peter Sanders
Advisors: Dipl.-Inform. Dennis Luxen
Dipl.-Inform. Sebastian Knopp (PTV AG)
Dipl.-Phys. Christian Jung (PTV AG)

Acknowledgements

First of all, I would like to thank my advisors: Dennis Luxen at the university and, at PTV AG, Sebastian Knopp and Christian Jung. They provided invaluable suggestions and showed great interest for my ideas. During my time at PTV I enjoyed working in the amicable working atmosphere and I am glad that I chose to write my thesis there. I am also grateful to my advisor Prof. Dr. Peter Sanders, who always has an open ear for his students.

Particular thanks and gratitude go to my parents, who are great and have always supported me, and still do.

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Karlsruhe, 07. 10. 2010

Daniel Karch

Inhaltsverzeichnis

1. Introduction & Previous Work **3**
1.1. Approximate Dictionary Matching 3
1.2. Geocoding 5

2. The Index **8**
2.1. A Graph Theoretical Point of View 9
2.2. Construction 10
2.3. The Query 10
2.3.1. Properties of Residual Strings 12
2.4. Reducing the number of residual strings 15
2.5. Indexing Tokens 17
2.6. Implementation Details 18
2.6.1. Generating Residual Strings 20
2.6.2. Edit Distance 21
2.6.3. The Tokenizer 22
2.7. Analysis 24
2.7.1. Memory Consumption 24
2.7.2. Query Time 25
2.8. Experiments 25
2.8.1. Implementation Note 28

3. Geocoding **30**
3.1. Dropping incompatible candidates 32
3.2. Tie-Breaking 33
3.2.1. The Neighborhood Graph 34
3.3. Searching in the Neighborhood Graph 38
3.4. Finding a Postal Address 39
3.5. Pseudocode & Analysis 42
3.5.1. Analysis 42
3.6. Searching in a Single Search Field 44
3.7. Rating Address Candidates 46
3.7.1. Matching the Query to a Candidate 46
3.7.2. The Rating Heuristic 48
3.7.3. Ignoring Light Candidates 51
3.8. Experiments 52
3.8.1. Random Queries 52
3.8.2. Real Queries 54
3.8.3. Parameters That Affect the Query Time 57
3.8.4. Single Field Search 59

3.8.5. Searching in the Neighborhood Graph 60

4. Open Problems 61

5. Conclusion & Future Work 62

A. Sample Queries 64

A.1. Randomly Distorted Queries 64

A.2. Logged Queries 65

Deutsche Zusammenfassung

Die vorliegende Arbeit befasst sich mit dem Problem der fehlertoleranten Georeferenzierung. Mit *Georeferenzierung* oder *Verortung* (englisch: Geocoding) bezeichnet man die Zuweisung raumbezogener Daten zu einem Datensatz. Anwendung findet dies beispielsweise in Online-Routenplanern (z.B. Google Maps, ADAC Maps etc.), Navigationssystemen oder zunehmend auch in Mobiltelefonen, aber auch im Gesundheitswesen und im Rahmen polizeilicher Ermittlungen.

Im folgenden Text wird das Problem behandelt, textliche Beschreibungen von Postadressen einem tatsächlichen Adressdatum zuzuordnen, d.h. einer Straße und einem Ort. Schwierigkeiten ergeben sich durch Eingabefehler, wie z.B. Tippfehler. Wir präsentieren Algorithmen und Datenstrukturen, die sich solcher Probleme annehmen und die automatische Korrektur von Fehlern vornehmen.

Der Hauptteil der Arbeit gliedert sich in zwei Teile. Kapitel 2 auf Seite 8 behandelt das Problem des *approximate dictionary matching*, zu deutsch etwa *unscharfe Suche in Wörterbuchern*. Wir beschreiben einen Index, der solche unscharfen Anfragen auf Wörterbüchern in sehr kurzer Zeit beantwortet. Zu einem vorgegebenen Wörterbuch und einer gegebenen Anfrage werden alle Zeichenketten bestimmt, die starke Ähnlichkeit mit der Anfrage aufweisen. Als Maß für die Ähnlichkeit wird hierbei LEVENSHTEIN-Abstand verwendet. Die in diesem Teil präsentierten Algorithmen und Datenstrukturen sind weitgehend unabhängig vom Problem der Georeferenzierung, der Index bietet sich also auch für andere Anwendungen an. Das Hauptaugenmerk lag darauf, sehr geringe Anfragezeiten zu erzielen, was auch gelang, wie in Abschnitt 2.8 auf Seite 25 experimentell gezeigt wird.

Kapitel 3 beschreibt unseren Ansatz zur Georeferenzierung von Postaddressen. Um möglichst schnell zu relevanten Ergebnissen zu kommen, verwenden wir Techniken, um den Suchraum möglichst früh so zu beschneiden, dass irrelevante Adressdaten nicht mehr behandelt werden müssen. Wir erreichen dies einerseits durch Anwendung bekannter Verfahren aus dem Gebiet des *Information Retrieval*, wie z.B. durch die Bestimmung der inversen Dokumenthäufigkeit (IDF) von Suchbegriffen. Außerdem nutzen wir geografische Informationen aus, die den Adressdaten zugrundeliegen, sowohl um die Suche zu beschleunigen als auch um die Qualität der Ergebnisse zu verbessern. Zur Bewertung der Qualität der Ergebnisse wird eine einfache Heuristik präsentiert, die ebenfalls auf Levenshtein-Abstand und IDF basiert.

Die Leistungsfähigkeit der Algorithmen wird in Abschnitt 3.8 auf Seite 52 durch empirische Analyse belegt. Die Testdaten umfassen etwa 80500 deutsche Städte sowie etwa 444000 Straßen. Experimente wurden sowohl mit zufällig erzeugten Eingaben durchgeführt als auch mit tatsächlichen Eingaben, wie sie von Benutzern eines existierenden Georeferenzierungssystems aufgezeichnet wurden.

Zum Abschluss werden Vorschläge für künftige Untersuchungen gegeben. Insbesondere die Erweiterung auf andere, weniger restriktive Adressschemata (z.B. `"Schule in der Innenstadt"` oder `"Kreuzung Poststraße, am Rathaus"`) könnten die Benutzerfreundlichkeit weiter verbessern, und es wäre interessant zu untersuchen, ob solche Anfragen unterstützt werden könnten, ohne die hier vorgestellten Datenstrukturen und Algorithmen signi-

fikant abändern zu müssen bzw. ohne zu viel Laufzeit einzubüßen. Ein weiterer lohnenswerter Schritt wäre die Anpassung der Algorithmen an die Möglichkeiten moderner Hardware, wobei sich natürlich die Frage der Parallelisierbarkeit stellt.

Diese Arbeit wurde als externe Diplomarbeit bei der PTV AG [1] in Karlsruhe verfasst.

1. Introduction & Previous Work

This work describes a fault-tolerant algorithm for geocoding postal addresses. It consists of two main parts: The description of a fault-tolerant dictionary index and its application in the domain of geocoding. In Section 1.1 we describe the *approximate dictionary matching* problem and previous approaches. Section 1.2 on page 5 gives an overview over the process of *geocoding*: What it is, why it is important, and what is the current state of the art. While fault-tolerant geocoding usually relies on a fault-tolerant (or *fuzzy*) index, those domains are largely independent, i.e., our fuzzy index is not only useful in the domain of geocoding, and our geocoder would also work with any other fuzzy index. Therefore we will describe the index data structure and our geocoder separately. Design decisions that were taken because of particular properties of the index / the geocoder are mentioned in the text.

In Section 2 on page 8 we present our approach to the approximate dictionary problem. We describe the construction of the data structure and the lookup procedure and conclude with an experimental Analysis. Section 3 on page 30 explains how we use the fuzzy index to build a fault-tolerant address search. Again, we conclude with an experimental evaluation of our algorithm. Furthermore we develop a rating heuristic, described in Section 3.7 on page 46, that we will use to rank results according to their quality. Finally, we give our conclusions and ideas for future work in Section 5 on page 62.

1.1. Approximate Dictionary Matching

The *approximate dictionary matching* problem is to find an approximate occurence of a string in a potentially large set of strings called a *dictionary*. This problem arises in many applications, online search engines are a prominent example. The problem is closely related to the *approximate string matching* problem, where the task is to find approximate matches of a short pattern in a potentially much longer string. Most of the techniques mentioned in the following apply to both problems, hence we will not distinguish between related work that centers on dictionary matching and work that centers on string matching.

If w and w' are words over some alphabet Σ, they *match approximately* if and only if their distance is small in respect to some distance metric. Popular metrics in this context include Hamming distance or Levenshtein distance [23], which is often called *edit distance*. In this work we will focus on Levenshtein distance. Two strings w and w' are said to have Levenshtein distance d if w can be transformed into w' by at most d single character insertions, substitutions, or deletions. The Levenshtein distance fulfills the requirements for a metric (non-negativity, identity of indiscernibles, symmetry, triangle inequality) and defines a metric space on the the set of all strings.

The approximate dictionary matching problem can be tackled as an offline problem, where it is allowed to apply preprocessing on the dictionary to speed up queries, or as an online problem, where it is not possible to process the dictionary beforehand. To support sublinear query times, we focus on the offline problem only, which is also called the *indexed* version of the problem.

Many successful methods for the approximate dictionary and string matching problems are based on *filters* [18, 34]. For most queries, only very small parts of the dictionary come into consideration as possible matches. A filter is an algorithm which is used to seperate these parts from those that cannot possibly result in an approximate match. Note that it is not necessary that each mismatch be dismissed by the filter. The remaining candidates then have to be verified by a non-indexed approximate string matching algorithm. There is usually a trade-off between the accuracy of the filter (i.e., how many mismatches it catches) and the time that is needed to evaluate the filter criterion. A filter is called *lossless* if it never discards an approximate occurence of a query. The filter we present here is lossless.

We present an index data structure that allows fast lookups of words contained in a dictionary. It uses a filter that is based on a technique called *FastSS* [40] and centers on the idea of *deletion neighborhoods* which were already described in [28]. It is therefore not unlike the neighborhood generation algorithm [30], but uses only deletions.

Some filtering algorithms split the input words into smaller parts to reduce the complexity of the problem. In [18] those are called *factor filters*. We use a simple splitting rule in our algorithm to reduce the size of the index, as described in [17] and in Section 2.4 on page 15.

A similar approach splits words into *n-grams* [32, 43], i.e. (potentially overlapping) substrings of length n. E.g., the word `"keyboard"` breaks down to the 3-grams `"key"`, `"boa"`, and `"ard"`. Given the length of a query string q and the edit distance to a word w one can derive a lower bound on how many n-grams q and w must share.

More sophisticated methods use *suffix trees* [45]. A suffix tree is a data structure that represents all suffixes of a text in linear space and can be constructed in linear time [42]. Originally suffix trees have been used for exact searches, but they have been successfully adapted to approximate string searching [41]. Although the suffix tree has space $\mathcal{O}(n)$ for a text with n characters, the constant factors are quite high, which is why it is often substituted by a similar but simpler data structure, the *suffix array* [25].

Other methods exploit the fact that edit distance defines a metric space on the set of all words in an alphabet Σ [8, 37]. One can then use properties of the metric space, such as the triangle inequality, to ease distance computations. In [6] the authors divide these approaches into two classes:

- *Clustering algorithms*, which divide the search space into a set of clusters, such that objects inside of a cluster share some representative information, and
- *Pivot-based algorithms*, where an element x is chosen from the set of objects, and each element stores its distance to x.

An example for a data structure that falls into both classes is the *BK-Tree* [4]. An element is selected as the root node and the subtrees are identified by the distance of their elements to the root node. The i-th subtree consists of all elements that have distance i from the root. This technique is then applied recursively to the subtrees, until the number of elements in a subtree falls under some threshold. Again, the triangle inequality is used to branch into or cut subtrees. A candidate set of possible matches is built by the union of all leaves that are

reached by the tree traversal. A rather weak result is that BK-Trees and its refinements need $\mathcal{O}(n^{\alpha}), 0 < \alpha < 1$, comparisons and node traversals on average [37]. See Chávez et al. [6] for a survey. An experimental evaluation of BK-Trees and several variants [29] reports on the size of the search space that is visited depending on the allowed error distance. Previous experiments on BK-Trees were done on a set of 100 000 English words. The experiments report on a nearly linear growth of the visited search space going up from 5% for edit distance 0 to slightly more than 40% for a distance of 4.

To speed-up edit distance computation itself, research focused on simple and practical bit-vector algorithms that compute a bit representation of the current state-set of the k-difference automaton for the query [46]. Words of length n with k and fewer differences can be matched against a query of size m in either $\mathcal{O}(nmk/w)$ or $\mathcal{O}(nm \log \sigma / w)$ time where w is the word size of the machine, and σ is the size of the pattern alphabet. These algorithms have been further improved to yield a bound of $\mathcal{O}(nm/w)$ and for arbitrarily large m there exists an algorithm that runs in expected time $\mathcal{O}(kn/w)$ [31]. This is much faster than the classic dynamic programming approach, which needs $\mathcal{O}(nm)$ time. In our application we are only interested in the distance if it is smaller than a threshold k, and it then suffices to compute a diagonal stripe of width $2k+1$ in the matrix, yielding an algorithm that runs in $\mathcal{O}(kl)$ time, where l is the length of the shorter string [14].

1.2. Geocoding

The term *geocoding* decribes the act of turning a textual description of a location, such as a postal address, into an absolute geographic reference. It forms a fundamental component of spatial analysis and finds application in a wide variety of contexts, such as crime analysis [35, 36], public health and epidemiological research [20, 21], or route guidance systems.

The increasing availability of reference datasets and freely accessible mapping services such as Google Maps or Microsoft Bing Maps has turned geocoding to a ubiquitous service. In contrast, geocoded data used to cost $4.50 per 1000 records in the mid-eighties [19] and didn't nearly provide the spatial accuracy of today's services. While the use of geographic information systems was previously limited to professionals only that were aware of the difficulties and limitations of the geocoding process [38], today's freely available online services don't require much knowledge from the user.

Goldberg et. al [11] identify four fundamental components in the process of geocoding: the *input*, *output*, *processing algorithm*, and *reference dataset*. The input is the decribed entity the user wishes to have geographically referenced, and said description must contain attributes that have previously been assigned to some datum in the reference dataset that represents the geographic reference. The output format can reach from simple geographic codes to complex two- or threedimensional geospatial entities (lines, polygons, polytopes etc.). The processing algorithms have moved from simple feature assignment to complex interpolation algorithms using diverse data sources. The accuracy and quality of the algorithms depend on the underlying reference dataset.

Most geocoding services geocode postal addresses only. Postal addresses are how people locate and navigate themselves [10], hence it is sufficient for many applications to restrict the geocoder to this case. Although even cheap cellular phones will have access to positioning systems like GPS or GALILEO in the near future, postal addresses will likely remain the prelevant form of location data in many contexts, due to their inherent redundancy and because they are more easily remembered than geocoordinates.

While postal addresses are the most common data to be geocoded, modern geocoders have attempted to process many different kinds of locational descriptions, such as points of interests, street intersections, zip codes, or even freeform textual descriptions of locations (as in "Italian restaurant near main station") [11, 39].

The output of a geocoder is usually a geometric entity such as a point (e.g. for an address), a line (e.g. for a street), or a polygon (e.g. for a town). This often requires the use of interpolation algorithms when the output geography is not part of the reference data [9, 11]. For example one may not want to store the precise coordinate of every street number in the reference dataset to save space, or one could calculate the coordinate of a street intersection on the fly.

In the process of geocoding there are several possible sources of error – geocoding is an inherently uncertain process. Quantifying the error is therefore a difficult task, some of the questions that arise are:

- Match rate: What percentage of the processed queries could be matched to a datum in the reference dataset?
- Correctness: When can we classify a match as a "correct" match for a given query?
- Positional accuracy: For a correct match, how accurate is the returned coordinate (or line, or polygon etc.). This is difficult especially for interpolated data where ground truth is not available.
- Quality of the input: Who is responsible for the input? Was does the user know about the deficiencies of the geocoder? Can the error be attributed to the geocoder or was the input "too bad"?

In [38], the authors compare and evaluate several online geocoding services according to the metrics *match rate*, *positional accuracy*, and *similarity*. Similarity in this case refers to the pairwise distance of matches among the five compared services. The tests were performed using address data from the Environmental Protection Agency.

To increase match rates, modern geocoders try to correct certain classes of errors, such as spelling errors. "Fuzzy" techniques have been developed that use word stemming, phonetic algorithms such as Soundex, substitution tables or distance metrics such as Levensthein or Hamming distance. Recent works concentrate on the less restricting Levenshtein Distance [5,16,39]. Phonetic methods often suffer from their restriction to certain languages; it would be difficult to apply the Soundex algorithm on a dataset that contains both english and french addresses.

Correcting typing errors and the like to match some input query against a reference datum is sometimes called *data cleaning*. In recent years methods have been developed that take advantage of the spatial information underlying the data. In [16, 39], a method is described where each attribute (like for instance "school" or "station") is associated with the spatial union of all entities that share this attribute. Match candidates are then formed by the spatial intersection of these unions (e.g. think of a query like "primary school near main station"). The authors claim that their system is capable of supporting regions with widely varying address formats, without region-specific customization or training. However, the average lookup time of 170ms on a single metropolitan area leaves room for improvement, and it is not clear how well their system scales to larger areas.

2. The Index

The indexing method we present here is based on the observation that two words that are similar with respect to edit distance must share a common subsequence of a certain length. We will prove that formally in the following lemma, but first we have to introduce the notation we will use. The *edit distance* between two words u and v over an alphabet Σ is the minimal number of insertions, deletions or substitutions that we have to perform to transform u into v. Note that we insert, delete or substitute only one character at a time. We define the functions

$$\begin{aligned} ins &: \mathbb{N} \times \Sigma \times \Sigma^* \to \Sigma^* \\ del &: \mathbb{N} \times \Sigma^* \to \Sigma^* \\ sub &: \mathbb{N} \times \Sigma \times \Sigma^* \to \Sigma^* \end{aligned}$$

such that

- $ins(n, c, w)$ inserts character c into word w after the nth character in w,
- $del(n, w)$ deletes the nth character in w, and
- $sub(n, c, w)$ substitutes the nth character in w with c.

For any two strings $u, v \in \Sigma^*$ with $ed(u, v) = d$, there must be a sequence of operations $op_1, \ldots, op_d \in \{ins, del, sub\}$, such that $(op_d \circ \cdots \circ op_1)(u) = v$.

$$\begin{aligned} u &:= \texttt{SURVEYS} \\ sub(4, \texttt{G}, u) &= \texttt{SURGEYS} =: u^{(1)} \\ del(7, u^{(1)}) &= \texttt{SURGEY} =: u^{(2)} \\ ins(5, \texttt{R}, u^{(2)}) &= \texttt{SURGERY} = v \end{aligned}$$

$$\Rightarrow ins(5, \texttt{R}, del(7, sub(4, \texttt{G}, u))) = v$$

Example 1: A minimal sequence of operations to transform SURVEYS into SURGERY.

Lemma 1. *Let $u, v \in \Sigma^*$ be words with edit distance $ed(u, v) \leq d$. Then there must exist a string $r \in \Sigma^*$ that is a subsequence of both u and v and has length $|r| \geq \max(|u|, |v|) - d$. We call r a* residual string *and u and v* original strings.

Proof. We give an algorithm to obtain such a common subsequence r:
Let $(op_i \circ \cdots \circ op_1)(u) = u^{(i)}$, i.e. $u^{(0)} = u$ and $u^{(d)} = v$.
For $i = 1 \ldots d$ repeat:

- If $op_i = del(u^{(i-1)}, n)$, delete the character in u.
- If $op_i = ins(u^{(i-1)}, c, n)$, there must be an occurence of c in v that is "responsible" for this insertion. Delete the occurence of c in v.
- If $op_i = sub(u^{(i-1)}, c, n)$, delete the character that would be substituted in u as well as the character that would be matched in v.

In each step of the algorithm the distance between the compared words decreases by one and the length of both words decreases by at most one. Hence, after d steps we arrive at a common subsequence of length at least $\max(|u|, |v|) - d$. □

Furthermore, we call the set of all residual strings of a string s and edit distance d its *deletion neighborhood* $N_d(s)$. The deletion neighborhood for a given string s and edit distance d can be computed by applying all possible combinations of d deletions on s. There are $\binom{|s|}{d}$ such combinations.

$d = 0$	$d = 1$	$d = 2$
string	_tring	__ring
	s_ring	_t_ing
	st_ing	_tr_ng
	...	...
	strin_	stri__

Table 1: The deletion neighborhood for $s =$ "string", $d \leq 2$. The underscores indicate the positions where deletions were performed.

We can use this observation to construct an error-tolerant index that can be queried very efficiently. Let $D = \{s_1, \ldots, s_n\}, s_i \in \Sigma^*$ be a dictionary of words over the alphabet Σ. Given a string $w \in \Sigma^*$ and a constant $d \in \mathbb{N}$, our goal is to quickly return all words from D whose edit distance to w is at most d. From the above lemma we know that any word s that satisfies this condition must share a residual string of a certain length with w. We call $s \in D$ a *candidate* for w if it has a residual string in common with w. Hence, it suffices to check the edit distance of w to these candidates and our task is now to develop a compact data structure that enables us to quickly find the set of candidates for a given query string.

2.1. A Graph Theoretical Point of View

We want to look at the problem from a graph theoretical standpoint.

Let

$$N(s) := \bigcup_{k=0}^{d} N_k(s)$$

and

$$\begin{aligned} R &:= \bigcup_{s \in D} N(s) \\ &= \{r \in \Sigma^* \mid \exists s \in D : r \in N(s)\} \end{aligned}$$

be the set of all residual strings for the dictionary D. Then the graph

$$\begin{aligned} G &= (R \cup D, E) \\ E &= \bigcup_{s \in D} N(s) \times \{s\} \end{aligned}$$

is a directed bipartite graph. We can assume without loss of generality that R and D are disjoint. If this isn't the case, it is trivial to alter the definition of R such that they are in fact disjoint.

Finding the set of candidates for a query string w now equates to finding the nodes $s \in D$ that have an incoming edge coming from a node $r \in (R \cap N(w))$, i.e. r is a residual string of w. See Fig. 1 on the facing page for an illustration.

2.2. Construction

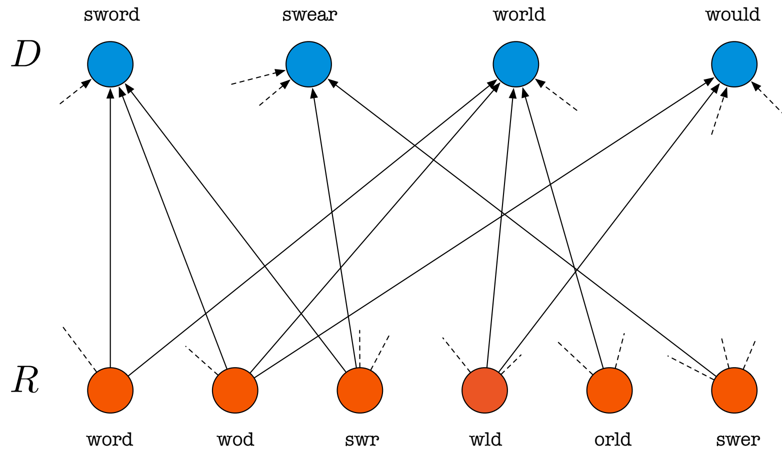
Every edge in the graph G is directed from a residual string to an original string. To construct the index, we will traverse a list of unique strings, i.e. no string will occur twice in this list. For each string, we generate the deletion neighborhood. Each residual string gets its own adjacency list, and when a string s is added to the index, we will append a pointer to s to the adjacency lists of every residual string in $N(s)$ (see Fig. 2 on page 12 for an illustration). Pointers to the adjacency lists are stored in an array of size $|R|$. We use a hash function to address them. For now let us assume that we can use a perfect minimal hash function, i.e. a bijective function $h : R \to \{0, \ldots, |R| - 1\}$.

During construction, it is convenient to be able to insert new edges in constant time, but once the graph is complete, we want to switch to a more efficient representation. We use a *forward star* representation, which is very compact and makes it possible to iterate quickly over the outgoing edges of a node.

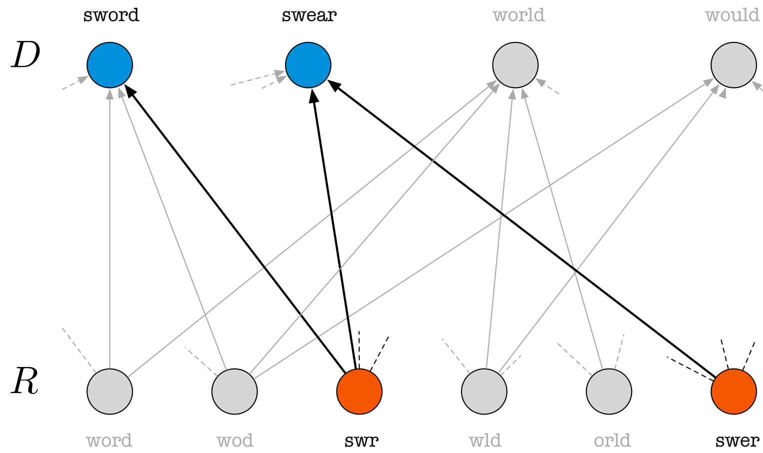
Simply put, it consists of two big arrays: `edges` of size $|E|$ and `first_edge` of size $|R| + 1$. We copy the adjacency lists created during construction to the array `edges`, one after another. The array of pointers is now called `first_edge`, and instead of pointers to adjacency lists, we store for each node the position of the first edge leaving it. For a node v, its edges are stored in `edges`, starting at position `first_edge[h(v)]` up to and excluding `first_edge[h(v)+1]`. See Fig. 3 on page 13 for an illustration.

2.3. The Query

The query is straightforward. Suppose that the graph as described above has been computed in a preprocessing phase. We want to find all dictionary entries with edit distance at most



(a) A cutout from the Graph for a dictionary containing the words "sword", "world", "swear" and "would".



(b) The relevant subgraph for the query $w =$ "sewer".

Figure 1: The dictionary and the residual strings modeled as a graph.

d from a word $w \in \Sigma^*$. We know that any word that matches w with at most d errors must have length l with $|w| - d \leq l \leq |w| + d$. From lemma 1 on page 8 we know that there must be a residual string of length $\max\{l, |w|\} - d$. Hence, the residual strings must be of length l' with $|w| - d \leq l' \leq |w|$. Therefore we must generate all residual strings of w with $0, \ldots, d$ deletions.

To find candidates for w, we simply compute its deletion neighborhood $N(w)$ for up to d deletions and query the index with each of the residual strings, as described in Algorithms 1 on the following page and 2 on page 14. If v is a candidate for a query w, that means that they share a residual string r that was obtained by deleting at most d characters from v as well as from w, therefore $|v| - |r| \leq d$ and $|w| - |r| \leq d$. According to Lemma 2 on the following page we know that $\mathrm{ed}(v, w) \leq |v| - |r| + |w| - |r| \leq 2d$. We still have to verify which candidates have distance d or less (see Algorithm 3 on page 14)

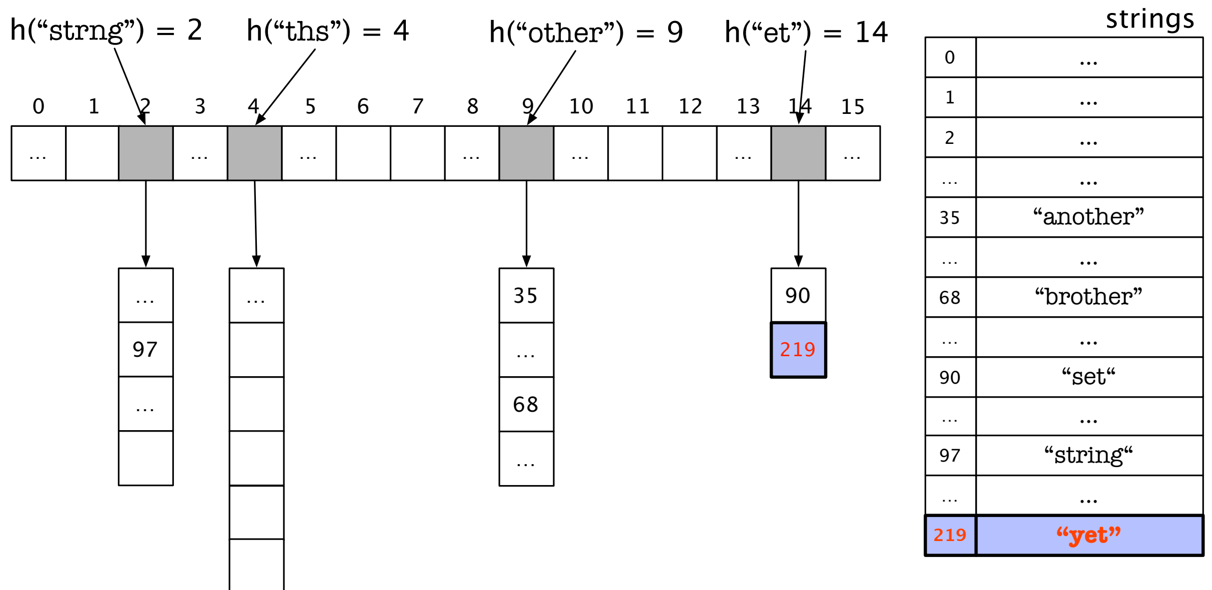


Figure 2: A sketch of the data structure during construction, right after the string "yet" has been inserted.

Algorithm 1: FindCandidatesForString

Input : A string w.

Output: The set of all strings in D that share a residual string with w.

1 $candidates \leftarrow \emptyset$

2 $N_w \leftarrow \texttt{DeletionNeighborhood}(w)$

3 **foreach** $r \in N_w$ **do**

4 $\quad C_r \leftarrow \texttt{FindCandidatesForResidualString}(r)$

5 $\quad candidates \leftarrow candidates \cup C_r$

6 **return** $candidates$

2.3.1. Properties of Residual Strings

It is easy to see that the existence of a common residual string of length $\max(|u|, |v|) - d$ does *not* imply that $\mathrm{ed}(u, v) \leq d$. E.g. the strings "yx" and "xz" have a common residual string of length $\max(2, 2) - 1 = 1$, but edit distance $\mathrm{ed}(yx, xz) = 2 > 1$. But we can still bound the edit distance:

Lemma 2. *If r is a common residual string of u and v,* $\mathrm{ed}(u, v) \leq |u| + |v| - 2|r|$.

Proof. Starting from u, we can arrive at r by deleting $|u| - |r|$ characters from u. We then obtain v by inserting $|v| - |r|$ characters into r. □

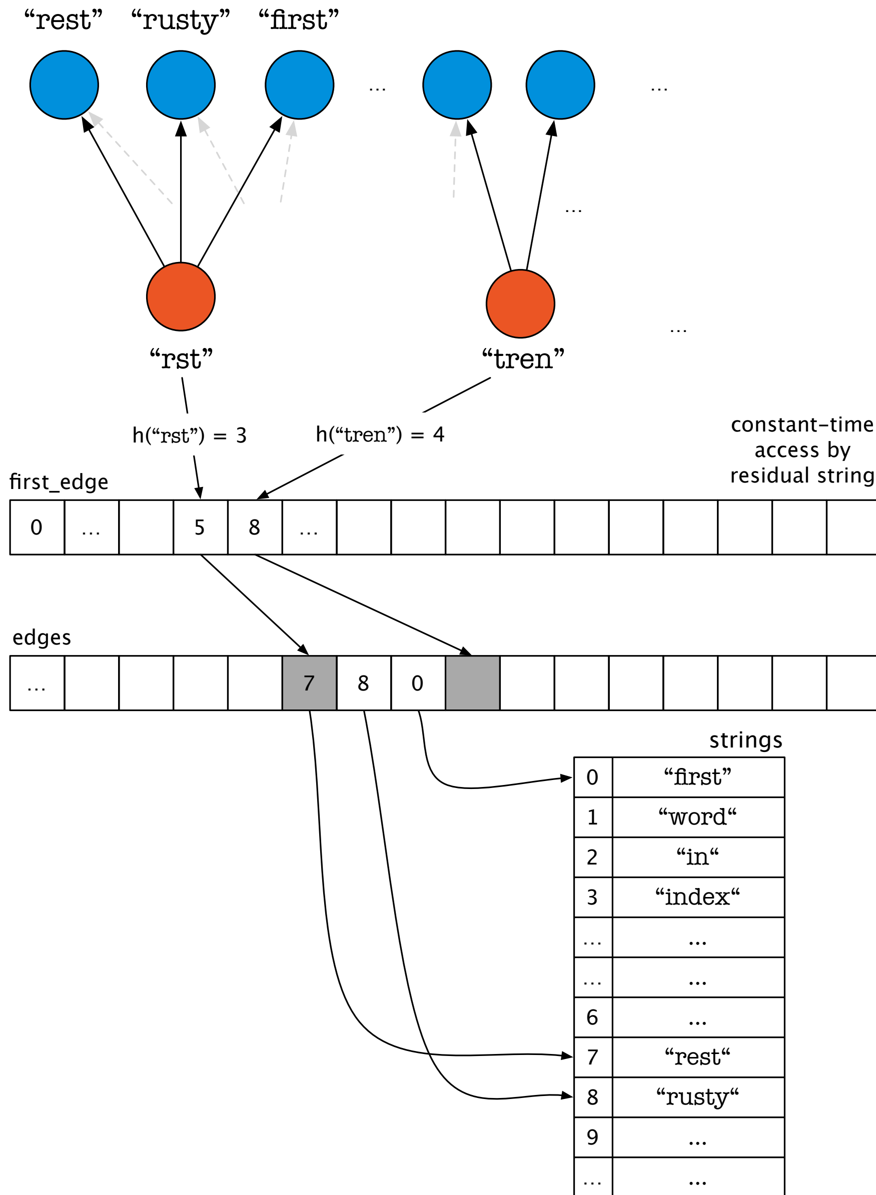


Figure 3: The *forward-star*-representation of the graph. To save space, we store each string only once. In edges we only maintain pointers.

Since we construct the index for up to d deletions, we know that a common residual string r of u and v implies that $\mathrm{ed}(u,v) \leq (|u|-|r|)+(|v|-|r|) \leq d+d = 2d$. In a sense the filter itself "solves the approximate dictionary problem approximately" before the verification phase: it returns all approximate matches with distance d and no matches with distance greater than $2d$.

Algorithm 2: FindCandidatesForResidualString

Input : A residual string r.
Output: The set of all strings in D that have r as a residual string.

```
1 candidates ← ∅
2 first ← first_edge[h(r)]
3 last ← first_edge[h(r)+1]−1
4 for i ← first to last do
5     candidates ← candidates ∪ {edges[i]}
6 return candidates
```

Algorithm 3: FindSimilarStrings

Input : A string w and a threshold d.
Output: The set of all strings in D that have edit distance at most d to w.

```
1 candidates ← FindCandidatesForString(w)
2 result ← ∅
3 foreach s ∈ candidates do
4     if EditDistance(w,s) ≤ d then
5         result ← result ∪ {s}
6 return result
```

We really have to consider each residual string of a word. In the case where $\mathrm{ed}(u,v) = d$, there may be only one common residual string, take e.g. the strings "xbcde" and "abcdx" and $d = 2$. The only common residual string with up to 2 deletions is "bcd".

The following lemma formalizes how many residual strings two strings share at the least:

Lemma 3. *Let* $N(w) := \bigcup_{k=0}^{d} N_k(w)$ *and* $N(u) := \bigcup_{k=0}^{d} N_k(u)$ *be the residual neighborhoods of* u *and* w *and* $\mathrm{ed}(u,w) = j \leq d$. *If* $|N(w)| = \sum_{k=0}^{d} \binom{|w|}{k}$ *and* $|N(u)| = \sum_{k=0}^{d} \binom{|u|}{k}$, *then*

$$|N(w) \cap N(u)| \geq \mu_j(u,w) := \sum_{k=0}^{d-j} \binom{\max\{|w|,|u|\} - j}{k}$$

Proof. If $\mathrm{ed}(u,w) = j$, there exists a common residual string v of length $\max\{|u|,|w|\} - j$. This string can be obtained from u and w by j deletions. Any residual string of v with at most $d - j$ deletions is also a residual string of u and w with at most d deletions. □

Corollary 4. *If* $|N(w) \cap N(u)| < \mu_j(u,w)$*, then* $\mathrm{ed}(u,w) > j$*.*

We can take advantage of this fact during the query. When a residual string r is hit during the query with string u, every string that has a pointer from this residual string gets marked as a candidate. But for a candidate w to have edit distance j or smaller to u, it must be hit $\mu_j(u,w)$ times. So, instead of marking every string that gets hit during a query, we mark only those that get hit often enough. Note that this is different from the previous behaviour only when searching for words with less than d errors, as $\mu_d(u,v) = 1$.

2.4. Reducing the number of residual strings

Since there are $\binom{|s|}{d}$ residual strings in $N_d(s)$, their number grows rapidly with the length of s. For instance, a word s of length $|s| = 30$ and edit distance 3 has up to $\binom{30}{3} + \binom{30}{2} + \binom{30}{1} + \binom{30}{0} = 4526$ residual strings. In comparison, a word of length 10 has only 176 residual strings.

Fortunately we can use the greater length to our advantage. It is intuitively clear that increasing the length of a string u under a fixed number of errors also leads to longer substrings of u with few errors.

Lemma 5. *Let* $u, v \in \Sigma^*$ *be words with distance* $d := \mathrm{ed}(u,v)$ *and* $u := u_1u_2$ *any partitioning of* u*. Then there must exist a partitioning of* $w = w_1w_2$ *such that* $\mathrm{ed}(u_1,w_1) \leq \lfloor d/2 \rfloor$ *or* $\mathrm{ed}(u_2,w_2) \leq \lfloor d/2 \rfloor$*.*

Proof. We can transform u into w by applying d edit operations on u:
Suppose that $op_1, \ldots, op_d$ are those operations, ordered "from left to right". We first apply the operations $op_1, \ldots, op_i$ that operate on u_1. Notice that u_2 remains unchanged during these operations. If $i \leq \lfloor d/2 \rfloor$, we're done. If $i > \lfloor d/2 \rfloor$, there remain only $d - i < \lfloor d/2 \rfloor$ operations to transform u_2. We set $w_1 := (op_i \circ \cdots \circ op_1)(u_1)$ and $w_2 := (op_{i+1} \circ \cdots \circ op_d)(u_2)$. Then it is clear that $\mathrm{ed}(u_2,w_2) \leq \lfloor d/2 \rfloor$. □

Consider the query $w = w_1w_2$ and an indexed word $u = u_1u_2$ with $\mathrm{ed}(u,w) = d$ and $\mathrm{ed}(u_1,w_1) \leq \lfloor d/2 \rfloor$ (the following arguments can be used in the same manner for the case where $\mathrm{ed}(u_2,w_2) \leq \lfloor d/2 \rfloor$). If u was split during the indexing phase into u_1 and u_2, we can retrieve u by a successful match of w_1 against u_1. What we don't know, however, is which prefix of w we need to consider. Suppose that we always split words in the middle during the indexing phase. Then u_1 is of length $|u_1| = \lfloor |u|/2 \rfloor$. We can easily find a rough bound on the length of the prefixes that we need to consider. Since the edit distance between u and w is d, we know that

$$|w| - d \leq |u| \leq |w| + d,$$

and hence

$$\left\lfloor \frac{|w|-d}{2} \right\rfloor \leq |u_1| \leq \left\lfloor \frac{|w|+d}{2} \right\rfloor.$$

Since $\text{ed}(u_1, w_1) \leq \lfloor d/2 \rfloor$, we can infer

$$|u_1| - \lfloor d/2 \rfloor \leq |w_1| \leq |u_1| + \lfloor d/2 \rfloor,$$

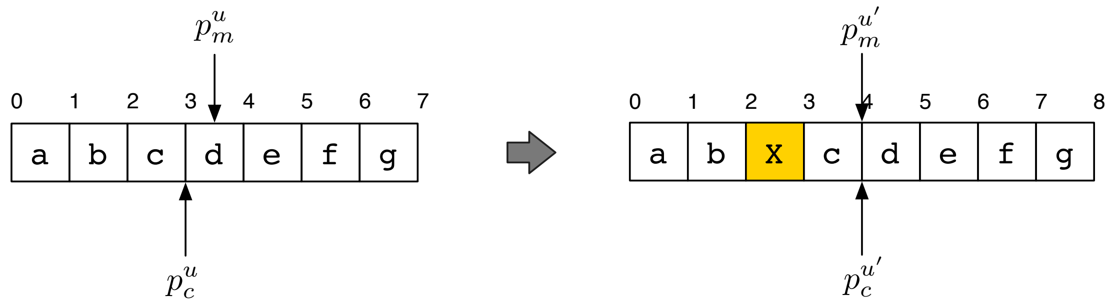
and further

$$\left\lfloor \frac{|w|-d}{2} \right\rfloor - \lfloor d/2 \rfloor \leq |w_1| \leq \left\lfloor \frac{|w|+d}{2} \right\rfloor + \lfloor d/2 \rfloor.$$

We know therefore that w_1 must be a prefix of w with length $|w_1| \in \{\lfloor |w|/2 \rfloor - d, \ldots, \lfloor |w|/2 \rfloor + d\}$. However, this bound is quite loose. We can do better by looking at what actually happens to u_1 when we perform edit operations on it.

To determine the length of w_1, which can be obtained from u_1 by at most $\lfloor d/2 \rfloor$ changes, we need to know where the last character of u_1 ends up in w. As the last character of u_1 is located right in the middle of u, it will be located "somewhere around the middle" of w. Let us take a look on how different edit operations affect c (the last character in u_1), and how they affect the middle of the word. Let $p_c^u := \lfloor |u|/2 \rfloor$ be a pointer that points after the last character of u_1 and $p_m^u := |u|/2$ be a pointer to the middle of u, and let $\delta^u := p_m^u - p_c^u \in \{0, 1/2\}$ denote the difference between those pointers in u. We want to see what happens to δ during different edit operations. Let $op(u) = u'$:

- If op is an insertion, the middle of u' will move a half step to the right, as the word grows by one. Hence, $p_m^{u'} = p_m^u + 1/2$. If the insertion takes place in u_1, the position of c increases by one ($p_c^{u'} = p_c^u + 1$ and $\delta^{u'} = \delta^u - 1/2$), otherwise it does not change ($p_c^{u'} = p_c^u$ and $\delta^{u'} = \delta^u + 1/2$).



- If op is a deletion, the middle of u' will move a half step to the left, as the word shrinks by one. Hence, $p_m^{u'} = p_m^u - 1/2$. If the deletion takes place in u_1, the position of c decreases by one ($p_c^{u'} = p_c^u - 1$ and $\delta^{u'} = \delta^u - 1/2$), otherwise it does not change ($p_c^{u'} = p_c^u$ and $\delta^{u'} = \delta^u + 1/2$).

- If op is a substitution, nothing changes in terms of δ.

We are interested in the difference between p_m^w and p_c^w, i.e. δ^w. We can already see from the argument above that each operation can increase or decrease δ by only $1/2$. We perform d operations, hence δ^u and δ^w will differ by at most $d/2$. We can show that more formally:

- If $\delta^w = p_m^w - p_c^w > 0$, the end of w_1 is right of the middle of w.
- If $\delta^w = p_m^w - p_c^w < 0$, the end of w_1 is left of the middle of w.

If $\text{ed}(u_1, w_1) = d_1$ and $\text{ed}(u_2, w_2) = d_2$, δ^w is maximized if we perform d_1 deletions in u_1 and d_2 insertions in u_2. Then,

$$\delta^w = p_m^u + ((d_2 - d_1)/2) - (p_c^u - d_1) = p_m^u - p_c^u + (d_2 + d_1)/2 = \delta^u + d/2.$$

It is minimized if we perform d_1 insertions into u_1 and d_2 deletions into u_2. Then,

$$\delta^w = p_m^u + ((d_1 - d_2)/2) - (p_c^u + d_1) = p_m^u - p_c^u - (d_1 + d_2)/2 = \delta^u - d/2.$$

Since $\delta^u \in \{0, 1/2\}$, δ^w is bounded by $-\lfloor d/2 \rfloor$ and $\lceil d/2 \rceil$.

Instead of querying the index with w, we can query it with the prefixes of length $|w_1| := \lfloor |w|/2 \rfloor - \lceil d/2 \rceil, \ldots, \lfloor |w|/2| \rfloor + \lceil d/2 \rceil$ and, along the same line of argumentation, with the suffixes of length $|w_2| := \lceil |w|/2 \rceil - \lceil d/2 \rceil, \ldots, \lceil |w|/2| \rceil + \lceil d/2 \rceil$. Hence we have to treat at most $d+1$ prefixes and $d+1$ suffixes.

Naturally there is a trade-off: Usually, most of the residual strings for w will have few original strings. The residual strings of w_1 and w_2, however, will match any string of the form $u := xy$ where x is similar to w_1 or y is similar to w_2. This is especially the case for many kinds of real-world data that consist of strings with common prefixes or suffixes. Think of german city names, for example; There are quite a lot of cities in Germany whose names end in the suffix `"stadt"`. It is therefore crucial that the strings that we split be of a certain length. Our experiments show that it is best to split only those words that are longer than a certain threshold. The threshold at which we consider words too long and split them is a parameter that affects construction time, query time and the space needed by the index. We will determine it in our experiments for several dictionaries.

2.5. Indexing Tokens

For our geocoding application, we want to index not only single words, but also strings that consist of several words or tokens. Take for example the string `"this is a string"`. We could just index the whole string as it is, because our index does not handle whitespace differently from any other character. But then we could not return `"this is a string"` as a candidate for the query `"string"` — the edit distance is way too big. To remedy this shortcoming, we extend our data structure to index *tokens* instead of whole strings. A token in this context is any contiguous substring that contains only letters, i.e. it is free from whitespace, hyphens etc.

To explain the modifications on the data structure, let us look at the process of inserting the string $s :=$`"this is a string"`. We will consider the unaltered index first: Remember

that we keep each distinct string only once in a list. Each residual string of "this is a string" will get a pointer to the position of s in this list.

In the altered index we will not only keep a list of the strings, but also a list of all tokens that appear in the strings. The string "this is a string" will no longer be pointed to by any residual string, but by the tokens "this", "is", "a", and "string". Each token on the other hand will be pointed to by its residual strings, i.e. "strng" points to "string" which in turn points to "this is a string". This is much easier to follow with the help of Figure 4 on the facing page.

2.6. Implementation Details

In a forward-star representation of a directed graph, the nodes are usually consecutively numbered. This has the benefit of being very space-efficient, and it takes only constant time to retrieve the first and last edge leaving or entering a node: If the node's number is v, the edges incident to it can be found in `edges` at the positions `first_edge[v]` to `first_edge[v+1]`.

In our data structure, we store only forward edges, i.e. edges leaving the nodes that correspond to residual strings. To simplify matters, we assumed that we could use a minimal perfect hash function to retrieve such a consecutive numbering of these nodes. This is, however, not how it is really implemented.

Actually, `first_edge` is an array whose size μ we choose ourselves. We use the hash function to map a residual string to an integer, and take the resulting integer modulo the size of the array to find a slot where we can store the edges leaving that residual string. In the general case we will get a considerable amount of hash collisions, depending on the quality of the hash function as well as the size of the array. We choose not to resolve those collisions: Suppose the residual strings r_1 and r_2 occupy the same slot in the array, i.e. $h(r_1) \equiv_\mu h(r_2)$. Then, a query string with residual string r_1 will return not only the original strings for r_1, but also the original strings of r_2. The same holds true for a query string with residual string r_2. This is because we cannot decide which residual string was responsible for placing a candidate into a given slot without further information. We choose to simply ignore this problem. Our index will still work correctly, but the average size of a candidate set increases as the number of hash collisions increases. Of course, we will have to do more edit distance checks, but they should be fast: If we choose a good hash function, it is unlikely that r_1 and r_2 are very similar to each other. Hence, original strings for r_1 and r_2 should also be dissimilar. Therefore, we can often abort the edit distance computation early.

Typically we would choose the size of `first_edge` to be bigger than the number of residual strings to minimize hash collisions. This means that there will inevitably be gaps in the array. Prior we stated that we could always find the edges to a node v in `edges[first_edge[h(v)]]` up to and excluding `edges[first_edge[h(v)+1]]`, but this is no longer true: The field `first_edge[h(v)+1]` may not be occupied by any residual string. To allow for finding the last edge in constant time, we implement a list-like structure on the array. If there is a gap between `first_edges[i]` and `first_edges[j]`, we set `first_edges[i+1] = -j` (see Fig-

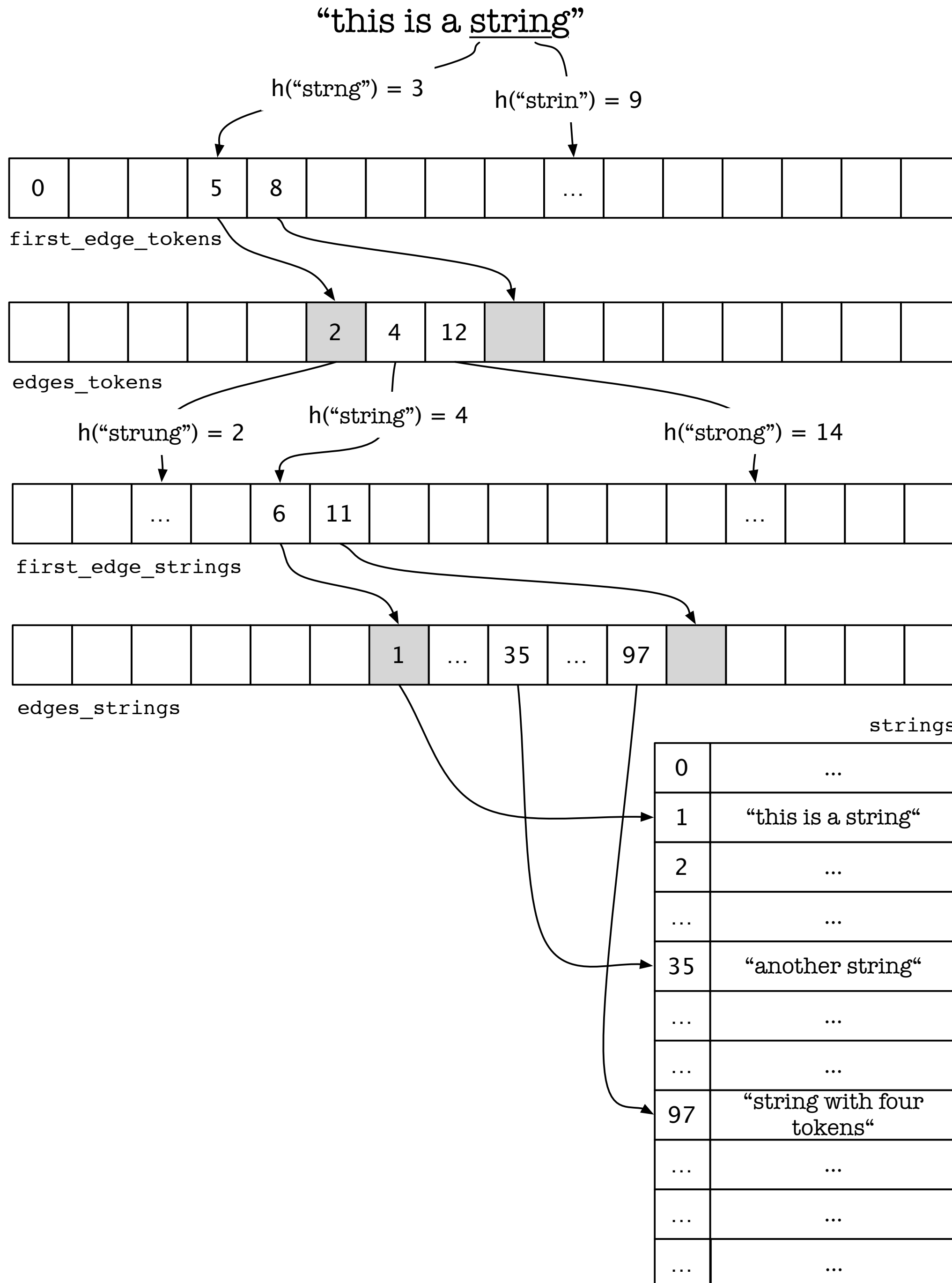


Figure 4: A sketch of the altered data structure with the indexed string "this is a string". Following the arrows from top to bottom, we can see that the tokens belonging to residual string "strng" are located in edges_tokens at positions 5 to 7. These tokens are "strung", "string", and "strong". Moreover we can find the strings containing the token "string" in edges_strings from positions 6 to 10. One of those strings is our original string "this is a string".

ure 5). This way we can traverse the array without having to traverse the gaps, needing only constant time per step.

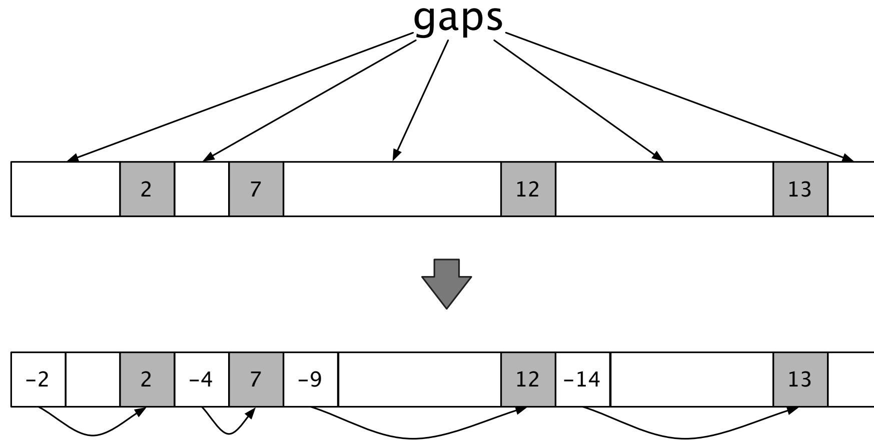


Figure 5: We implement a singly-linked list to bridge gaps in the array.

2.6.1. Generating Residual Strings

To generate the residual strings $\mathrm{rs}(w,d)$ for a given string $w = vx, v \in \Sigma^*, x \in \Sigma$ and d deletions, we implement the following recurrence:

$$\mathrm{rs}(w,0) = \{w\}, \tag{1}$$

$$\mathrm{rs}(w,d) = \emptyset, \text{ if } |w| < d, \tag{2}$$

$$\mathrm{rs}(vx,d) = \mathrm{rs}(v,d-1) \cup \{ux \mid u \in \mathrm{rs}(v,d)\} \tag{3}$$

We use dynamic programming to avoid repeatedly computing the same sets, hence each set $\mathrm{rs}(v',d')$ that appears on the right side of (3) has to be computed only once. Let us first look at the cost that arises after the recursive calls have returned. We then have to unify the two sets. If $|w| = n$, then both sets consist of $\mathcal{O}(n^d)$ residual strings. To compute the union, we have to sort both sets, which can be done in time $\mathcal{O}(n \cdot n^d \log n^d) = \mathcal{O}(dn^{d+1} \log n)$ (a comparison takes time linear in the length of w). Hence, we know the "direct" cost that arises from the computation of a set rs(w, d), and it depends only on d and on the *length* of w, not on w itself. We therefore define

$$c_n^i := \mathcal{O}(in^{i+1} \log n)$$

as the direct cost that is associated with a string of length n and i deletions. To compute the complete cost, we need to know how many sets $\mathrm{rs}(w',d')$ will be generated. Note that in (3), both recursive calls are performed on v, which is a prefix of w, therefore we only need to consider sets $\mathrm{rs}(w',d')$ where w' is a prefix of w and $d' \leq \min\{|w'|,d\}$. Then we can bound the total cost by

$$\sum_{m=1}^{n} \sum_{i=0}^{\min\{m,d\}} c_m^i < n(d+1) \cdot c_n^d = \mathcal{O}(d^2 n^{d+2} \log n).$$

2.6.2. Edit Distance

As we measure similarity between strings in terms of edit distance, we need a way to compute the edit distance between two given strings. The classical algorithm to compute the Levenshtein-Distance is based on dynamic programming and computes the edit distance between two words u and v in $\mathcal{O}(|u| \cdot |v|)$ time and $\mathcal{O}(\min\{|u|, |v|\})$ space.

In our application we usually only need to know if $\mathrm{ed}(u, v) \leq d$ for some constant d, which means that we can abort the computation as soon as we know that the edit distance between u and v is at least $d+1$ (see Fig. 6). We can also directly abort the computation if $||u| - |v|| > d$.

		c	o	n	s	t	r	u	e	d
	0	1	2	3	4	5	6	7	8	9
c	1	0	1	2	3	4	5	6	7	8
o	2	1	0	1	2	3	4	5	6	7
n	3	2	1	0	1	2	3	4	5	6
t	4	3	2	1	1	1	2	3	4	5
r	5	4	3	2	2	2	1	2	3	4
a	6	5	4	3	3	3	2	2	3	4
c	7	6	5	4	4	4	3	3	3	4
t	8	7	6	5	5	4	4	4	4	4

Figure 6: The dynamic programming matrix is filled column-wise from left to right. The minimal value in each row or column is a lower bound on the edit distance of the compared words. If we want to check if the edit distance between `"construed"` and `"contract"` is at most 2, we can abort the computation in the penultimate column.

An interesting extension to the Levenshtein-Distance is the Damerau-Levenshtein-Distance that takes into account not only insertions, deletions and subsitutions, but also transpositions of characters. In his seminal work [7] Damerau states that more than 80% of human misspellings fall into these classes of error. The Damerau-Levenshtein-Distance can also be computed in $\mathcal{O}(|u| \cdot |v|)$ time [44].

There have been developed faster methods that use bit-parallelism, which could be used to further improve our index [33].

2.6.3. The Tokenizer

In our application, we often have to split a string into tokens, i.e. contiguous strings of characters, delimited by whitespace, hyphens, colons etc. Since we tokenize the strings already during the preprocessing phase, it is not necessary to do the same work again in the query phase. Instead, we can store for each token the position of its first character and its length. The strings in our dictionary are rather short, so this information fits into two Bytes per token (see Fig. 7). This is of course not possible for the query string. There are some tokens

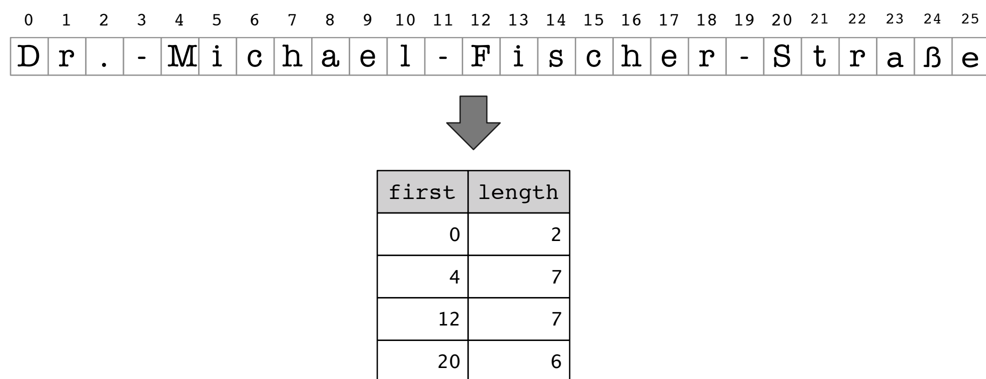


Figure 7: We can store information about the tokens to speed up the tokenizing during the query phase.

that receive a special treatment. In the setting of geocoding, we will use the index to retrieve street names. In Germany, about half of the street names have the suffix `"straße"`, which is german for street or road, and there are other similar suffixes such as `"weg"`, `"gasse"` etc. Sometimes these suffixes are separated from the street name by a hyphen or a blank (as in `"Longué-Straße"` or `"Rintheimer Straße"`) and sometimes they are not (as in `"Hauptstraße"`). There is no fixed rule as to when one has to use whitespace or hyphens, hence we cannot expect the user to know how the respective street is filed in our index. We want to treat `"Longué-Straße"` the same as `"Longuéstraße"`. We do this by repeatedly querying an index that contains only the special tokens for different suffixes of a string. There are more advanced methods to check this, e.g. based on finite automata [2], that can check for several tokens at once in time linear in the length of the suffix.

The tricky part is to identify special tokens that contain errors. Consider a special token t and an erroneous version t' of that token with $\mathrm{ed}(t, t') \leq d$. Then, $|t| - d \leq |t'| \leq |t| + d$. If $w = vt$ is a street name that contains a special token and $w' = v't'$ is an erroneous version of w, we don't know exactly where to split w'. Consider the following example:

$$w = \texttt{Musterstrasse}, w' = \texttt{Mustersxtrasse}$$

If "strasse" is the longest special token in our index and "weg" is the shortest, and if we want to allow one error in a special token, we have to look at the suffixes of length $2, \ldots, 8$. We will find that

$$\begin{aligned} \text{ed}(\texttt{strasse}, \texttt{sxtrasse}) &= 1 \\ \text{ed}(\texttt{strasse}, \texttt{xtrasse}) &= 1 \\ \text{ed}(\texttt{strasse}, \texttt{trasse}) &= 1 \\ \text{ed}(\texttt{gasse}, \texttt{rasse}) &= 1 \\ \text{ed}(\texttt{gasse}, \texttt{asse}) &= 1 \end{aligned}$$

Let $w' = v't'$, where t' is the erroneous version of a special token. Then, $v' \in \{\texttt{Muster}, \texttt{Musters}, \texttt{Mustersx}, \texttt{Mustersxt}, \texttt{Mustersxtr}\}$.

Example 2: A query where we cannot easily determine the correct version for a split. The special token "straße" is internally represented as "strasse", because many people use the latter version. In Switzerland the letter 'ß' is not used at all, and it is always written as 'ss'.

We solve this problem by splitting the word into overlapping parts. In the above example, we would split "Mustersxtrasse" like this:

$$\texttt{Mustersxtrasse} \rightarrow (\texttt{Muster}\underline{\texttt{sxtr}}, \underline{\texttt{sxtr}}\texttt{asse})$$

Since we are not sure which parts of "sxtr" belong to the prefix and which belong to the suffix, we leave them in both the suffix and the prefix and mark them as *optional*. When performing a lookup with the string "Muster<u>sxtr</u>", we have to query the index for the strings "Muster", "Muster<u>s</u>", "Muster<u>sx</u>", "Muster<u>sxt</u>", and "Muster<u>sxtr</u>", if we want to make sure that we actually hit the correct candidate. During the indexing phase, the candidate "Musterstrasse" will have been correctly split into the pair $(\texttt{Muster}, \texttt{strasse})$, which we will now find as a candidate. To compute the distance between "Muster" and "Muster<u>sxtr</u>", we match "as much as we need", i.e., we return the smallest edit distance between "Muster" and "Muster$x$", where $x \in \{\varepsilon, \texttt{s}, \texttt{sx}, \texttt{sxt}, \texttt{sxtr}\}$.

We treat the tokens "<u>sxtr</u>asse" and "strasse" similarly, but perform the match backwards, i.e., we match "essa<u>rtxs</u>" against "essarts". The added cost for the extra lookups is quite moderate.

		M	u	s	t	e	r	s	x	t	r
	0	1	2	3	4	5	6	7	8	9	10
M	1	0	1	2	3	4	5	6	7	8	9
u	2	1	0	1	2	3	4	5	6	7	8
s	3	2	1	0	1	2	3	4	5	6	7
t	4	3	2	1	0	1	2	3	4	5	6
e	5	4	3	2	1	0	1	2	3	4	5
r	6	5	4	3	2	1	0	1	2	3	4

Figure 8: The best way to match "Mustersxtr" against "Muster" is to match none of the optional characters.

2.7. Analysis

There are various parameters that affect the memory requirements of the index, and there is clearly a trade-off between memory and query time. If we don't use perfect hashing, one of the parameters is the size of the array `first_edges`, i.e. the hash-table. Since we don't resolve hash collisions, a too small choice of the size for this array will result in big candidate sets that have to be verified. The split parameter, on the other hand, can be used to reduce the number of residual strings as well as the number of edges pointing from residual strings to original strings, thus allowing us to choose a smaller hash table.

2.7.1. Memory Consumption

The size of the data structure is dominated by the arrays edges and `first_edge`. To bound the number of edges, notice that a word w can have at most $\sum_{k=0}^{d}\binom{|w|}{k} = \mathcal{O}(|w|^d)$ residual strings for d errors, each of which gets an edge to the original word. The size of the array `first_edge` corresponds to the number of distinct residual strings if we use perfect minimal hashing, or to whatever size we choose it to be if we allow hash collisions.

To give a bound on the total number of residual strings for a dictionary D, we must know the distribution of words in the dictionary. Let α_l be the number of words in D with length l. Then the total number of residual strings is bounded by

$$\sum_{l\geq 1}\alpha_l \sum_{k=0}^{d}\binom{l}{k}.$$

In some of our sample dictionaries (e.g. the titles of the articles in the english Wikipedia or the novel “Moby Dick”, the distribution of word lengths resembled a binomial distribution (see. Figure 9 on page 26).

Consider the following experiment: We want to generate a string s with length at most n. We throw a coin n times that shows head with probability p, and add a character to s each

time the coin shows head. Then the length of s is a random variable X whose distribution is binomial with parameters n and p.

If we assume that the length of the strings in a dictionary D is a random variable X that follows a binomial distribution with parameters n and p, then the expected number of strings with length l is

$$\mathrm{E}[\alpha_l] = |D| \Pr[X = l] = |D| \cdot \mathrm{B}(k|n,p) = |D| \binom{n}{l} p^l (1-p)^{n-l},$$

yielding an upper bound on the expected number of residual strings:

$$|D| \sum_{l=1}^{n} \binom{n}{l} p^l (1-p)^{n-l} \sum_{k=0}^{d} \binom{l}{k}$$

2.7.2. Query Time

The query can be divided into two separate phases: The filtration phase and the verification phase. In the filtration phase, we prune the search space by generating a set of candidates that contains every valid match and additionally some invalid matches. Suppose that our index has been generated for up to d deletions. For a query q of length m we have to generate $\sum_{k=0}^{d} \binom{m}{k} = \mathcal{O}(m^d)$ residual strings in $\mathcal{O}(d^2 m^{d+2} \log m)$ time, and we have to compute the hash value for each of them, which takes $\mathcal{O}(m)$ time per residual string. We also need to mark every original string as a candidate. If there are cnd such candidates, the filtration phase needs $\mathcal{O}(d^2 m^{d+2} \log m + cnd)$ time.

We have to check the actual edit distance of each candidate to the query string. If we use the classic dynamic programming approach, this takes $\mathcal{O}(m^2 cnd)$ time, but since we are only interested in the exact edit distance if it is not greater than d, it suffices to compute a diagonal strip of width $2d+1$ in the matrix, yielding a time bound of $\mathcal{O}(md \cdot cnd)$ for the verification phase.

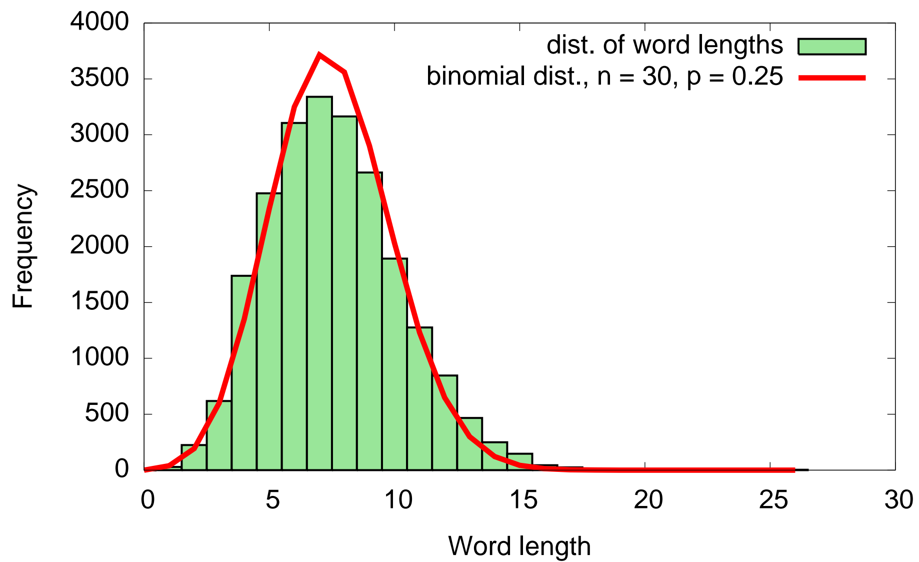
Summing all up, a query can be performed in time $\mathcal{O}(d^2 m^{d+2} \log m + cnd + md \cdot cnd) = \mathcal{O}(d^2 m^{d+2} \log m + md \cdot cnd)$.

2.8. Experiments

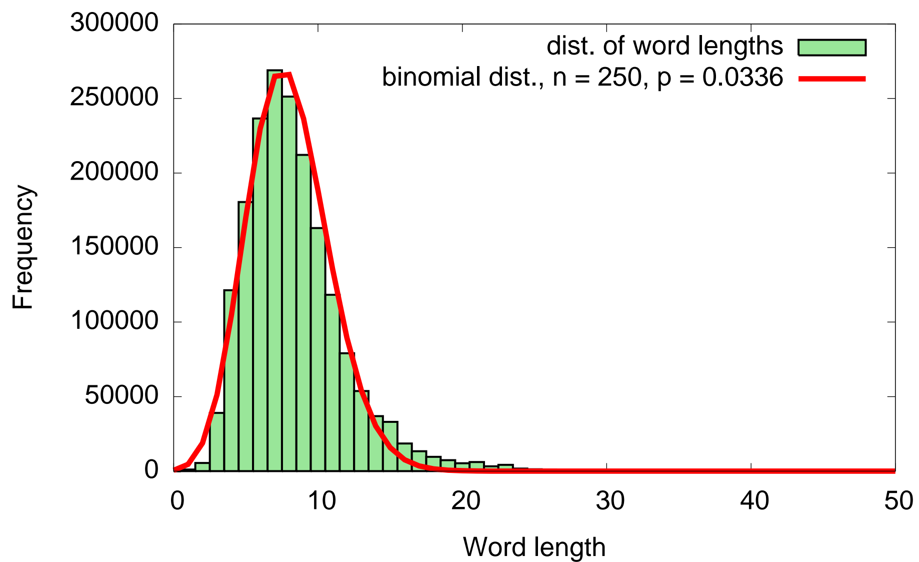
We conducted experiments to see how the query times and the memory consumption are affected by the size of the dictionary as well as by the various parameters. All experiments were performed on an Intel Core i7 920@2.67GHz with 12GB RAM on Linux 2.6.27. The programs were implemented in C++ using GCC 4.3.2.

The parameters are

- the number of *slots*: The size of the table `first_edge`. We hash the residual strings into this table. If we choose it too small, the number of hash collisions grows.



(a) Moby Dick



(b) Wikititles

Figure 9: The distribution of word lengths for natural language dictionaries sometimes resembles a binomial distribution.

- the *split* parameter: The threshold at which we decide to split a word into halves, as described in section 2.4 on page 15.

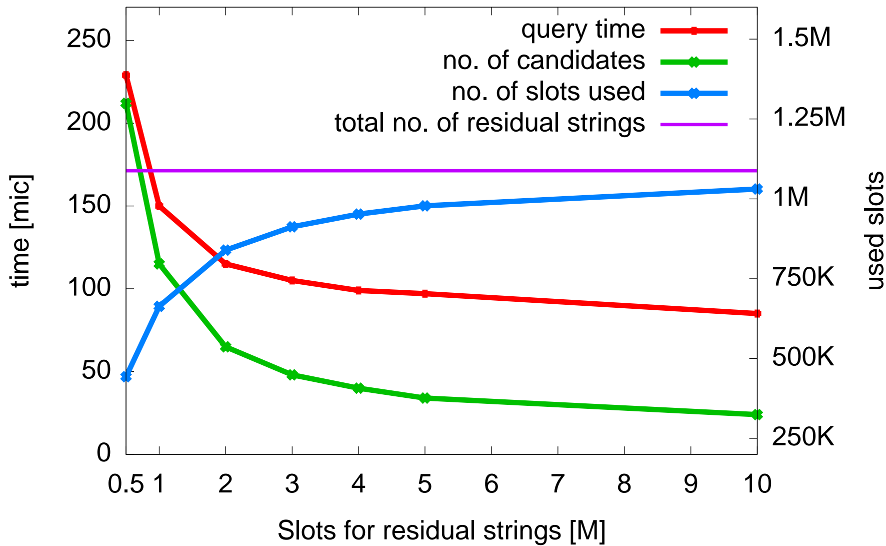


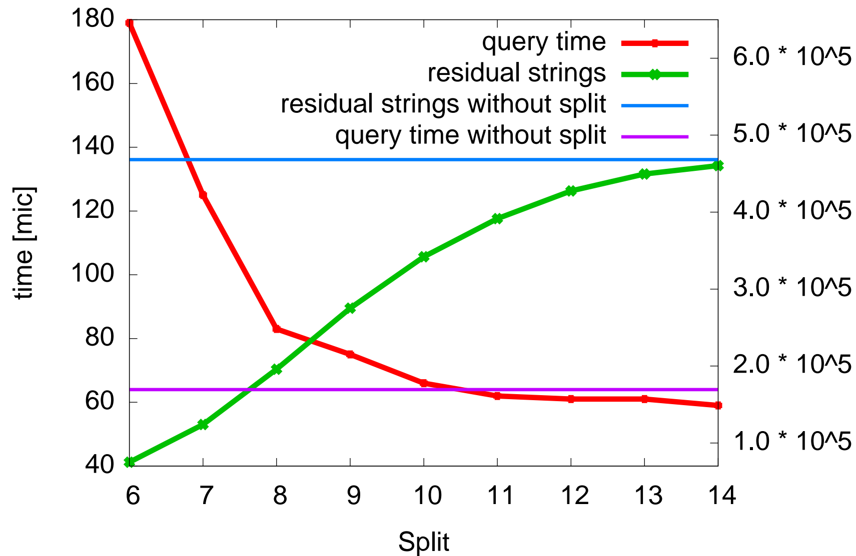
Figure 10: The query time degrades if we assign too few slots to `first_edge`.

Figure 10 shows how the size of the array `first_edge` affects the query time. The dictionary used was the list of distinct words in the novel "Moby Dick" by HERMAN MELVILLE [27], as available on Project Gutenberg [15]. It contains 17149 distinct words and the index was constructed to support the correction of up to 3 errors. As can be seen from the plot, the total number of distinct residual strings for this dictionary is about 1.1M. If we choose the size of the array smaller than this value, hash collisions are inevitable. It is interesting to note the symmetry between the number of slots that are occupied and the number of candidates that have to be verified.

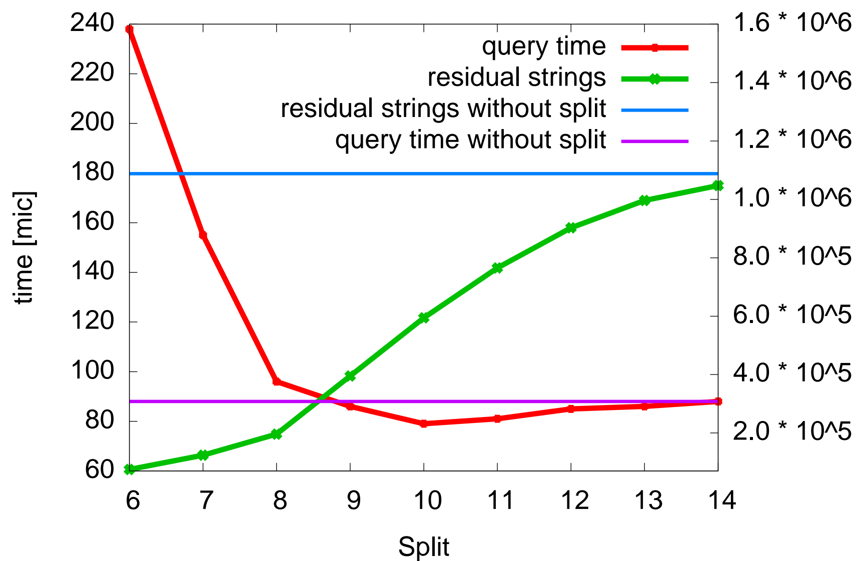
For Figure 12 on page 29 we choose the size of `first_edge` big enough, such that there are only very few hash collisions. The plot shows how the size of the dictionary affects the query time and the time that is needed to construct the index. As can be seen, the query time increases as the size of the dictionary increases. This is due to the fact that the average number of candidates that have to be verified grows as well.

Figure 11 on the following page shows the effect of the split parameter. The smaller we choose the split parameter, the fewer residual strings are generated. Unfortunately, the split words match too many candidates if we choose the split parameter too small.

Surprisingly there seems to be a "sweet spot" where splitting even improves the query time. The reason for this becomes clear in Figure 13 on page 29: The smaller we choose the split parameter, the fewer time is spent generating the residual strings. The verification phase, on the other hand, takes significantly longer. For the dictionary in question ($d = 3$), the optimal split parameter is 10.



(a) $d = 2$



(b) $d = 3$

Figure 11: If we choose a sensible value for the split parameter, we can reduce the number of residual strings and even accelerate the query.

2.8.1. Implementation Note

Figure 13 on the facing page also shows where there is room for further improvement. Improving the verification phase should be straightforward, as the dynamic programming approach that we use for edit distance computations is not optimal. The construction of the residual strings can probably be improved by altering the hash function: At present we generate the residual strings by successively deleting characters from the original string. This

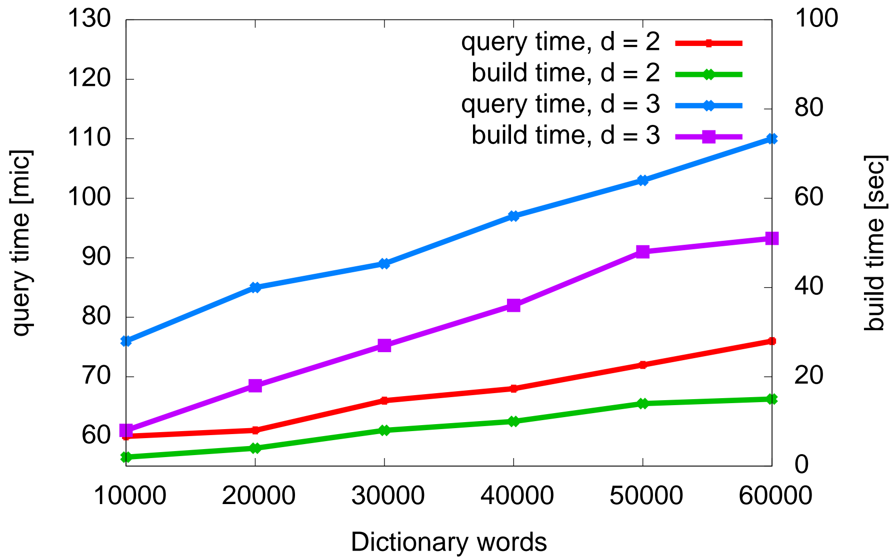


Figure 12: Build time and query time grow with the size of the dictionary.

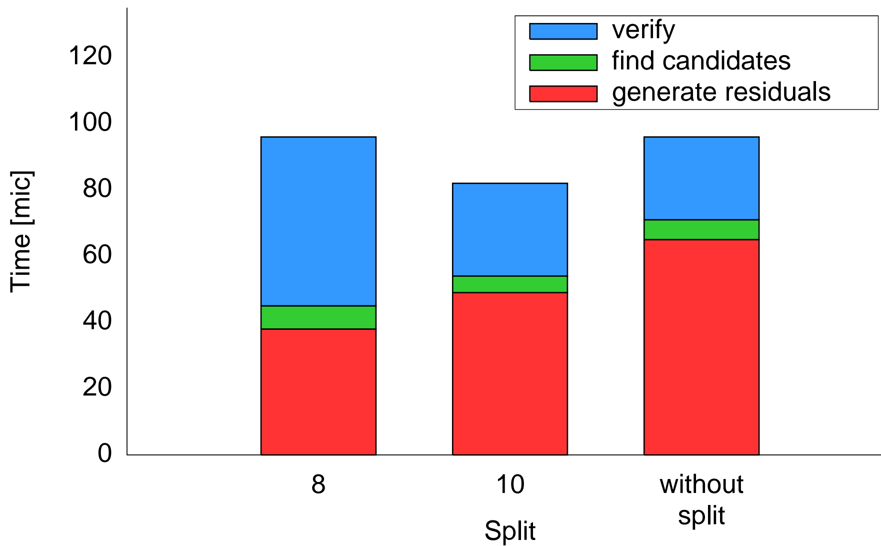


Figure 13: If we don't split the strings, most of the time is needed for the construction of the residual strings.

involves copying the string every time. Thereafter we call the hash function, which takes a `std::wstring` and produces an `unsigned int`. We could avoid the copying of the strings by altering the hash function such that it takes the original string and the positions of the characters to be deleted, thus making it unnecessary to actually copy the string and delete characters from it.

3. Geocoding

A central aspect of this thesis is the application of the fault-tolerant index to the domain of geocoding.

Our goal is to return a geographic location given a text-based address query. The information one would like to retrieve could be references to any kind of spatial entity, like towns, streets, parks, lakes etc. In our scenario, we only care about postal addresses, i.e. streets and towns. Potential uses of such a service include:

- Interactive Address Search: A service where a human user formulates a textual query that describes a location which the user wishes to have geographically referenced. The service may return a number of results from which the user can then choose.
- Batch-Processing: A company may have a data base of erroneous postal addresses (e.g. retrieved from scanned images by OCR) that it wants to automatically check and clean if necessary.
- Address Validation: A service provider may require customers to enter an address to register for their services. A geocoding service could then be used to verify that the address does in fact exist.

For the towns, we store their names, positions and a *rank*. The rank will be used to sort the towns by their importance. One could use a town's population, the number of streets in a town or any other criteria for importance. Additionally we store a parent-child relationship between a *principal town* and its *districts* (e.g. `"Berlin"` is the principal town of `"Kreuzberg"`, while `"Kreuzberg"` is a district of `"Berlin"`).

Definition 6. *Let x be a town with principal town y (x and y are equal if x is a principal town itself). The set that contains y and all of its districts is called the* perimeter *of x.*

Streets are fully defined by their name and the town they belong to. Note that it is common for different towns/streets to share their name: There are more than 20 `"Neustadt"` and over 12000 `"Hauptstraße"` in Germany. Therefore we store with the towns only a cursor into a table where we store the actual strings. Each string, on the other hand, gets assigned a list of town IDs.

In the following explanations, it is important to distinguish *towns* and *streets*, which are always associated with a town `id` and represent actual spatial entities, from *candidates*, which are merely strings. We will name street candidates $s, s_1, \ldots$, town candidates $t, t_1, \ldots$, or c in the more general case. Actual towns/streets that reference a spatial entity will be called v, w, x etc.

The challenge is now to quickly interpret a given query and to return the intended address, or maybe several addresses. When we talk about an *address*, we mean a pair of a town and a street that is located in that town. We do not consider house numbers or postal codes.

There are several classes of errors a user of a geocoding system could make:

Town		
id	:	int
position	:	(float,float)
rank	:	int
parent_id	:	int
string_id	:	int

TownStrings		
id	:	int
town_ids	:	[int]
string	:	varchar

StreetStrings		
id	:	int
town_ids	:	[int]
string	:	varchar

Table 2: The data model that we use. Streets are defined by their name and the town they belong to.

- typing errors, e.g. `"Hambzrg"` instead of `"Hamburg"`
- missing information, e.g. `"Frankfurt Oder"` instead of `"Frankfurt an der Oder"`
- additional information, e.g. `"Offenbach am Main"` instead of `"Offenbach"`
- wrong assignment of streets to towns, e.g. `"Mollstraße, Berlin Kreuzberg"` instead of `"Mollstraße, Berlin Mitte"`.

In short, it is unlikely that a human user will always be able to enter a postal address exactly as it is stored in the data base. In the remainder we will show a method to quickly prune the search space by combining the index which is based on edit distance with the geographic information that we obtain from the address data.

To improve the quality of results as well as the performance of the query, we want to exploit our knowledge of the geographic location of the data. We will use that information in different ways:

1. Already at an early stage of the algorithm, we can drop most of the candidates if we can make sure that they cannot be part of a result.
2. The user will have the possibility to specify his query by providing a second town in the input, e.g. "STREET *in* TOWN *near* BIGTOWN" instead of "STREET *in* TOWN".
3. If a street cannot be found in the town provided by the user, we will search also in the *perimeter*, i.e. other towns that belong to the same principal town.

How this is achieved will be explained in the remainder of this section.

3.1. Dropping incompatible candidates

Each street in our data is associated to a town. We call the set of towns $\mathscr{T}$, the set of streets $\mathscr{S}$, and the sets of candidate strings $C_{\mathscr{S}}$ and $C_{\mathscr{T}}$ respectively. If a street runs through more than one town, e.g. if it connects two towns, we treat it as two distinct streets. Consider a query `"S in T"`, where S shall be a street and T a town. If we query the town and street index, each will return a set of candidates $\text{cand}(S) \subset C_{\mathscr{S}}$ and $\text{cand}(T) \subset C_{\mathscr{T}}$. Keep in mind that a candidate at this point holds no geographic information; it is only a string that represents the name of possibly hundreds or thousands of towns or streets.

If c is a candidate, let $\text{towns}(c) \subset \mathscr{T}$ be the town IDs that correspond to the candidate. We are looking for pairs $(t,s) \in C_{\mathscr{T}} \times C_{\mathscr{S}}$ such that s represents a street that actually exists in a town with name t. Obviously a candidate $s \in \text{cand}(S)$ can only occur in such a pair if there is a town candidate $t \in \text{cand}(T)$ such that $\text{towns}(s) \cap \text{towns}(t) \neq \emptyset$. In this case we say that s and t are *compatible*.

Definition 7. *We call a street (town) candidate c* compatible *to a set of town (street) candidates $\mathscr{C}$ if*

$$\text{towns}(c) \cap \left(\bigcup_{c' \in \mathscr{C}} \text{towns}(c') \right) \neq \emptyset$$

Typically, we can reduce the size of the candidate sets significantly by dismissing incompatible candidates. See Fig. 15 on the next page for an illustration and Fig. 14 on the facing page for an example.

For a set of Towns $\mathscr{T}' \subset \mathscr{T}$ and a set C of candidates, finding the subset of compatible candidates is simple: If for a candidate $c \in C$ the intersection $\text{towns}(c) \cap \mathscr{T}'$ is not empty, we can keep the candidate , otherwise we remove it from the set of candidates.

In the geocoding algorithm, dropping incompatible candidates at an early stage is crucial to the performance of the query. On a very high level, the algorithm works as follows:

1. Retrieve a set of candidates from the fault-tolerant index.

2. Compute a rating for each candidate.

3. Choose the candidates with the best rating and return them.

The running time of the query is dominated by the second step, i.e. by the rating of the candidates, because it involves a lot of edit distance computations. The easiest way to speed up those computations is probably to avoid them as much as possible, therefore we try to dismiss candidates as soon as we know that they cannot possibly be part of a successful result. It is important to emphasize that we drop most of the candidates *without even looking at the strings*. In fact this step can be performed very quickly, since it involves only intersections of `std::vector<unsigned int>`s.

Query: "Longue-Straße, Sinsheim"	**town candidates**	**street candidates**
before intersection	129	2816
after intersection	17	11

Figure 14: An example query where the search space is reduced drastically by dismissing incompatible candidates.

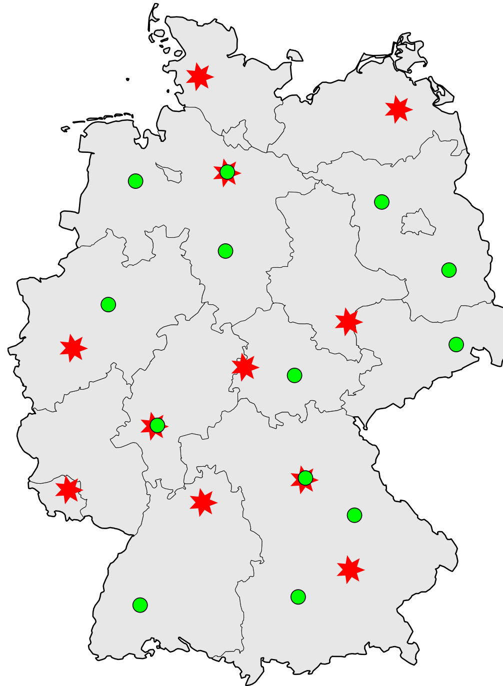


Figure 15: The towns that correspond to the town candidates are indicated by stars, the towns that correspond to the street candidates by circles. We are only interested in towns that are marked by both a star and a circle.

3.2. Tie-Breaking

Sometimes it is not possible to return a unique result because there are several locations that match the query. In such situations, one cannot guess correctly the intention of the user without further information. For this reason it should be possible for the user to provide additional information (e.g., "Neustadt *near* *Coburg*") to distinguish the expected result from other results.

To specify the query, the user will typically choose a city that is close to the original town in a geographic sense and more important according to a certain criteria. One can think of such a town as a *landmark*.

Clearly, not all towns come into consideration as a landmark. Consider the towns x_{small} and x_{big}. The bigger town x_{big} is a landmark for x_{small} if

1. it is close to x_{small} in a geographic sense and
2. there is no town in between that is "more important": We assume that the user associates a town with the nearest town that he believes to be significant enough.

In the remainder of this section we will describe a graph that models those relationships.

3.2.1. The Neighborhood Graph

The graph that we are going to build should model the relationship between towns and their landmarks. This means that for any town x there should be a path to y if the distance between x and y is sufficiently small and if y is more important for x than any other town between x and y. What exactly we mean by *between* and *more important* will become clear in the following text. We will use *Voronoi Diagrams* and *Delaunay Triangulations* to construct the graph.

A Voronoi diagram is a decomposition of a metric space $\mathscr{M}$ into a set of *Voronoi cells* by a discrete set of objects, e.g. points in the space (the *Voronoi sites*). Each Voronoi site x defines a cell $\mathscr{C}_x$ consisting of all points in $\mathscr{M}$ that are closer to x than to any other site. The points that are equidistant to more than one site build the segments of the diagram. In our case,

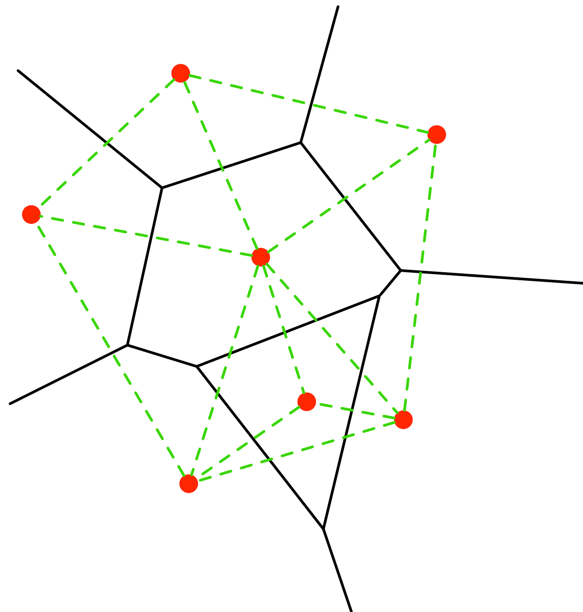


Figure 16: A Voronoi diagram and the corresponding Delaunay Triangulation.

the metric space is the euclidian plane and the sites are the principal towns.

We obtain the dual graph of a Voronoi diagram by drawing edges between sites if and only if they share a segment in the Voronoi diagram. The resulting Graph is a triangulation and is called a *Delaunay Triangulation*. We will need the following well-known lemma about Delaunay Triangulations (without proof):

Lemma 8. *If a circle passing through two of the input points doesn't contain any other of them in its interior, then the segment connecting the two points is an edge of a Delaunay triangulation of the given points.*

Let $V = \{v_1, \ldots, v_n\}$ be the set of towns, ordered by descending rank and $V_i := \{v_1, \ldots, v_i\}$. Let $\mathrm{coord}(v_i) \in \mathbb{R}^2$ denote the coordinate of v_i. We begin with a Graph $G = (V, \emptyset)$ and maintain a Voronoi diagram $\mathrm{Vor}(V)$. Then we successively delete the nodes from V, yielding a new Voronoi diagram after each deletion step. In every step we delete the node with the lowest rank of all remaining nodes. Let $\mathscr{C}_v^i$ denote the Voronoi cell of node v in $\mathrm{Vor}(V_i)$ and consider the step when node v_i is to be deleted:

If $\mathscr{C}_{x_1}^i, \ldots, \mathscr{C}_{x_k}^i$ are the cells adjacent to $\mathscr{C}_{v_i}^i$ in $\mathrm{Vor}(V_i)$, we add the edges $(v_i, x_1), \ldots, (v_i, x_k)$ to G (see Fig. 17). By the order in which we delete nodes, we know that $x_1, \ldots, x_k$ are ranked

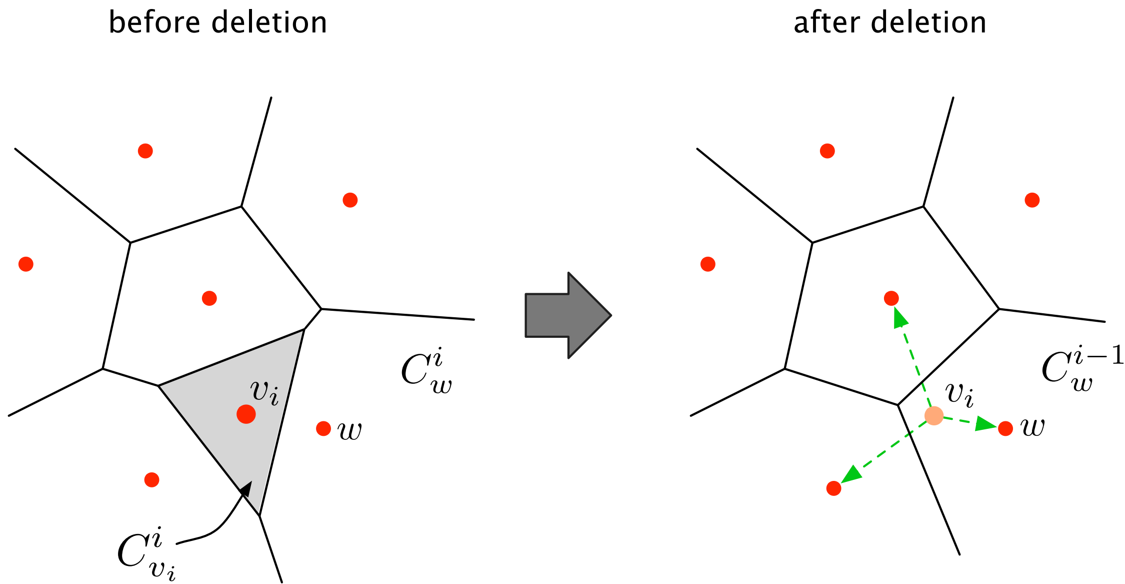


Figure 17: On the left: A cutout of the Voronoi diagram (black) and Delaunay graph (green, dashed) at step i. Node v_i will be deleted. On the right: The situation after v_i has been deleted. v_i gets assigned a neighborhood.

higher than v_i, otherwise they would have been deleted before v_i. We also know that v_i gets assigned an edge to its nearest neighbour x_j in $\mathrm{Vor}(V_i)$, because their Voronoi cells are adjacent: According to lemma 8, the edge $\{v_i, x_j\}$ must be in the Delaunay triangulation of V_i, therefore their cells must share a segment in $\mathrm{Vor}(V_i)$. In other words, *every town will have an edge to the nearest town that is ranked higher*.

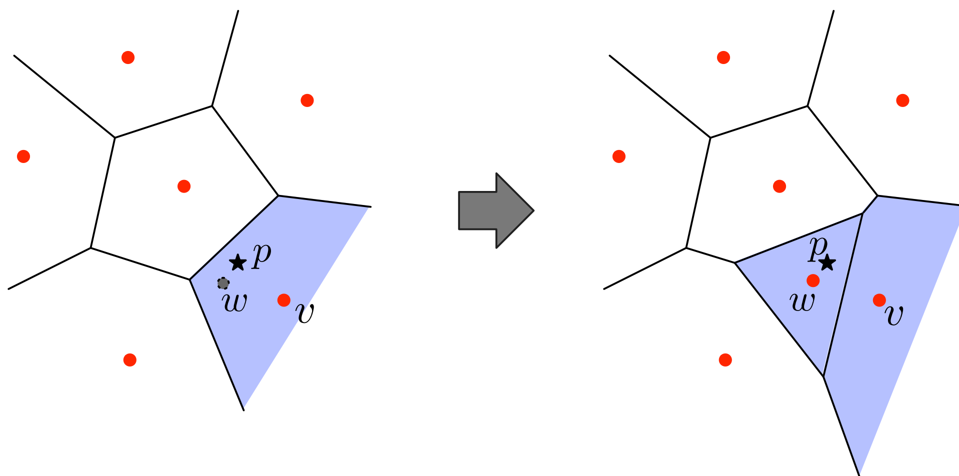
Notice that the edges we add to the neighborhood graph after deletion of v_i are in the Delaunay triangulation of V_i, because they are segments of the cell $\mathscr{C}_{v_i}^i$ in $\mathrm{Vor}(V_i)$. In fact, the algorithm can be implemented "backwards": Instead of deleting nodes from the Voronoi

diagram, we can start with an empty graph and incrementally add the nodes $v_1, \dots, v_n$, maintaining a valid Delaunay triangulation and Voronoi Diagram in every step. The edges that we add to the triangulation during the insertion of v_i are the same edges that would be added in the above algorithm after v_i would be removed from $\mathrm{Vor}(V_i)$.

Lemma 9. *Every town $v_j, j > i$ with $\mathrm{coord}(v_j) \in \mathscr{C}^i_{v_i}$ has a path to v_i in the neighborhood graph.*

Proof. The proof will be stated in terms of Voronoi diagrams, but "backwards", as explained above. In this proof we will call a cell $\mathscr{C}_v$ *blue* iff there exists a path from v to v_i. The other cells are *white*. When a node is inserted into a blue cell or into a white cell that is adjacent to a blue cell, its cell becomes blue. To show that there is a path from v_j to v_i, we have to show that v_j is inserted into or next to a blue cell.

Once a node's cell is blue, it will stay blue obviously. But this is not only true for the node, but for *any coordinate* in a blue cell:



Suppose a point p lies inside of a blue cell C_v. For the point to "change" its cell, we have to insert a node w such that p's distance to w is smaller than its distance to v. Then $\mathscr{C}_w$ must be adjacent to $\mathscr{C}_v$, thus becoming a blue cell itself. Hence, p remains in a blue cell.

Since $\mathrm{coord}(v_j)$ lies inside of the blue cell $\mathscr{C}^i_{v_i}$, its own cell $\mathscr{C}^j_{v_j}$ will be blue, too. See Fig. 18 on the facing page for a sketch. □

See Algorithm 4 for the pseudo-code.

Corollary 10. *If for two towns v_i and v_j, $i < j$ there is no directed path from v_j to v_i, there must be a higher ranked town v_l, $l < i$ that is closer to v_j.*

Proof. From the above lemma follows that $\mathrm{coord}(v_j)$ does not lie inside $C^i_{v_i}$. Then it must lie inside another Voronoi cell $C^i_{v_l}, l < i$. □

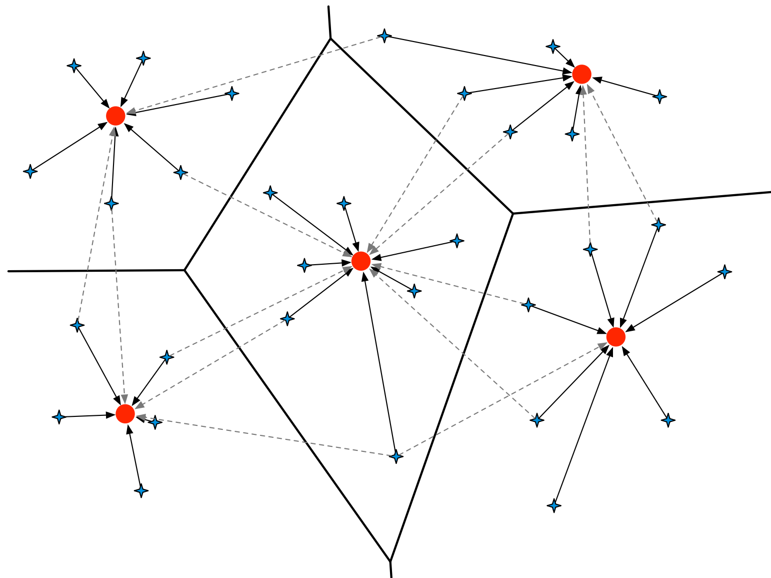


Figure 18: An illustration of lemma 9 on the facing page: The above sketch represents the state of the Voronoi diagram after the five biggest towns (the fat red points) have been inserted. Any smaller town (the small crosses) that will be inserted afterwards will have a path to the center of the cell where it is located (the solid lines). Note that they may also have paths to other Voronoi sites (the dashed lines).

Definition 11. *We call the set of nodes that are reachable from a node v its* neighborhood $\nu(v)$.

We could already use the graph as described above to enhance the quality and performance of the search, but there are still some issues that we need to resolve. The neigborhood is too big and there are many redundant edges. In the neighborhood graph for Germany, the average number of neighbours is 280. Especially the highest ranked town, Berlin, is a neighbour of every german town, which is hardly desirable. Suppose we want to dismiss superfluous neighbours of a town v. We drop a neighbour w if there is another neighbour x that is "between" v and w and "important enough" to dominate w.

We will use light sources as an analogy. Think of a town as a point-shaped light source. The importance of a town corresponds to the brightness of its light, which decreases as the distance to the light source increases.

Definition 12. *Let u be a town with rank r_u. The* brightness *of u at distance l from u is*

$$br_l(u) = \frac{r_u}{l^2}$$

To decide if a town w should be dropped from the neighborhood of a town u, we "look towards" w. If there is a light source v between u and w that outshines w, we keep only v as a neighbour and drop w. If w is a neighbour of u, it can only be dominated by nodes that lie inside the sector of a circle with radius $dist(u, w)$ and angle α. See Fig. 19 for an illustration. To check which neighbours can be dropped, we have have to compare each pair of outgoing

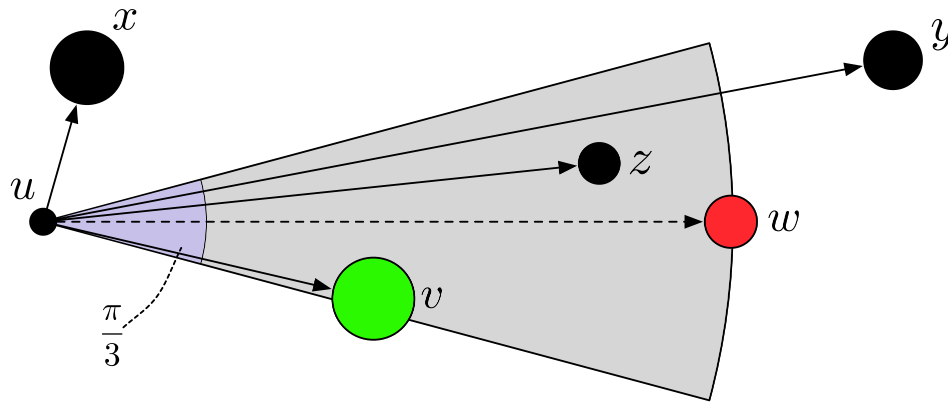


Figure 19: The size of the nodes indicates their brightness as it is perceived from node u. w is dominated by v, but neiter by x (angle too big) nor by y (distance too big) nor by z (not as bright as w). The edge (u, w) is dropped.

edges of each node. In our case the resulting graph has an average out-degree of 9.

The domination rule is simple and can be evaluated quickly, but it can lead to problems. Look at the graph in Fig. 20 on the facing page. There, every node $v_i, i \in 1, \dots, 6$ is dominated by node v_{i+1}. After the domination phase, only node v_7 remains, even if our intuition tells us that some of the nodes $v_1, \dots, v_6$ could also remain in the graph. This is because we don't check if a node v dominating another node w is dominated itself.

As we will show in our experiments in Section 3.8.5 on page 60, the described method works well with the graph obtained from german address data. However, it has not yet been tested on different graphs. It would be interesting to test this method also on countries whose graphs differ structurally from the german graph, e.g. Chile: Germany is almost as wide as it is long, and important towns are distributed across the whole country, whereas Chile is very long and narrow, and the capital Santiago is by far the most important city. It is possible that our algorithm behaves very differently on other countries, but as mentioned, this is object to further study.

3.3. Searching in the Neighborhood Graph

We want to search for a pair of towns (t_1, t_2), where t_2 is a bigger town than t_1 and they are near each other in a geographical sense, i.e. t_2 is a landmark of t_1. A possible query could be `"Eberbach near: Sinsheim"`.

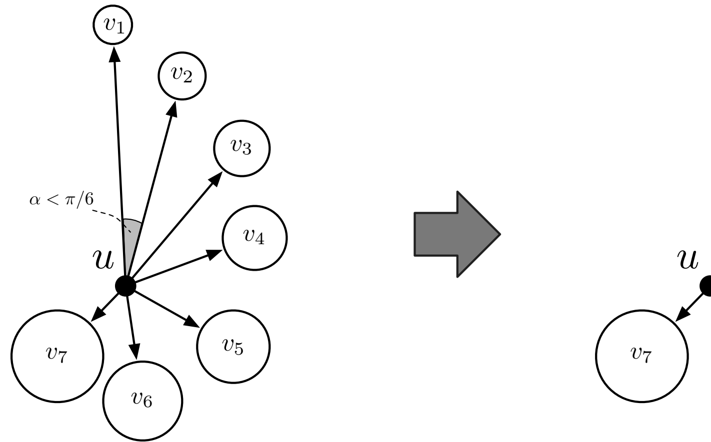


Figure 20: A graph instance where only one neighbour will survive the domination phase.

`Sinsheim` is a landmark for `Eberbach` if there is an edge (`Eberbach`, `Sinsheim`) in the neighborhood graph. We find such an edge as follows: First, we query the index for sets of candidates cand(`Eberbach`) and cand(`Sinsheim`) and find the corresponding towns $T_1 :=$ towns(cand(`Eberbach`)) and $T_2 :=$ towns(cand(`Sinsheim`)).

A pair $(x, y) \in T_1 \times T_2$ is a possible result if y is a neighbour of x, i.e. if $y \in \nu(x)$. To find possible results, we use the following algorithm:

1. Mark all towns in T_2.

2. For each $x \in T_1$, look at all neighbours $y \in \nu(x)$. If y is marked, (x, y) is a possible result.

The above example is visualized in Figure 21 on the following page.

3.4. Finding a Postal Address

In the remainder of this section we will describe how we utilize the neigborhood graph and the parent-child relation to construct a fault-tolerant address index. In the following we will use the term "error" a bit loosely. When we speak of an "error in the candidate x", we mean that it differs from the query — of course the candidates all refer to reference data, in this respect they don't actually contain errors.

The address search consists of up to five phases, the first of which is responsible for the preparation of the query string and further initialization. During the remaining four phases, the actual search is performed.

Initialization. The query string is split into tokens and normalized, i.e. transformed to lower-case, german umlauts are substituted according to the official rules (*ü* to *ue*, *ß* to *ss* ...)

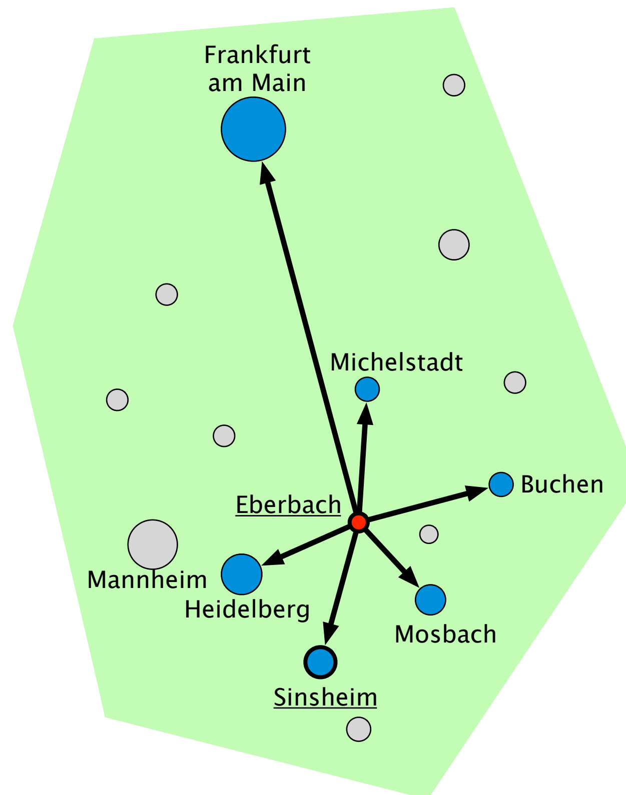


Figure 21: A neighborhood search for $(t_1, t_2) =$ (`Eberbach, Sinsheim`): `Sinsheim` matches t_2 and is therefore marked. It is then found again as a neighbour of `Eberbach`, which matches t_1. Note that `Mannheim` isn't a neighbour of `Eberbach` — it is dominated by `Heidelberg`. `Frankfurt am Main`, on the other hand, is a neighbour despite the greater distance, because there is no other important town in between.

etc. The candidate sets for the street and the town are computed, or, if the keyword *`near:`* is provided, for the street and two towns. Incompatible candidates are dropped.

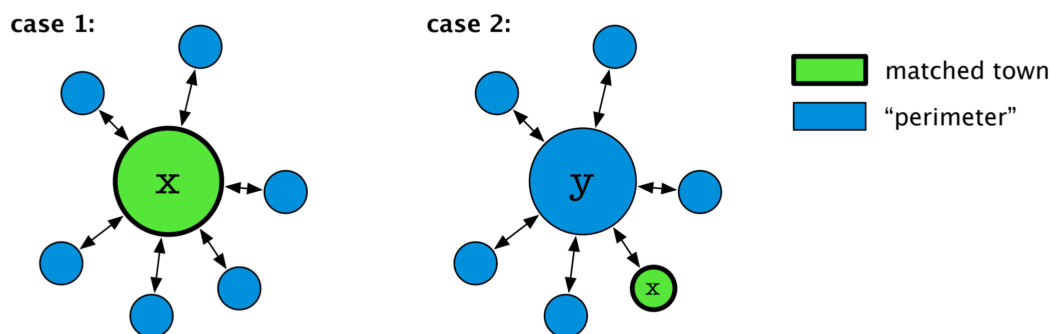
The initialization stage is obligatory, whereas the several search phases are performed only if the preceding phases did not succeed to find a good-enough match.

Neighborhood Graph Search. The user may supply two towns instead of one, separated by the keyword *`near:`*. The query `"Offnbach near: Frankfrt"` is then interpreted as "find a town that resembles the query `'Offnbach'` and has a neighbour that resembles the query `'Frankfrt'`". This means that we look for a pair of towns that is connected by an edge in the neighborhood graph. The number of pairs that have to be considered is usually small. Then we proceed to search the street inside the matching town candidates. Note that the street search, in this phase as well as in all following phases, is always fuzzy.

Usually a user of a geocoding service will want to type as little as possible. Therefore most people will probably not use this feature. However, the neighborhood graph could be very useful in an interactive setup, where the user can choose the correct town in a list of possible matches. After a user has typed enough letters to identify the town (e.g. "Musterstadt"), we could follow the edges in the neighborhood graph to show possible matches to choose from (e.g. "Musterstadt near Stuttgart", "Musterstadt near Bremen", ...), without the need to actually enter the remaining parts. Existing online geocoding services, such as Google Maps [26], already provide this feature.

Exact Town Search. In this phase we demand that one of the query tokens appear without errors in a candidate. I.e., the query "Frankfrt, Main" will match Frankfurt am Main because "Main" appears without errors in the candidate. The candidate Frankfurt, however will not be matched. We call such a match *partially exact*. We then try to identify the street inside the partially exact town matches.

Perimeter Search. If we successfully identified partially exact matches during the first phase, but could not match a street in these towns with a sufficient score, we look for the street in nearby districts (the *perimeter*, see Definition 6 on page 30). Recall that in our data each town is either a principal town or a district. If a candidate x is a principal town, we search for the street also in all districts of x. If a candidate x is a district of the principal town y, we search for the street also in y as well as in all of its districts:



If the town provided by the user is matched against a candidate x which is subsequently corrected to a town y in the same perimeter, we still calculate the rating for x.

Fuzzy Search. If we still haven't found a good enough match, we further relax the requirements for the town candidates. Now we also allow errors to occur in every token of a town candidate, but we don't do a full perimeter search. If the matched candidate is a district, we still search inside the principle town, but not inside any of the other districts.

The intention behind this separation is that we think of erroneous input as an exception. There may be errors, and then we want to correct them, but it is also possible that the user

enters the query correctly, and then there is no need to jump through all those hoops. After each phase we will evaluate the candidates and calculate a score for each of them. We only advance to the next phase if the results that have been found so far are not yet satisfying. The first phase can be omitted if the user does not provide the keyword "*near:*" and the fuzzy search will usually only be performed if there are no partially exact matches.

3.5. Pseudocode & Analysis

In this section we give the complete algorithm in pseudo-code and an analysis of the run time. The pseudo-code for the subroutine TRANSITIVECLOSURE is omitted because it is straightforward and of little interest in this context. The subroutine INSERT that is used to construct the Delaunay triangulation is based on the *QuadEdge* data structure introduced in [13]. The C++ implementation is adapted from [24]. The routines work on *nodes*, which are simple objects consisting of a *coordinate*, a *rank*, and a *neighborhood*.

The function NEIGHBORHOODGRAPH (Alg. 4 on the facing page) takes as input a set of nodes $T := \{t_1, \ldots, t_n\}$, ordered by ascending rank. It constructs the Delaunay Triangulation of T and the corresponding neighborhood graph at the same time. Finally, the edges to dominated town nodes get deleted from the neighborhood graph.

The predicate DOMINATES (Alg. 5 on the next page) takes three nodes u, v, w and an angle α and returns true if and only if w is dominated by v (see Figure 19 on page 38).

3.5.1. Analysis

We can divide Alg. 4 on the next page into three parts:

1. A single insertion step can take $\mathcal{O}(n)$ time, but in practice the average number of edges to be flipped to restore the Delaunay properties is small (< 9). Guibas, Knuth, and Sharir have shown that if the insertion order is randomized, the expected time is $\mathcal{O}(1)$ per insertion [12]. In our case, the insertion order depends on the distribution of the towns, therefore we can't apply this bound, unfortunately.

   The location of the triangle where the next node is to be inserted can be done optimally in $\mathcal{O}(\log n)$ time, but we opt for a simpler approach: We randomly select a triangle and move into the direction of the node until the correct triangle is reached. If the nodes are uniformly distributed, the expected time to find the correct triangle is $\mathcal{O}(\sqrt{n})$ [24].

   If we assume that the towns are uniformly distributed and the insertion order is randomized, then we construct the Delaunay triangulation in an expected time of $\mathcal{O}(n^{3/2})$. The preliminary neighborhood graph is computed during the construction of the Delaunay triangulation, needing only constant time in each iteration.

2. The Transitive closure can be computed by doing a breadth first search from every node in the graph that is obtained in the first phase. This takes $\mathcal{O}(nm)$ time, where m is the number of edges in the preliminary neighborhood graph. In each step, the edges

Algorithm 4: NeighborhoodGraph

Input : A set of nodes $T := \{t_1, \ldots, t_n\}$
Output: A neighborhood graph $N(T)$

/* The neighborhood graph: */
1 $V_N \leftarrow T$
2 $E_N \leftarrow \emptyset$
3 $N \leftarrow (V_N, E_N)$

/* The Delaunay Graph: */
4 $V_D \leftarrow \{p_0, p_1, p_2\}$ /* the encompassing triangle */
5 $E_D \leftarrow \{(p_0, p_1), (p_1, p_2), (p_2, p_3)\}$

/* Construct the delaunay graph and the */
/* preliminary neighborhood graph at the same time. */
6 **for** $i \leftarrow 1$ **to** n **do**
7 $\quad D \leftarrow$ Insert(t_i, D)
8 $\quad M \leftarrow neighborhood(t_i, D)$ /* t_i's neighborhood in D */
9 $\quad E_N \leftarrow E_N \cup \{t_i\} \times (M \setminus \{p_0, p_1, p_2\})$

10 $E_N \leftarrow$ TransitiveClosure(E_N)

/* Find dominated nodes and drop them. */
11 **for** $i \leftarrow 1$ **to** n **do**
12 $\quad M \leftarrow neighborhood(t_i, N)$ /* t_i's neighborhood in N */
13 $\quad$ **foreach** $u \in M$ **do**
14 $\quad\quad$ **foreach** $v \in M \setminus \{u\}$ **do**
15 $\quad\quad\quad$ **if** Dominates(t_i, u, v) **then**
16 $\quad\quad\quad\quad neighborhood(t_i, N) \leftarrow M \setminus \{v\}$

17 **return** N

Algorithm 5: Dominates

Input : Three points u, v, w and an angle α
Output: **true** iff v is located inside the sector of a circle with radius $dist(u, w)$ and angle α and if it appears brighter than w when viewed from u.

1 **if** $dist(u, v) > dist(u, w)$ **then**
2 $\quad$ **return false**

3 $b_v \leftarrow$ Brightness(u, v)
4 $b_w \leftarrow$ Brightness(u, w)
5 **return** $b_v > b_w$ **and** $\angle(\vec{uv}, \vec{uw}) \leq \alpha/2$

Algorithm 6: Brightness

Input : A node v and a distance l

Output: The brightness that is perceived at distance l from v

1 **return** $rank(v)/l^2$

inserted into the neighborhood graph are edges that are inserted into the current Delaunay triangulation. Since the total number of edges that are inserted during the construction of the triangulation is $\mathcal{O}(n)$, the preliminary neighborhood graph also has $m = \mathcal{O}(n)$ edges and the transitive Closure can be computed in $\mathcal{O}(n^2)$ time.

3. To check for a node u which neighbours can be dropped, we have to check each pair of neighbours of u. The check `Dominates(u,v,w)` runs in constant time. In the worst case, t_i has degree $i-1$, i.e. every node has an edge to every more important node. In that case, this phase takes $\mathcal{O}(n^3)$ time.

Summing all up, we can see that the neighborhood graph can be built in $\mathcal{O}(n^3)$ time, but the actual performance is better than this bound suggests: Before the domination phase the average degree in the graph is 283, as opposed to the theoretical worst case of all 12376 nodes (the principal towns) that are present in the graph.

In our experiments it takes under 7 minutes to build the graph, and actually most of the time is needed to write the graph to a file. The graph for Germany takes up 2.6 MB.

3.6. Searching in a Single Search Field

The Address search supports lookup of towns as well as postal addresses, i.e. streets paired with towns. From the programmer's point of view, the easiest way to accept such a query is to have it typed into designated text fields, i.e. a “Street:” and a “Town:” field.

From the user's points of view, however, it is more convenient to enter a query into a single text field, with street and town in arbitrary order, delimited by whitespace or a comma. This poses the additional problem of parsing the search string into an adequate format that we can further process. Specifically we would like to know which parts of the search string make up the town and which parts can be interpreted as a street name. It is also possible that the search string describes only a town and does not contain a street name, but we don't support searches for streets without an associated town.

It is sensible to assume that the substrings that make up the town as well as the street are contiguous, i.e. the town string does not contain tokens that belong to the street and vice versa.

A simple method to split a search string into a town string and a street string is simply to split at all possible positions and to repeatedly query the index. We call this an *exhaustive search*. For example, the query `"Alte Bahnhofstraße München"` can be split into (`"Alte Bahnhofstraße München"`, “”), (`"Alte"`, `"Bahnhofstraße München"`), (`"Alte Bahnhofstraße"`, `"München"`), and (`""`, `"Alte Bahnhofstraße München"`). We don't need

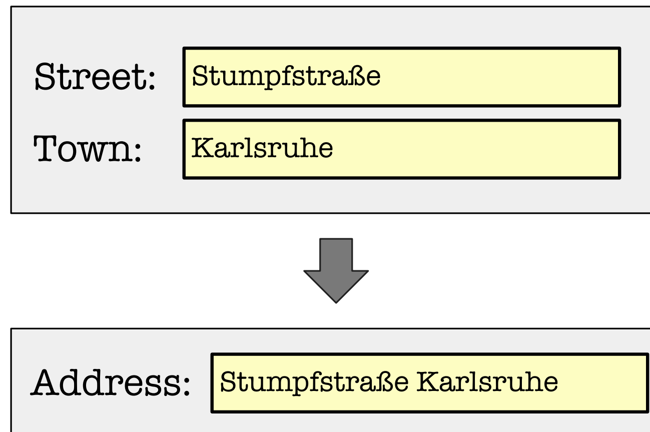
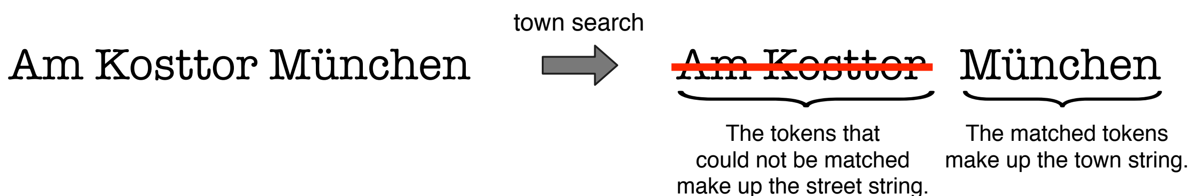


Figure 22: A single search field is more convenient for the user.

to treat the case where the town string is empty. If (s_1, s_2) is a pair of nonempty substrings, we have to perform two lookups, one where s_1 is the town and another where s_2 is the town. For a search string that consists of m tokens, we would have to perform $2(m-1)+1 = 2m-1$ lookups. We can then return the overall best result.

While this method will always find the best possible way to treat the search string, it takes much longer than before. We therefore propose another approach, that has to perform only two lookups. Since there are much less towns than streets and the town names typically consist of only one token, a town lookup is quite fast. We now perform a town lookup for the complete search string, although possibly the town makes up only a small fraction of the string. We alter the rating heuristic such that unmatched tokens in the query are not punished. During the computation of the rating we already determined which of the tokens in the search string could be matched to a candidate. Consequently we can treat the matched tokens as the town string and the unmatched tokens as the street string. Then we start another, complete address query with these strings. In most cases we should find addresses with the second,



faster approach. But there are cases where it does not work as expected. Consider the street `Durlacher Tor` in the town `Karlsruhe`. We expect the query `"Karlsruhe Durlacher Tor"` to be identified as `"`*`Town:`*` Karlsruhe; `*`Street:`*` Durlacher Tor"`. But unfortunately, `Durlach` is a subdistrict of `Karlsruhe`, hence the query will be identified as `"`*`Town:`*` Karlsruhe Durlacher; `*`Street:`*` Tor"` and will not return the expected result. Since this is a special case, it's not that tragic if the query takes a little longer. So, if there is no result or

only a result with a low rating, we restart the search and this time, we perform an exhaustive search.

3.7. Rating Address Candidates

After we have dismissed most of the search space, we are left with a hopefully small set of candidates. We still have to decide which of those candidates is the best match for the user's query. Often there will be several candidates that compete for the top position. Depending on our application, it may be feasible to return a list of candidates and let the user choose the match which suits him best. This is the case in an interactive setup where a user can directly verify if the returned result is indeed valid. But one can also imagine situations where it is unacceptible to have more than one result of the same quality. For instance, a company could have a data base of addresses in a non-canonical format and would like to correct them automatically using a form of batch-processing. The rating heuristic that we discuss here is best suited for the interactive setup.

To develop a rating heuristic, let us recall the different kinds of errors that we will encounter:

- Typing errors
- Missing or redundant tokens
- Incorrect pairing of a street and a town, i.e. a street that exists, but not in the provided town.

The first step on the way to a robust rating heuristic is to align the query to a candidate, i.e. find an optimal mapping from the tokens in the query to the tokens in the candidate.

3.7.1. Matching the Query to a Candidate

The typical query consists of a string *street* and a string *town* and will lead to a set of candidates for both street and town. Both the query string and the candidate string may consist of several tokens, i.e. substrings delimited by whitespace, hyphens etc. Any token in one of the strings may occur somewhere in the other string, with or without errors.

Consider the query `"Rotenburg (Tauber), Stainbach"` and the candidate `"Steinach bei Rothenburg ob der Tauber"`. For the human observer it is trivial to find an optimal matching: `"Rotenburg"` with `"Rothenburg"`, `"Tauber"` with `"Tauber"`, and `"Stainbach"` with `"Steinach"`. The short tokens `"ob"` and `"der"` don't occur in the query string.

What humans do in this case, probably without noticing it, is solving a well-known problem from graph theory, albeit on a very small graph. Informally speaking, we compare each token from one string with each token from the other string and decide which of them look most similar to each other. In graph terminology, we construct a Minimum Weighted Bipartite Matching, where "minimum weighted" corresponds to "most similar" in the last sentence. The problem is also known as the Assignment Problem.

Definition 13. *Given two sets A (agents) and T (tasks) and a weight function $c : A \times T \to \mathbb{R}$, find a bijection $f : A \to T$, such that the cost function*

$$\sum_{a \in A} c(a, f(a))$$

is minimized.

If there are more agents than tasks, we create "dummy tasks". If t is a dummy task, we set $c(a, t) = 0$ for each agent a. This will not change the optimal assignment. The case where there are more tasks than agents can be handled similarly.

In our case, let $Q = \{q_1, \ldots, q_l\}$ be the query's tokens and $C = \{c_1, \ldots, c_l\}$ the candidate's tokens. Then we set

$$A := Q$$
$$T := C$$

and

$$c(q, c) := \begin{cases} 0, & \text{if either } q \text{ or } c \text{ is a dummy} \\ \mathrm{ed}(q, c), & \text{else} \end{cases}$$

There are several algorithms which efficiently solve the assignment problem, such as the HUNGARIAN ALGORITHM [22] or algorithms based on linear programming, amongst others. The instances that we consider in our application are rather small, the average number of tokens in a query or candidate does not exceed 4.

Based on the matching, we calculate a rating for each candidate. Suppose the underlying index has been computed for up to d errors. The rating should take into account the following considerations:

- Each token that matches with at most d errors should be awarded some points.
- 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 $[0, 1]$, thus making it easy to compare different candidates.
- The heuristic should be able to distinguish between tokens that are "important" and tokens that do not provide much information.

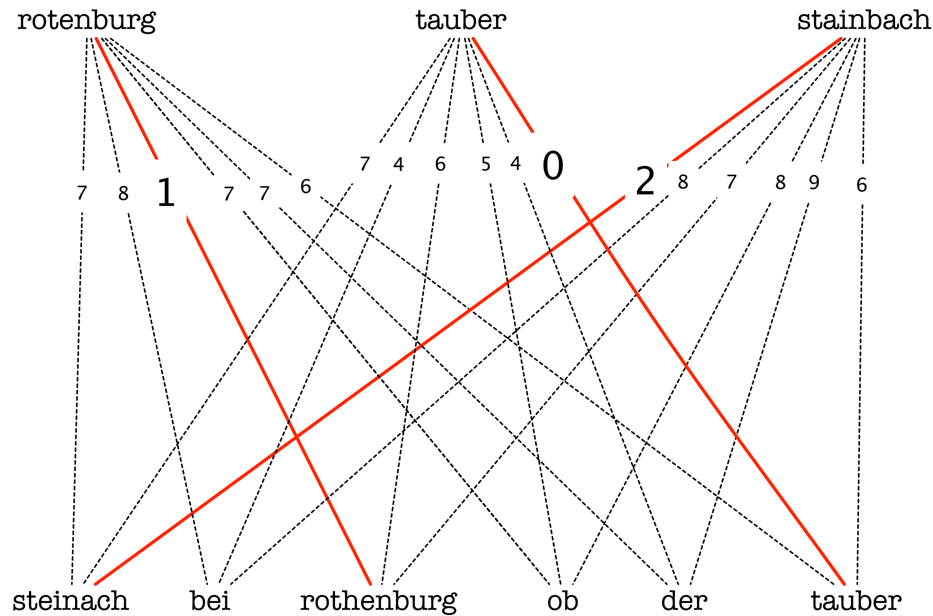


Figure 23: An optimal assignment of the query `"Rotenburg (Tauber), Stainbach"` to the candidate `"Steinach bei Rothenburg ob der Tauber"`. Dummy nodes are omitted.

3.7.2. The Rating Heuristic

We will now define the rating heuristic that we will use to choose the best match among the candidates. Our index supports the correction of errors up to a constant d, independent on the length of the considered word. We also want to take into account the lenghts 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.

Definition 14. *The* token similarity *between two tokens q and c is*

$$\text{sim}(q,c) := 1 - \text{err}(q,c),$$

where

$$\text{err}(q,c) := \begin{cases} \frac{\text{ed}(q,c)}{|c|}, & \textit{if } \text{ed}(q,c) \leq d \\ 1, & \textit{else} \end{cases}$$

The candidates are entries that are actually present in our data base, while the query may contain errors. Therefore we treat c as the "reference token" and normalize the error rate by the length of c.

$q :=$ "Stainbach", $c :=$ "Steinach"

$$\text{sim}(q, c) = 1 - \frac{\text{ed}(q, c)}{|c|} = 1 - \frac{2}{8} = 3/4$$

Example 3: The tokens "Stainbach" and "Steinach" are quite similar.

Let us now discuss what we mean by "important" tokens. In textual descriptions of an entity, it is usually only a subset of the tokens that refers directly to the described entity, while other parts of the description are made up of articles, prepositions etc. E.g. the description "Neustadt an der Weinstraße" refers to a town in Germany that could be unambiguously described by the tokens "Neustadt" and "Weinstraße", whereas "an der" would not suffice. It is striking that prepositions etc. are usually very short words in most languages, but we cannot simply link the importance of a word to its length, as there are lots of exceptions to that observation. However, a possible reason why these words are so short may be that they are used so frequently, therefore it seems a sensible assumption to base a token's importance on its frequency within the data. This concept has been used successfully in the domain of information retrieval under the term *inverse document freqency* (IDF) [3, 5]. We use the formula

$$\text{IDF}(t) := \log \frac{|T|}{\text{freq}(t)}$$

to denote the IDF of a token t in relation to the set T of tokens in a dictionary, where $\text{freq}(t)$ is the number of occurrences of t in that dictionary. Note that we will compute the IDF only on candidate tokens, hence $\text{freq}(t)$ will always be greater 0.

$|T| = 120000$, freq("Rothenburg") $= 4$, freq("der") $= 293$,

$$\begin{aligned}\text{IDF}(\texttt{"Rothenburg"}) &= \log \frac{120000}{4} \\ &= 10.31 \\ \text{IDF}(\texttt{"der"}) &= \log \frac{120000}{293} \\ &= 3.71\end{aligned}$$

Example 4: The token "der" occurs much more frequently in our town data base than the token "Rothenburg". Therefore we interpret "Rothenburg" as more important.

Given a query Q, we would like to determine the importance of each token $q \in Q$ in relation to the rest of the query. This is obviously only possible if q occurs also in T, or if it can be matched against a token that is present in T. To this end, let

$$M_{\leq d} := \{(q, c) \in Q \times C \mid q \text{ matches } c \text{ with at most } d \text{ errors}\}$$

be the set of pairs that could be matched given a query Q and a candidate C, and let

$$M^{Q}_{>d} := \{q \in Q \mid q \text{ does not match anything with at most } d \text{ errors}\}$$

be the set of query tokens that could not be matched to anything in the candidate with less than d modifications. Likewise, we define $M^{C}_{>d}$ to be the set of candidate tokens that could not be matched to the query.

For our rating heuristic we want to compute the importance of the matched query tokens in relation to the tokens that could not be matched. We define the *weight* of a token q as

$$w^{Q}_{C}(q) := \begin{cases} \text{IDF}(c), & \text{if } (q, c) \in M_{\leq d} \\ \text{IDF}_{\text{avg}}, & \text{if } q \in M^{Q}_{>d}, \end{cases}$$

where IDF_{avg} denotes the average weight of all tokens in T.

The rating for the query Q and a candidate C is then given by

$$\text{rating}^{Q}(Q, C) := \left(\sum_{(q,c) \in M_{\leq d}} (\text{sim}(q, c))^{\alpha} w^{Q}_{C}(q) \right) / \left(\sum_{(q,c) \in M_{\leq d}} w^{Q}_{C}(q) + \sum_{q \in M^{Q}_{>d}} w^{Q}_{C}(q) \right)$$

The parameter α is used to adjust how strongly errors are punished. In our experiments we set $\alpha := 2$. Notice that the rating is not influenced by the tokens in $M^{C}_{>d}$: For the query "`Frankfurt`" we also want a very good rating for the candidates "`Frankfurt am Main`" and "`Frankfurt an der Oder`". On the other hand, if the query is "`Frankfurt an der Oder`", we want the candidate "`Frankfurt an der Oder`" to receive a better rating than the candidate "`Frankfurt`". If we use this rating heuristic to sort candidates by their quality in relation to a candidate, the missed tokens in the candidate are not taken into account at all, in other words the order of "`Frankfurt`", "`Frankfurt an der Oder`" and "`Frankfurt am Main`" is undefined. Therefore we define a second rating

$$\text{rating}^{C}(Q, C) := \left(\sum_{(q,c) \in M_{\leq d}} \text{IDF}(c) \right) / \left(\sum_{c \in C} \text{IDF}(c) \right),$$

where we divide the weight of the matched tokens by the total weight of the candidate. The complete rating is then given by

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

Usually we want to give more weight to the matched parts of a query, therefore we choose $\gamma > 1/2$. In our experiments we chose $\gamma := 3/4$. The rating lies always between 0 and 1. We can use this heuristic to sort the candidates according to their quality, and return only the best rated candidates.

`"Frankfrut/Mein Innenst."` ↔ `"Frankfurt am Main"`

$$Q := \{\texttt{"frankfrut"}, \texttt{"mein"}, \texttt{"innenst"}\}, C := \{\texttt{"frankfurt"}, \texttt{"am"}, \texttt{"main"}\}$$

frankfrut — 2 — frankfurt
mein — 1 — main
~~innenst~~
~~am~~

Token	Frequency	IDF
frankfurt	3	10.31
am	507	5.18
main	34	7.89
avg	1.19	11.24

$$\begin{aligned} \text{rating}^{Q}(Q,C) &= ((7/9)^2 \cdot 10.31 + (3/4)^2 \cdot 7.89)/(10.31 + 7.89 + 11.24) \\ &= 10.68/29.44 = 0.36 \\ \text{rating}^{C}(Q,C) &= (10.31 + 7.89)/(10.31 + 7.89 + 5.18) \\ &= 0.78 \\ \Rightarrow \text{rating}(Q,C) &= 3/4 \cdot 0.36 + 1/4 \cdot 0.78 = 0.47 \end{aligned}$$

Example 5: The tokens `"bei"`, `"ob"` and `"der"` are not as important as the other tokens.

3.7.3. Ignoring Light Candidates

Whatever trick we use to speed-up the query, the slowest queries are always those where there are lots of candidates to be rated and verified. Consider the example query `"Alte Waibstadter Straße"`. Recall that the index will return as a candidate any entry that matches any of these words. `"Waibstadter"` will match only a handful of candidates, but the frequent tokens `"alte"` and `"straße"` will completely blow up the candidate set.

Since we weight the tokens with IDF weights, the frequent occurence of those two tokens also means that their importance is insignificant in comparison to the more descriptive token `"waibstadter"`.

To solve this problem, we decide not to insert those unimportant tokens into the index. Consider the example from above, `"Alte Waibstadter Straße"`. The relative weights of the tokens are 0.31, 0.6 and 0.09, respectively. If we decide that the query must contain tokens that make up a fraction μ of the total weight, then we can ignore the lightest k tokens, if the sum of their weights is not greater than μ. E.g., for $\mu > 0.4$, we can ignore the tokens `"Alte"` and `"Straße"`. In a query, there must be a token that matches against `"Waibstadter"`,

otherwise we don't return "Alte Waibstadter Straße" as a candidate, and this is also the desired behaviour. Indeed, we need this modification if we apply the rating heuristic as explained above. The heuristic places emphasis on the matched query tokens, therefore a query consisting solely of the token "straße" would receive a rating of at least γ when matched against "Alte Waibstadter Straße". Ignoring the light tokens avoids this, and we don't need to alter the rating heuristic.

Note that it depends on the candidate in question if we ignore a token or not. In the candidate "Alte Waibstadter Straße", the tokens "alte" and "straße" are negligible, but in the candidate "Alte Straße" the token "alte" is much more important than the token "straße", therefore we keep it.

The described modification has a great impact on the performance of some queries, especially those that were previously very slow. The reason is simply that we don't have to rate that many irrelevant candidates.

3.8. Experiments

In this section we will evaluate experiments we conducted on the address index to measure query times, the match rate, memory requirements etc. There are some parameters that affect the behaviour of the index and we will try to give sensible default values that lead to good query times without blowing up the index memory-wise.

The data that was used to build the index comprises the german street and town names. In total there are about 80500 distinct town names and 444000 distinct street names. A street name consists of 2.5 tokens on average, while town names consist of 1.1 tokens on average.

The queries are either randomly generated, or are logged data from users of an existing geocoder. Experiments were performed on an Intel Core i7 920@2.67GHz with 12GB RAM on Linux 2.6.27. The programs were implemented in C++ and compiled with GCC 4.3.2.

3.8.1. Random Queries

For the random queries, it would be easiest to insert, delete or substitute random characters in a query. 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 try to introduce more “realistic errors” that can roughly be divided into three classes:

Typing 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 QWERTZ-layout).

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

Phonetic Errors. Depending on the respective language there are several sources of error:

- doubled characters where there should be a single character

  Example: `"Bensheim"` → `"Bennsheim"`

- single character where there should be two of the same

  Example: `"Mannheim"` → `"Manheim"`

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

  Example: `f` ≡ `v`, `"Vechta"` → `"Fechta"`

- Two consecutive vowels that occur in the same syllable are called a *diphthong*. In the german language, several different diphthongs sound the same or similar:
  - `ei` ≡ `ey` ≡ `ay` ≡ `ai`
  - `eu` ≡ `äu` ≡ `oy` ≡ `oi`
  - ...

  Example: `Hoyerswerda` → `Heuerswerda`

We use a function *distort* that takes a string s and an integer n and randomly introduces n phonetic/typing errors into s.

$$
\begin{aligned}
&\text{distort}(\texttt{Waldstrasse}, \texttt{Mainz}, 2)\\
&\rightarrow \text{distort}(\texttt{Wasldstrasse}, \texttt{Mainz}, 1)\\
&\rightarrow \text{distort}(\texttt{Wasldstrasse}, \texttt{Meinz}, 0)\\
&\rightarrow \texttt{Wasldstrasse}, \texttt{Meinz}
\end{aligned}
$$

Example 6: The function *distort* randomly introduces errors into a string.

For our experiments, we use a set of existing, *relevant* addresses R, and a set of non-existing, *irrelevant* addresses I, which are composed of randomly chosen town and street names. Ideally, we would like to return correct results for relevant address queries and no result for irrelevant address queries. Light candidates were ignored during the indexing phase,

as described in Section 3.7.3 on page 51: The rating heuristic has been developed with that modification in mind and does not work well without it. Hence, we don't report match rates for the index that is obtained without it.

For the tests, we performed searches with 1000 relevant and 100 irrelevant addresses. We classify the results returned by the index as follows:

- *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 depicted in Table 3 on the next page. All tests were performed for both the single-field search and the multi-field search. In the case of the single-field search, we resorted to the exhaustive search, as described in section 3.6 on page 44 (this decision will be explained in Section 3.8.4 on page 59). First of all, note that the query times decrease as we increase the error rate. The reason for that is simple: The higher the number of errors, the lower the number of candidates that have to be verified. The number of true positives is practically the same for both the multi-field search and the single-field search and drops significantly at 5 errors. At that point, three errors are introduced into the street name and the edit distance becomes too big. While the single-field search doesn't produce too many false positives, the multi-field search is maybe too aggressive in its effort to interpret the query and returns up to 48 false positives for 100 irrelevant address queries. The relevant queries that can not be successfully recovered are mostly false negatives (i.e. not found at all), but sometimes a false result is returned.

3.8.2. Real 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 that is in use at PTV. We will call that geocoder PTV-GEO. The test data consisted of 1383 queries that were given to PTV-GEO as well as the output that it returned. PTV-GEO accepts input of a street and a town in separate fields and supports lookups of street/town pairs and towns without an associated street. It classifies the queries according to the quality of the achieved results into five classes:

	relevant			irrelevant		
Errors	**TP**	**FN**	**II**	**TN**	**FP**	**Time (ms)**
0	1000	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

(a) multiple fields

	relevant			irrelevant		
Errors	**TP**	**FN**	**II**	**TN**	**FP**	**Time (ms)**
0	1000	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

(b) single field

Table 3: Matching rates and query times for distorted addresses. The test queries comprise 1000 relevant and 100 irrelevant addresses. Hence we get $TP + FN + II = 1000$ and $FP + TN = 100$ in each row.

Exact. Queries where each token can be matched without errors to a token in the result. The only differences allowed between query and result are those that are handled during a normalization phase (e.g. `"str."` ≡ `"straße"` or `"ö"` ≡ `"oe"`).

Example: `"Groebenzell; Parkstr."` → `"Gröbenzell; Parkstraße"`

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

Example: `"Bergisch Gladbach; Kaule"` → `"Bergisch Gladbach; Auf der Kaule"`

High/Medium/Low. Queries that contain errors. PTV-GEO assigns the labels “High”, “Medium”, and “Low”, depending on how confident it is that the result is a correct interpretation of the input.

Example: `"Fielderstedt; kleiststr."` → `"Bonlanden, Filderstadt; Kleiststraße"` (classified as “Medium”)

We tested our address search with those queries. Unfortunately we didn't have correct reference results, hence it was not generally possible to check automatically if our results were correct, and it had to be done by hand. We classified 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.

Example: `"erfurt; rwichartstraße"` → `"Erfurt; Reichartstraße"`

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

Example: `"Nurnberg; Hinterer Floßanger"` → `"Nürnberg"`

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

Example: `"bad orp"` → `"Diemelstadt Orpethal"`

In total the test data consisted of 1383 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 and removed those that were not present in the data (e.g. searches 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 4. The match rates for exact or partially exact queries are unsurprisingly very high for

		PTV-GEO			Our Geocoder		
Class	**# Queries**	**strong**	**weak**	**nothing**	**strong**	**weak**	**nothing**
Exact	100	100	0	0	100	0	0
Partial	99	99	0	0	99	0	0
High	81	78	2	1	77	2	2
Medium	34	17	14	3	27	6	1
Low	15	9	3	3	13	1	1

Table 4: The match rates of our geocoder on real queries compared to PTV-GEO.

both geocoders. Our geocoder gives better results for erroneous queries. For those queries PTV-GEO tends to omit the street and return only the town (i.e. lots of weak matches). While these results certainly look promising, they might be even better in practice: Unfortunately we only had test queries where PTV-GEO successfully returned a (sometimes incomplete or incorrect) result. Queries that could not be answered at all were not present in the logged files that we had at our disposal.

3.8.3. Parameters That Affect the Query Time

The parameters that affect the query time are those that define the behaviour of the underlying index as described in section 2 on page 8: The size of the table where we store the hashed residual strings and the split parameter. Fig. 24 shows the effect of the split parameter on the town index. As expected, the split parameter provides a trade-off between query-time and memory usage. Splitting above length 9 reduces the size of the index file by factor 2, while the query time goes up by factor 3.

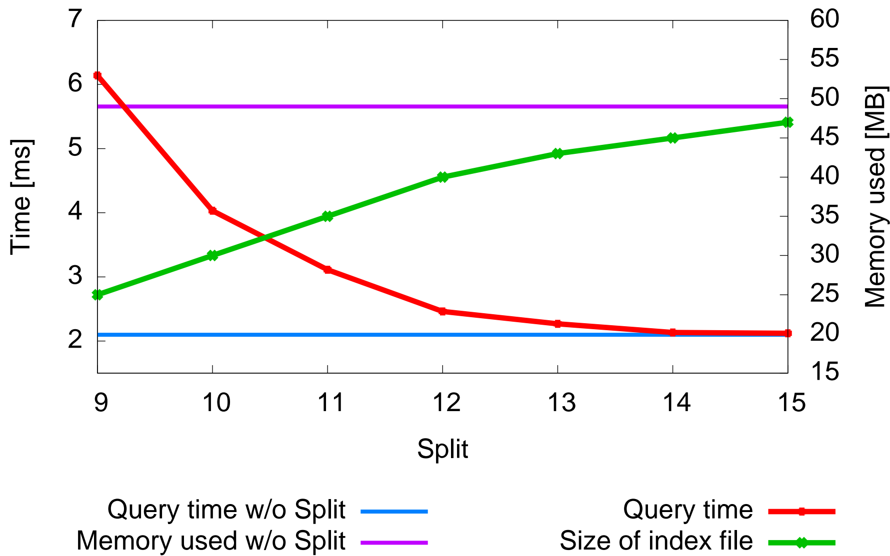


Figure 24: The average query time under different values for the split parameter. Here, the street index did not use the split parameter, while the values for the town index are shown as the x-axis.

As observed in the last section, the query times decrease when the error rate increases. Table 5 on the next page shows the average lookup time for different types of queries along with the number of candidates that have to be verified. To see why the number of candidates drops as we add errors to a query, we want to look at an example. Consider the string `"word"` in an english dictionary. If this string is input correctly, it can be transformed into other words easily by a single operation: `"word"` can become `"sword"`, `"lord"`, `"words"`, `"worm"`, `"world"` etc. If we introduce an error into the string and input the query `"womrd"`, for example, only few strings will match the query. We effectively trim the result set by introducing errors into the query, hence the query is fastest when there are errors in both town and street.

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 row of experiments. The techniques are:

Error	Query Time [ms]	Town Candidates	Street Candidates
none	2.70	73	660
street only	2.03	73	470
both	1.52	46	470

Table 5: The average query time for clean queries, queries where the street contains errors, and queries where both the town and the street contain errors.

Filter Incompatible Candidates (FIC). As described in section 3.1 on page 32, we keep only those town and street candidates that are geographically compatible.

Filter by Edit Distance (FED). The number of town candidates is typically much smaller than the number of street candidates and it is not too expensive to filter out the candidates that cannot be part of a successful match because of their edit distance to the query. We do this before we search for streets in the towns that correspond to the town candidates.

Ignore Light Tokens (ILT). As described in section 3.7.3 on page 51 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 `"Alte Waibstadter Straße"` will be represented only by `"Waibstadter"`, 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.

ILT	FIC	FED	Query Time [ms]
×	×	×	570
×	×	✓	566
×	✓	×	199
×	✓	✓	126
✓	×	×	10.68
✓	×	✓	10.58
✓	✓	×	3.45
✓	✓	✓	2.09

Table 6: The effect of the features ILT, FIC and FED on the lookup time.

As we can see in table 6, disabling ILT absolutely destroys the performance. To see why this is so, consider the most frequent token in the street dictionary, `"straße"`. If we randomly choose a street, the probability that it contains the token `"straße"` is about 1/3. Without ILT, *any* query that contains the token “straße” will return *all* candidates that contain this token. Hence, if we randomly choose a candidate 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 50000

candidates for our data. In our experiments the actual number of candidates was even bigger, 67000 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.

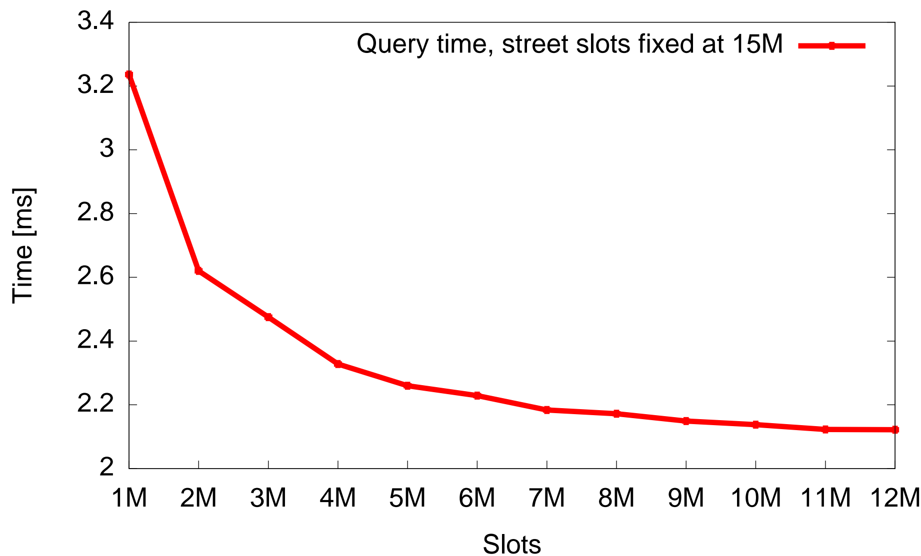


Figure 25: The effect of the size of the array `first_edge`. The red line shows the query time with different values used in the town index, while the table in the street index was chosen "big enough". The street index shows the same behaviour.

Fig. 25 shows how the query time is affected by the size of the table for the residual strings. We can see that the effect is not overly dramatic. Both the town index and the street index show the same behaviour. If we set both indexes to 1M slots, we still get an average query time of 4.2 ms.

3.8.4. Single Field Search

In Section 3.6 on page 44 we described two ways to perform a search from a single address string that contains both the street and the town. Parsing the search string is a significant complication. It is often difficult to decide which parts of the string make up the street and which parts make up the town. Our "clever" approach is to first identify the town and then split the string into two fields based on this guess. If the result is not convincing, we perform an exhaustive search (see Section 3.6 on page 44).

Query: `"Straße der 17 Juli Brlin"`
Correct Result: `"Straße des 17. Juni, Berlin"`
Our guess for the town: `"An der Straß"` (which is an actual town)

→ Search for `"Straße der"` (town), `"17 Juli Brlin"` (street)
→ No match
→ Exhaustive search leads to the expected result.

Example 7: In some cases it is difficult to identify which parts of the query make up the town.

On a sample of 1000 queries with 2 errors per query, as decribed in Section 3.8.1 on page 52, the multi-field search correctly identifies 98.8% of the addresses and takes 2.44 ms on average. The exhaustive single-field search correctly matches 98.6% of the queries in 19.72 ms, while our "clever" approach matches 97.2% in 20.01 ms. Surprisingly it takes longer than the naïve, exhaustive search. This suggests that the results are often not satisfactory — we then fall back to the exhaustive search. Remember that a query with m tokens has to be repeated for $2m - 1$ interpretations of the search string. An address string is composed of 3.6 tokens on average, and the increase of the lookup time from 2.44 ms to 19.72 ms is about what we expected.

Unfortunately, the described "short circuiting" is not of much use. The question remains how we could robustly parse the query string without having to try too many combinations and without sacrificing accuracy.

3.8.5. Searching in the Neighborhood Graph

To test the Neighborhood search (as described in Section 3.3 on page 38), we picked 50 pairs of towns from a map of Germany. The query strings were of the form `"small town` ***`near:`*** `big town; street"` and the queries have been performed both as multi-field lookups and as single-field lookups. We did not introduce further errors into the string.

The average lookup time was 4.1ms in the multi-field search and 43.4ms in the single-field search. As the addresses did not contain errors, each query was successful in the multi-field search. In 6 of the 50 cases, however, no connection from the small town to the big town could be found, despite their closeness. For instance, the query `"Leingarten` ***`near:`*** `Heilbronn; Ebertstraße"` returned the result `"Leingarten, Großgartach; Ebertstraße"` instead of `"Leingarten, Großgartach near Heilbronn; Ebertstraße"`, although the two towns are only 7km apart. The single-field search correctly identified 49 of the 50 addresses. Sample queries can be found in Appendix A.2 on page 67.

4. Open Problems

The geocoder presented here has its fair share of quirks that need to be addressed. Some of the problems are best explained with the help of an example. Ideally, the address search should work equally well on diverse address schemes in several countries without the need to customize it for a certain language. As an example for the adaptation to different languages, we want to look at how abbreviations are treated. The abbreviation `"St."` conforms to `"Sankt"` in german and to `"Saint"` in french. We don't use language-specific substitution tables, hence our index is not aware of the meaning of the string `"St."`.

Now consider the query `"St. Peter-Ording"`, which refers to the german town `Sankt Peter-Ording`. Our address search actually returns another result first, namely the town/street-pair `St. Peter, Nordring`.

This suggests either that we need language-specific rules to some degree, or that we have to develop a more sophisticated rating heuristic.

Another problem lies in the way we choose our results. If there are lots of good matches, it is usually sufficient to return those with the best rating. In the case of a bad query, however, it is difficult to decide when it becomes "too bad", i.e. when we should drop the results altogether. In the end, the user should always be able to see the resemblance of the query and the result.

5. Conclusion & Future Work

In this work we presented an approach to the problem of fault-tolerant geocoding of postal addresses. We implemented a fast index data structure to support quick approximate lookups in dictionaries and developed a simple technique to trade off lookup time for space. We showed through experiments that our index works indeed quite well on natural language dictionaries.

The index was then used to implement a fault-tolerant geocoder that supports the correction and lookup of postal addresses within few milliseconds. To achieve accuracy and fast query times, we applied filtering techniques based on the spatial distribution of the underlying data as well as on well-established methods from the domain of information retrieval. A comparison with an existing commercial geocoder showed that our application is competitive.

There is of course room for future work on both the index data structure and the address search. The index is fast, but its memory requirements are substantial. In our application, the dictionaries are not too big, therefore we can afford the space. The address index for Germany fits into ~200MB, which is quite reasonable considering the specs of today's servers and even desktop computers. But the question remains if there is a way to significantly reduce the space requirements without sacrificing query times. This is of interest especially when thinking of mobile devices. The main memory of mobile phones is typically much smaller than that of a desktop PC, and the extensive use of hashing makes our index a less then perfect fit for mobile devices. Nonetheless, the address search could work well on mobile devices if we used a smaller, less memory-intensive index.

An obvious modification would be to parallelize parts of the index. Many tasks are partly or completely independent, such as edit distance computations, the generation of residual strings etc.

For the address search, we used two indexes: A town index and a street index. This way we could answer queries for postal addresses quickly. We accept user input in a single search field and try to find out which part of the query makes up the town and which part makes up the string. Parsing the search string would be much more difficult if we deciced to support other spatial entities, such as points of interests, forests, lakes etc. It would be interesting to see if it is sufficient to use a single index that holds all entities, i.e. streets, towns etc. Then it would be possible to allow much broader queries such as queries for street intersections or points of interests (e.g. `"school in city center"`) without the need to find the correct assignment of parts of the string to the correct index.

The lookup times of our address index are very fast, especially when considering the common use case of an interactive search. We need only a few milliseconds to answer a query, which is only a small fraction of the time a user needs to enter the query on a keyboard. It should be possible to start some of the work already while the user is still typing, maybe it would even be possible to show suggestions based on the incomplete input.

The rating heuristic that is used in our address search is quite simple and easy to calculate. But of course it is not competitive with a human operator in terms of the accuracy with which

results are matched to the query string. Humans who are accustomed to a language and a local address scheme are indeed hard to beat: They have background knowledge about the spatial distribution of towns, phonetic idiosyncrasies of their language, abbreviations and naming conventions etc. It would therefore be interesting if there is a small subset of simple rules that captures enough of this background knowledge to give comparable results.

A. Sample Queries

A.1. Randomly Distorted Queries

No errors

```
Holderstraße;Lüntorf
Sellenahne;Heckershausen
Bösings Kamp;Coesfeld
Beim Schlössle;Amtzell
Gewerbegebiet an der Morgensonne;Geyer
```
...

1 Error

```
holderstraß;lüntorf
qellenahne;heckershausen
bosings kamp;coesfeld
ebim schlössle;amtzell
gewerbegebiet an der morfensonne;geyer
```
...

2 Errors

```
holdestraße;püntorf
sellneahne;heckershqusen
bödings kamp;clesfeld
beim schlösle;amtzelk
gewerbegebiet an der morgensone;geyewr
```
...

3 Errors

```
holdersdstraße;lpntorf
ssellenane;heckersausen
böings kamo;ccoesfeld
beim schhlösle;amtell
gewebregefiet an der morgensonne;geaer
```
...

4 Errors

```
holderrrstraße;lünntorw
sellenwanhe;hreckershhausen
```

```
böslings amp;fcoesffeld
beiim schlösle;amtozll
gewerbegebiet an der morgensojmnne;geyro
...
```

5 Errors

```
holertraeß;lüntoofr
eselldnahne;hecekrshauken
bösingw kuanp;coeesefld
bveim shclösslee;amtzeel
gewerbogerbiet an der morgenslonne;gbeyerr
...
```

A.2. Logged Queries

Exact Search

```
passau;neue rieser str.
leipzig-thekla;theklaer-str.
verl kaunitz;fürstenstraße.-
berlin mitte;schumannstraße
pforzheim;lukas-moser-str.
marburg;körnerstraße
simmertal;
remscheid;
gelsenkirchen;theodor otte strasse
markt schwaben;
...
```

Partially Exact Search

```
hamburg;sieldeich -
bonn;theaterstr.
blumberg;elisenauer str
kattenes;moselufer
schwabach;grünewaldstraße
köln;mannsfelder str
bochum;am chursbusch
hamburg;wrangelstr.
berlin;dörpfeldstr.
herzebrock;möhlerstraße
...
```

"High"

```
dortmund;walther-kohlmann-sraße
karlsruhe;momberstr.
villingen schwellingen;
hasbergen an;osnabrücker straße
frankfurt;lyonerstr.
esslingen am neckar;berkheimerstr.
hamburg lohbrügge;agnes-wolffson-straße
hannover;hildesheimerstr
schneeberg;teich-straße
schondorf;uttingerstr.
```
...

"Medium"

```
bernburg saale;
rednitzhembach;königsplatz
burgwedel großburgwedel;furberst
damp;yachthafen
aachen;philippe nehri weg
eutin;am hauptbahnhof
langförden;in der praterei
nürnberg;hinterer floßanger
laustiz;
damp;aqua tropicana
```
...

"Low"

```
bad homburg;bad homburg
oschsenhausen;kolpingstr.
mülheim an der ruhr;nollendorfstraße
hÃ¼nstetten;am sÃ¼dhang
travemünde;helling
klettwitz;lausitzallee
berlin;sigfried str höchenschonchausen
wupperthal;wupperthal hauptbahnhof
düren;a
bonn;auf der minne
```
...

Neighborhood Search

```
Köditz bei: Hof;Birkenweg
Ötigheim bei: Bietigheim;Kreuzstraße
Wörth bei: Karlsruhe;Mercedes Benz
Sindelfingen bei: Stuttgart;Maichinger Str
Aarbergen bei: Wiesbaden;Untere Weinbergstraße
Dülmen bei: Münster;Dammweg
Duderstadt bei: Göttingen;Am Stadtberg
Teistungen bei: Duderstadt;Bergstraße
Sinsheim bei: Heidelberg;Longue-Straße
Parchim bei: Schwerin;Südring
```

…

References

[1] PTV Planung Transport Verkehr AG. Stumpfstraße 1, 76131 Karlsruhe.

[2] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, June 1975.

[3] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. Modern information retrieval, 1999.

[4] W. A. Burkhard and R. M. Keller. Some approaches to best-match file searching. *Commun. ACM*, 16(4):230–236, April 1973.

[5] Surajit Chaudhuri, Kris Ganjam, Venkatesh Ganti, and Rajeev Motwani. Robust and efficient fuzzy match for online data cleaning. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, page 313–324, New York, NY, USA, 2003. ACM.

[6] Edgar Chávez, Gonzalo Navarro, Ricardo Baeza-Yates, and José L. Marroquín. Searching in metric spaces. *ACM Comput. Surv.*, 33(3):273–321, September 2001.

[7] Fred J. Damerau. A technique for computer detection and correction of spelling errors. *Commun. ACM*, 7(3):171–176, 1964.

[8] Vlastislav Dohnal, Claudio Gennaro, Pasquale Savino, and Pavel Zezula. D-index: Distance searching index for metric data sets. *Multimedia Tools Appl.*, 21(1):9–33, 2003.

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

[10] Pierce Eichelberger. The importance of addresses: the locus of gis. *Proceedings of the GIS/LIS '94 Annual Conference and Exposition*, 1993.

[11] Daniel W. Goldberg, John P. Wilson, and Craig A. Knoblock. From text to geographic coordinates: The current state of geocoding. *Journal of the Urban and Regional Information Systems Association*, 19(1):33–46, 2007. http://urisa.org/goldberg.

[12] Leonidas Guibas, Donald Knuth, and Micha Sharir. Randomized incremental construction of delaunay and voronoi diagrams. *Algorithmica*, 7:381–413, 1992. 10.1007/BF01758770.

[13] Leonidas Guibas and Jorge Stolfi. Primitives for the manipulation of general subdivisions and the computation of voronoi. *ACM Trans. Graph.*, 4(2):74–123, April 1985.

[14] Dan Gusfield. *Algorithms on strings, trees, and sequences : computer science and computational biology*. Cambridge Univ. Press, January 1997.

[15] http://www.gutenberg.org. Project gutenberg.

[16] Tanuja Joshi, Joseph Joy, Tobias Kellner, Udayan Khurana, A Kumaran, and Vibhuti Sengar. Crosslingual location search. In *SIGIR '08: Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval*, page 211–218, New York, NY, USA, 2008. ACM.

[17] Daniel Karch, Dennis Luxen, and Peter Sanders. Improved fast similarity search in dictionaries. *SPIRE*, 2010.

[18] Juha Kärkkäinen and Joong Chae Na. Faster filters for approximate string matching. In *ALENEX*, 2007.

[19] N Krieger. Overcoming the absence of socioeconomic data in medical records: validation and application of a census-based methodology. *Am J Public Health*, 82(5):703–10, 1992.

[20] 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(5):471–482, Sep 2002.

[21] N Krieger, P Waterman, K Lemieux, S Zierler, and JW Hogan. On the wrong side of the tracts? evaluating the accuracy of geocoding in public health research. *Am J Public Health*, 91(7):1114–1116, 2001.

[22] H. W. Kuhn. The Hungarian method for the assignment problem. *Naval Research Logistic Quarterly*, 2:83–97, 1955.

[23] Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. Technical Report 8, 1966.

[24] Dani Lischinski. Incremental delaunay triangulation. page 47–59, 1994.

[25] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. In *SODA '90: Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, page 319–327, Philadelphia, PA, USA, 1990. Society for Industrial and Applied Mathematics.

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

[27] Herman Melville. *Moby Dick*, volume 15. 1991. Missing Chapter 72.

[28] M. Mor and A. S. Fraenkel. A hash code method for detecting and correcting spelling errors. *Commun. ACM*, 25(12):935–938, 1982.

[29] Girish Motwani and Sandhya G. Nair. Search efficiency in indexing structures for similarity searching. *CoRR*, cs.DB/0403014, 2004.

[30] E. W. Myers. A sublinear algorithm for approximate keyword searching. *Algorithmica*, V12(4):345–374, October 1994.

[31] Gene Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *J. ACM*, 46(3):395–415, May 1999.

[32] Gonzalo Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33, 1999.

[33] Gonzalo Navarro and Ricardo Baeza-yates. A hybrid indexing method for approximate string matching. *Journal of Discrete Algorithms*, 1:2000, 2001.

[34] Gonzalo Navarro, Ricardo Baeza-yates, Erkki Sutinen, and Jorma Tarhio. Indexing methods for approximate string matching. *IEEE Data Engineering Bulletin*, 24:2001, 2000.

[35] Andreas M. Olligschlaeger. Artificial neural networks and crime mapping, 1998.

[36] Jerry H. Ratcliffe. Research article.

[37] Ricardo and Gonzalo Navarro. Fast approximate string matching in a dictionary. In *String Processing and Information Retrieval*, page 14–22, 1998.

[38] Duangduen Roongpiboonsopit and Hassan A. Karimi. Comparative evaluation and analysis of online geocoding services. *International Journal of Geographical Information Science*, 24(7):1081–1100, 2010.

[39] Vibhuti Sengar, Tanuja Joshi, Joseph Joy, Samarth Prakash, and Kentaro Toyama. Robust location search from text queries. In *GIS '07: Proceedings of the 15th annual ACM international symposium on Advances in geographic information systems*, page 1–8, New York, NY, USA, 2007. ACM.

[40] B. Stiller T. Bocek, E. Hunt. Fast Similarity Search in Large Dictionaries. Technical Report ifi-2007.02, Department of Informatics, University of Zurich, April 2007. http://fastss.csg.uzh.ch/.

[41] Esko Ukkonen. Approximate string-matching over suffix trees. In Alberto Apostolico, Maxime Crochemore, Zvi Galil, and Udi Manber, editors, *Combinatorial Pattern Matching*, volume 684 of *Lecture Notes in Computer Science*, chapter 17, page 228–242. Springer-Verlag, Berlin/Heidelberg, 1993.

[42] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[43] Julian R. Ullmann. A binary n-gram technique for automatic correction of substitution, deletion, insertion and reversal errors in words. *Comput. J.*, 20(2):141–147, 1977.

[44] Robert A. Wagner and Roy Lowrance. An extension of the string-to-string correction problem. *J. ACM*, 22(2):177–183, April 1975.

[45] Peter Weiner. Linear pattern matching algorithms. In *SWAT '73: Proceedings of the 14th Annual Symposium on Switching and Automata Theory (swat 1973)*, page 1–11, Washington, DC, USA, 1973. IEEE Computer Society.

[46] S. Wu and U. Manber. Agrep – a fast approximate pattern-matching tool. In *Proceedings USENIX Winter 1992 Technical Conference*, page 153–162, San Francisco, CA, 1992.