A Depth-First Branch-and-Bound Algorithm for Geocoding Historic Itinerary Tables

Daniel Blank
Media Informatics Group
University of Bamberg
Bamberg, Germany
daniel.blank@uni-bamberg.de

Andreas Henrich
Media Informatics Group
University of Bamberg
Bamberg, Germany
andreas.henrich@uni-bamberg.de

ABSTRACT

The work in this paper is motivated from two different perspectives: First, gazetteers as an important data source for Geographic Information Retrieval (GIR) applications often lack historic place name information. More focused historic gazetteers are a far cry from being complete and often specialize only on certain geographic regions or time periods. Second, research on historic route descriptions—so called itineraries—is an important task in many research disciplines such as geography, linguistics, history, religion, or even medicine. This research on historic itineraries is characterized by manual, time-consuming work with only minimalistic IT support through gazetteers and map services.

We address both perspectives and present a depth-first branch-and-bound (DFBnB) algorithm for deducing historic place names and thus the stops of ancient travel routes from itinerary tables. Multiple phonetic and character-based string distances are evaluated when resolving parts of an itinerary first published in 1563.

CCS Concepts

•**Information systems** → *Geographic information systems;*

Keywords

itinerary resolution, historic place name disambiguation, geocoding, depth-first branch-and-bound, gazetteer enrichment

1. INTRODUCTION

Many geographic information retrieval (GIR) applications relying on map services and the disambiguation of toponyms found in texts make use of gazetteers. However, these are often incomplete, especially with respect to historic places and their spelling variants.

In this work, we try to enhance gazetteers with historic spelling variants of toponyms. This is of particular interest in the humanities where historical researchers "*... need to know about alternative names for places, about how names have evolved over time, and the specific historical contexts in which names were used*" [26, *p.* 127].

Besides the identification of alternative names and spelling variants of places, our second goal is the geocoding and visualization of the stops of ancient travel routes themselves. This is an important aspect in many disciplines in the humanities such as geography, linguistics, history, religion, or even medicine. It has to do with the fact that former scholars who constructed the itineraries were traveling—often at the same time—as geographers, doctors, historians, merchants, and pilgrims [13]. Grounding the place names found in an itinerary is difficult even for humans (see e.g. [14]) and IT support is often limited to gazetteers and map service tools.

Itinerary resolution from tables can be considered a four-step process [4]: i) *optical character recognition*, ii) *itinerary parsing*, iii) *toponym resolution*, and iiii) *route finding*.

We analyze itineraries which consist of several itinerary tables (formally defined in Section 3.1). The tables can also exist standalone. They contain toponym-distance-pairs. Each pair represents a route stop at a certain location described by the corresponding toponym. In addition, distance information indicates how far the current location/stop is away from the previous stop. Distances can be explicitly given in medieval miles for example or implicitly when route stops indicate dates, times, and locations of the travelers. From this information, distances can be roughly deduced.

Although we target historic itinerary tables, our general approach is by no means restricted to this scenario. The approach can be applied for all kinds of itineraries and travel routes. Recently, [1] proposed a technique for identifying itineraries in the web which complements our approach (For a more detailed description of the approach taken in [1] see Section 2.2.). Since there is a huge amount of itineraries out there in the web [1], the application of our approach can help in the toponym resolution of recent texts as well.

In this paper, we consider the third step of itinerary resolution where it is the task to ground places named in the itineraries. As preliminary steps, we assume an optimal character recognition and itinerary parsing, including the extraction of distance information whenever possible. In a fourth step, which we will target in future work, it is an important research question to identify the roads which underly the current route description and which have been traveled. These roads do not necessarily correspond to actual roads and can thus—in addition to current road networks—be derived from contour information and digital elevation models together with external data such as textual descriptions.

We choose a DFBnB approach [2] which makes use of two kinds of distance information: (1) a string distance between toponyms and (2) a geographic distance modeling the travel distance between locations on the earth.

The remainder of this paper is organized as follows. Section 2 addresses related work in the field. Afterwards, Section 3 describes our approach. Test data and an evaluation is given in Section 4 before we conclude our work in Section 5.

2. RELATED WORK

From an implementation side, work on heuristic search in general (see e.g. [10]) and DFBnB in particular [2] is the algorithmic foundation of our work. Runtime-related aspects of DFBnB are discussed in Section 4.3.2. In addition, we rely on previous work on phonetic and character-based string matching. The algorithms we use are briefly introduced in Section 3.2. This related work section addresses the geocoding of (historic) itineraries (Section 2.1), itinerary retrieval (Section 2.2), IR applications relying on historic spelling variants of toponyms (Section 2.3), and related research in the (spatial) humanities (Section 2.4).

2.1 Geocoding Itineraries

There is a huge amount of research on geocoding in GIR applications (for an overview see [18]). Some of the approaches focus on itineraries and their particular characteristics. Our work is different because of two reasons: Itineraries and for example hiking descriptions are usually characterized as short texts containing (partial) sentences [9, 20, 21]. This allows the use of language analysis and NLP techniques for the geocoding. However, we focus on itinerary tables consisting only of toponym sequences. In addition, distance values may denote distances to previous/following stops. Second, we focus on historic itinerary tables with many unknown toponyms because they represent spellings from the Late Middle Ages and the Early Modern Age.

Of course, there are similarities to other geocoding approaches. The idea in [21] is to analyze micro-toponyms usually not found in gazetteers and estimate their footprint based on an unsupervised clustering-based geocoding algorithm. The disambiguation uses spatial relations in the texts and density-based clustering techniques to find arbitrarily shaped clusters and detect outliers. Unreferenced toponyms are presumed in bounding polygons such as convex hulls and circumscribed circles. [21] seeks for the definition of new, current toponyms. On the opposite, we look for past, nowadays out-dated toponyms.

Graph-based approaches are common for the geocoding of itineraries. [20] describes an approach for resolving French hiking descriptions. The locations are the vertices and the edge weights are modeled as a weighted sum of eight factors. From these eight factors only geographical distance and a modified view of orientation apply to our scenario. Rather than deriving orientation from words such as north, north-east, and the like we use a directionality definition (see below in Section 3.3). Other factors identified in [20] such as the sequence of the displacement do not apply since in our case the order of the stops is predetermined. Factors such as temporality (indicated by keywords such as before, after, then, etc.) or elevation (derived from expressions such as to climb, to come down, etc.) are also not applicable in our case since they depend on textual input data and NLP techniques. What distinguishes our approach further from [20] is that we plan to incorporate elevation information only in a follow-up process where the actual roads are to be determined. To us, matching the stops and finding the roads are two sub-processes which should also be handled separately if desired by the user.

There is also some work on the disambiguation of road names in text route descriptions using an exact-all-hop shortest path algorithm [31]. This work targets obsolete and misspelled toponyms and road names. Similar to us, no external knowledge is employed besides a flat gazetteer and a graph-based algorithm is used to find the best solutions on noisy input data. However, there are major differences compared to our approach. Spelling variants are not considered in [31]. Subvertexes are ambigious road names which are in each vertex characterized by the same spelling. In our case, different toponyms which may represent a spelling variant are represented as vertexes. In [31], graph weights are geospatial distances whereas we model string distances between toponyms as graph weights. The approach in [31] allows subpaths as acceptable solutions, that is, certain stops on a route may be skipped although maximizing the number of stops is desired. In contrast, with our modeling approach we only allow routes of full length where every stop must be contained in a route. No vertex filtering is applied in [31] when building the graph. We filter vertexes directly during graph creation by applying distance thresholds and interior angles calculated from azimuths. As another difference, different geometries are considered. Whereas [31] relies on lines/roads our algorithm focuses on point matching.

To our knowledge, no computational approach besides [4] has been proposed targeting the geocoding of historic itinerary tables characterized as lists of toponym-distance-pairs lacking additional information useful for the geocoding.

2.2 Itinerary Retrieval

The main purpose of itinerary retrieval [1] is the identification of itinerary tables found on the web with the help of a focused crawler. Machine learning is applied to classify a candidate table as an itinerary. Spatial efficiency measures—local and general efficiency—derived from the 2-opt heuristic for solving the traveling salesman problem [7] are applied. "*Local efficiency is the fraction of consecutive stop pairs whose reversal would lead to a longer total route distance.* [...] *General efficiency is the fraction of all unique, non-consecutive edge pairs that would result in a longer total route if their endpoints were swapped.* [1]" The authors in [1] claim that both measures have a significant impact on classification accuracy.

Different to [1], the identification of itinerary tables is not our goal. Instead, we want to geocode (historic) place names from itinerary tables. Thus, the work in [1], although it does not target ancient itineraries, could be an important preprocessing step for the automatic identification of itinerary tables and complement our work. From [1] we capture the idea of local and general efficiency to make our approach more effective by applying filter criteria. For example, we apply a directionality filter based on the interior angles calculated from azimuths for eliminating inefficient paths which is also the goal of using the two efficiency measures from [1, 7].

2.3 Historic Spelling in GIR Applications

There is a huge amount of work addressing spelling variation in IR (for references see [12]). Considering historic

spelling variants is an important task, too. Ernst-Gerlach and Fuhr [12] analyze the retrieval with historic spelling in German independent from GIR. Similar to our scenario is the 'write as you speak' assumption which characterizes the itinerary tables we consider and which is typical for the German language before 1900 [12].

String similarity measures are frequently used for the tasks of name and address matching. Their use in GIR applications based on the German language is less frequent but there is an interesting approach in this regard related to our work. [25] focuses on string similarity matching of historic spelling variants of toponyms in an isolated manner with no direct GIR application in mind. It compares 21 string similarity measures for toponym matching in a setting with datasets from 11 different countries. The experimental setting is straightforward. 2,000 gazetteer entries which are described by more than one name are first selected per language. The union of alternative names of those sets of toponyms makes up the query set. Each entry in the query set is then tested and if the highest string similarity matches the correct toponym it is considered a match. The number of matches is used as the main effectiveness indicator. For toponyms in the German language, measures using skip-grams, trigrams, and bigrams perform best. We also use these measures and analyze if the findings in [25] apply to our setting.

2.4 Research in the Spatial Humanities

Gazetteer integration is concerned with the matching of historic toponyms from different gazetteers. In [3] the matching is performed by first applying a geospatial range filter and then calculating string distances between the gazetteer entries. What is helpful here is the fact that the source toponyms for which matching toponyms in an unrelated dataset are to be found are already annotated with geospatial coordinates. This is however not the case in our scenario.

Many studies in the humanities deal with ancient itineraries [13, 14, 16]. For the analysis, map services and gazetteers are used [14] and the demand for additional IT support when analyzing the itinerary tables has been noticed before [4].

Often, itinerary tables are not directly available. Instead, researchers analyze textual itineraries and extract itinerary tables with unknown entries themselves (see e.g. [14]). Then they geocode the entries by hand. We target this second step and we want to provide algorithms which can help researchers here.

The algorithm we present in this paper makes use of filter conditions in order to improve effectiveness. We apply geospatial filters. A filter based on directionality—as mentioned before—and a filter based on geospatial distance in the form of metric shells on the globe. This is for example also done in [11] where distance information (e.g. a mile) is transfered to shells on a map defined by a starting point and upper and lower bounds on the distance [11, Section 7.2].

3. GEOCODING ITINERARY TABLES

With this paper, we present an algorithm for the geocoding of historic itinerary tables. Besides its use in the humanities, this can also be helpful for GIR applications when the techniques are used for gazetteer enrichment. We tackle the problem with a graph-based approach and compare the historic spelling variants in the itinerary tables with the place names contained in a gazetteer which are then represented as nodes in a graph. While creating the graph we apply filters (e.g. using distance bounds between locations we obtain or based on the interior angle of stops calculated from azimuths). Vertexes are linked by edges and edge weights reflect string distances between the toponyms. By doing so, we can apply DFBnB to determine a ranking of potential travel routes (represented by their stops).

We will in detail describe our approach for the geocoding of (historic) itinerary tables. First, the notation is introduced in Section 3.1. Then, the string distance measures and their application are described in Section 3.2. Finally, the DFBnB algorithm is outlined in Section 3.3.

3.1 Notation

Before digging into the details of resolving historic names with the DFBnB algorithm, we take a more formal look and use the following definitions:

- An *itinerary table* I is denoted as a list of pairs, each consisting of a historic toponym $h_i \in H$ and corresponding distance information m_i in medieval miles: $I = \langle (h_i, m_i) | i = 0, ..., N-1 \rangle$.
- To resolve historic names, we make use of a *gazetteer* G made up of gazetteer entries $g_j \in G$. Each gazetteer entry is represented as a tuple: $g_j = (t_j, lat_j, lon_j, alt_j, pop_j)$:
  - t_j denotes the toponym string.
  - lat_j/lon_j represent latitude/longitude information.
  - alt_j stores alternative names and spelling variants concatenated by commas.
  - pop_j states the population.
- *External parameters* are denoted in Greek letters:
  - δ^{str} representing a string distance threshold used to decide if a toponym from the gazetteer qualifies for further analysis during an expansion step of the DFBnB algorithm.
  - δ^{geo}_{lo} and δ^{geo}_{hi} denoting lower and upper bounds of a medieval mile respectively used for deriving geospatial distances in km from the medieval mile information m_i associated with an entry (h_i, m_i) of an itinerary table.
  - α as a threshold for an interior angle used to check if a route part offers some way of "directionality" typical for ancient route descriptions.

3.2 String Distances

The following Section 3.2.1 briefly describes the string distances used in our study. Section 3.2.2 outlines in more detail how we deal with multi-word toponyms and alternative spelling variants also available in the gazetteer.

3.2.1 String distances used

We will summarize different techniques for the computation of string distances $dist(s_1, s_2)$ between two input strings s_1 and s_2, both, character-based and phonetic string distance measures. The use of phonetic distances is motivated by the fact that in Germany up to the 20th century it had been quite common to write as you speak [12].

Levenshtein distance: As a first candidate we use the well-known Levenshtein distance (lev) [17] which equally

weights insert, delete, and replacement operations. It measures the cost for transforming one string into the other. The relative Levenshtein distance is used dividing distance values by the length of the longer string. This leads to distance values between zero and one.

Jaro distance: We use the Jaro distance $1-sim_{jar}(s_1, s_2)$ and

$$sim_{jar}(s_1, s_2) = \begin{cases} 0 & \text{if } z = 0 \\ \frac{1}{3}\left(\frac{z}{|s_1|} + \frac{z}{|s_2|} + \frac{z-p}{z}\right) & \text{otherwise} \end{cases}$$

with p being half the number of transpositions needed for transforming one string into the other and z denoting the number of matching characters of the two strings. $|s|$ denotes the length of a string s. Characters from s_1 and s_2 match only if they equal each other and are not farer away than $\left\lfloor \frac{\max(|s_1|,|s_2|)}{2} \right\rfloor - 1$.

Since in the German language prefixes were added for making toponyms less ambiguous (e.g. *Bischofsheim* became *Tauberbischofsheim*), we do not apply the Jaro-Winkler distance where matching prefixes obtain a bonus [28].

For both, the Levenshtein and the Jaro distance we apply the jellyfish python library (*https://pypi.python.org/pypi/jellyfish*, version 0.5.1).

***q*-gram distance**: The q-gram similarity counts the number of character q-grams (also known as character n-grams) which the two strings s_1 and s_2 have in *common*. This value *common* is then divided by a normalization factor. Since we need a distance value, we subtract the similarity value from one to obtain a distance measure: $dist_{qgr}(s_1, s_2) = 1 - \frac{2*common}{|s_1|+|s_2|}$. We test bigram (*2gr*) and trigram (*3gr*) distances.

Skip-gram distance: Skip-grams are conceptually similar to q-grams. The main difference is that during the extraction of the q-grams some characters are skipped. Afterwards, the skip-grams are handled just as q-grams. Let us consider the example of *Aub* (we just compare lowercase toponyms). Usually, without padding, *au* and *ub* would result as bigrams. In case of one-skip bigrams, also *ab* is taken into account since the character *u* is skipped. We test one-skip bigrams (*1s2*), two-skip bigrams (*2s2*), and two-skip trigrams (*2s3*).

DAS: The DAS distance (*das*) for string matching is proposed in [15]. It is a character-based measure which can roughly be summarized as follows. If characters in the strings match, this is of course desired. If the matching occurs with a positional shift of plus/minus one, this is still acceptable. Otherwise, the distance value gets a huge increase.

Cologne phonetics: The Cologne phonetics [24] (*col*) was developed in 1969 as a phonetic string matching algorithm for the German language following a rule-based approach. It often replaces the well known Soundex variants usually applied for the English language.

For the Cologne phonetic as well as the other algorithms providing phonetic codes presented in the following we use the abydos python library (*https://pypi.python.org/pypi/abydos*, version 0.2.0).

To derive a distance measure, phonetic codes must be matched. For this purpose in this paper we apply the Levenshtein distance as described above.

Soundex: Soundex (*sdx*) [22] is a well known phonetic string matching algorithm for the English language. It has been widely applied and mutual variants of the algorithm exist. We use the basic version here for comparison reasons only which is also known as American Soundex.

Phonet: Phonet [19] is a string matching technique targeting German names and addresses. It was developed in 1999. The rule set of Phonet 1 (*ph1*) massively increased afterwards culminating in a second version of the algorithm called Phonet 2 (*ph2*).

NYIIS: The New York State Identification and Intelligence System Phonetic Code (NYIIS) [27] is a rule-based approach providing codes for names in a similar fashion as soundex. We abbreviate it in our evaluation as *nys*.

Listing 1: Dealing with partial matches

```
1 def str_dist_p(s_1,s_2,val,dist):
2  if '␣' in s_1:
3   s_1_split = s_1.split('␣')
4   for part in s_1_split:
5    new_string_dist = dist(part,s_2)
6    if new_string_dist < val:
7     val = new_string_dist
8    break
9 return val
```

3.2.2 Computing string distances

String distances are computed by transforming the toponyms from the gazetteer and the itinerary table to lowercase and replacing characters such as punctuation marks, brackets, slashes, and backslashes by a whitespace. Multiple occurrences of whitespaces are trimmed to a single one. Afterwards, the actual distance between the two strings representing toponyms is computed and the result is stored. This applies well for strings such as *Franckfurth* and *Frankfurt* consisting of only a single token. However, in cases when multiple words make-up a toponym this is not sufficient.

To better estimate string distances between two toponyms such as *Franckfurth* and *Frankfurt am Main* for example, toponyms must be split. This is done within `string_dist_p` as shown in Listing 1. The parameter `val` stores the initial result computed as explained before without splitting the strings and `dist` represents the distance function.

The second line of this method tests if the first toponym contains a whitespace character. If this is the case, the string is split. Afterwards, the parts of the string are processed sequentially and for each part the distance to the second string is computed and stored as `new_string_dist`. If such a value is smaller than an earlier computed distance value `val`, then `val` is updated. Afterwards, that is after the first part of `s_1` has been processed the loop breaks and `val` is returned. This in fact means that the for-loop within this method could be replaced by an if-statement. However, the inclusion of this break-statement is questionable as the following example shows.

Consider the two toponyms s_1 = *Haßfurt am Main* and s_2 = *Westheim bei Haßfurt*. Here, it is desired that both toponyms match. The word *bei* means *close to* here. Transferred to our research problem, we prefer a matching of *Westheim bei Haßfurt* since it points the user toward the correct geographic region[1]. The reason for placing the break-statement can be observed when computing string distances

[1]Discounting these matches is not done here. It is part of future work when testing distances for the matching of substrings [6].

Listing 2: The DFBnB algorithm

```
1  dfbnb(idx, path, cost_to_here):
2   if idx == N:# end is reached
3    if cost_to_here < result[k-1].cost:
4     update_result(path, cost_to_here)
5   else:# idx < N: still stop(s) to process
6    expansion = compute_expansion(idx, path)
7    for node in expansion:
8     new_cost = cost_to_here + node.cost
9     est = new_cost + lb_cost_to_end(idx)
10    if est < result[k-1].cost or idx == N-1:
11     new_path=path[:]
12     new_path[idx]=node.g
13     dfbnb(idx+1,new_path, new_cost)
```

between s_1 = *Haßfurt am Main* and s_2 = *Frankfurt am Main* for example. Both toponyms show a similar pattern as both lie on the river Main. Without the break-statement and without further adjustments we would obtain a perfect match because of *am* and *Main* here which is undesired though.

The input toponyms s_1 and s_2 of string_dist_p are also switched to test both directions. In addition, alternative names of a toponym are also analyzed if they exist. They are iterated and each name is again compared to the toponym from the itinerary table as described before.

3.3 DFBnB for Itinerary Resolution

A DFBnB algorithm for identifying the stops of an itinerary table is sketched in Listing 2. Its search graph is shown in Figure 1. The algorithm outputs the k-best routes. From a conceptual point of view, it can be designed in two different ways. A naïve algorithm would start with the first/last stop of the itinerary table and then iteratively move toward the end/start. This corresponds to standard DFBnB. In a more informed fashion, the algorithm could pick the stops in order of decreasing chance to find a good solution for the particular stop. This means processing the itinerary stops h_i in ascending order of a lower bound string distance $\check{d}_i$ from the particular stop h_i to any of the gazetteer entries $t_j \in G$, i.e. $\check{d}_i = \arg\min_{t_j \in G} dist(h_i, t_j)$. In case of ties, less ambiguous toponyms are preferred.

In the following, the informed version of the algorithm is described as the naïve version can be derived from this description in a straightforward way by assuming all $\check{d}_i$ values being the same and picking stops in the order given by the itinerary table or reversely.

The algorithm in Listing 2 takes three arguments: first, idx counting the level of the search tree on which the algorithm currently operates (starting with 0 and ending with N when the virtual end node is reached), second, the path the algorithm analyzed so far, and third, costs_to_here, i.e. the cumulated string distances of all itinerary table stops visited so far on the current path. The initial call[2] to the algorithm is: dfbnb(idx=0,path=[None]∗N,cost_to_here=0).

[2]We initialize the path with an empty list of length N. This is done for ease of implementation and explanation as it makes the expansion step more comfortable (see below). It costs 8 bytes per empty path entry whereas if we decide to append to the list during the processing of the algorithm, this would cost runtime performance and also involve storing an additional position variable. Of course, this can be done in less than 8 bytes. However, the presented solution also allows for a shorter and clearer explanation (see Listing 3 for an explanation of the expansion step).

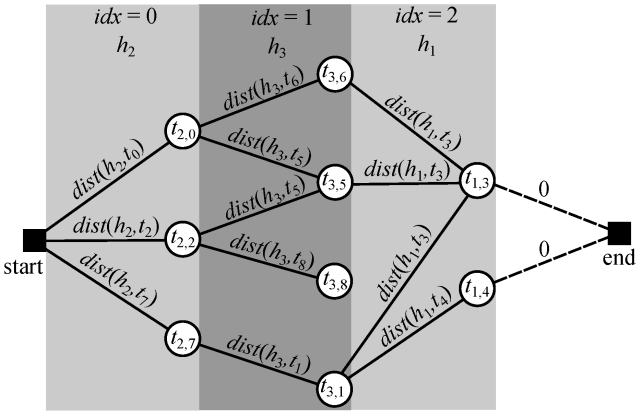

Figure 1: Example DFBnB search graph with $N = 3$.

The beginning of Listing 2 first checks if N nodes have been processed so far, that is, the virtual end node is reached. In this case, it is analyzed if the current solution is better than the worst solution in the result list so far, i.e. the solution at position $k-1$ of the result list. If this is the case, the results are updated within update_result. In the else-part starting with line 5 the search graph is expanded. This is done for each branch as long as the estimated cost est is smaller than the k-best solution so far or the end node can be reached with no additional cost (idx == N−1). The expansion list (see line 6) is computed by Listing 3.

First, the current depth of the search path idx needs to be mapped to the corresponding ID i of the table entry h_i. Afterwards, two geospatial distance values d_to_here and d_from_here are aggregated: d_to_here is the distance from the closest—in terms of positions i in the itinerary table—itinerary entry to the current entry and d_from_here represents the distance from the current itinerary entry to the next successor, again in terms of positions i, i.e. the next itinerary entry that has already been processed. Values of -1 are returned if no predecessor or successor has yet been processed. Note that the lists pre_n[] and suc_n[] store the up to $n-1$ closest predecessors and successors of the current itinerary table entry in terms of position differences given by the order in the itinerary table.

Identifying promising gazetteer entries starts in line 10 when the algorithm iterates over the entries of the gazetteer. Line 12 checks if filtering based on the geospatial distance is desired (use_geodist=True). The second condition idx > 0 assures that at least one stop has been processed before because at least two stops are necessary for computing geospatial distances. From line 13 onward the algorithm computes, if possible, geospatial distances from a previous stop to the current as well as from the current to a following stop. If the obtained distance values do not lie within the distance bounds $m_i \cdot \delta_{lo}^{geo}$ and $m_i \cdot \delta_{hi}^{geo}$ (δ^{geo} as external parameters give lower/upper bound estimates of a medieval mile), the algorithm skips the current gazetteer entry and continues with the next.

In line 24 starts the filtering by interior angle at a stop if indicated by use_bearing=True. This kind of filtering can only be done as soon as three stops have been processed (idx > 1).

Listing 3: Search graph expansion

```
def compute_expansion(idx,path):
 # expansion list to be filled
 expansion=[]
 # get id of stop from idx value
 i=map_idx_to_pos(idx)
 # dist from previous stop to here
 d_to_here = aggregate_dist(pre_n[0], i)
 # dist from here to following stop
 d_from_here = aggregate_dist(i, suc_n[0])
 for g_j in G:# iterate through gazetteer
  # geo distance filtering
  if use_geodist and idx > 0:
   if d_to_here != -1:
    pre_g = path[pre_n[0]]
    keep = geodist(pre_g,g,d_to_here,δ^geo_lo,δ^geo_hi)
    if not keep:
     continue
   if d_from_here != -1:
    suc_g = path[suc_n[0]]
    keep=geodist(g,suc_g,d_from_here,δ^geo_lo,δ^geo_hi)
    if not keep:
     continue
  # geo bearing filtering
  if use_bearing and idx > 1:
   # (1,*) predecessors, (1,*) successors
   if pre_n[0] > -1 and suc_n[0] > -1:
    g1 = path[pre_n[0]]
    g2 = g_j
    g3 = path[suc_n[0]]
   # (2,*) predecessors, no successor
   elif pre_n[0] > -1 and suc_n[0] == -1:
    g1 = path[pre_n[1]]
    g2 = path[pre_n[0]]
    g3 = g_j
   # no predecessor, (2,*) successors
   elif pre_n[0] == -1 and suc_n[0] > -1:
    g1 = g_j;
    g2 = path[suc_n[0]]
    g3 = path[suc_n[1]]
   keep = geobearing(g1,g2,g3,α)
   if not keep:
    continue
 cost = dist(t_j,h_i)
 if cost <= ď_i + δ^str:
  expansion.append((g_j,cost))
 return sort_by_dist_and_pop(expansion)
```

Otherwise no interior angle can be computed. We test the three possible scenarios: one or more predecessor(s) and successor(s), two or more predecessors and no successor, and no predecessor and two or more successors. The method actually testing if a gazetteer entry should be kept geobearing(.) uses the parameter α. Angles below this threshold are not permitted and corresponding gazetteer entries are pruned.

Finally, if a gazetteer entry g_j was not pruned by geofiltering, neither based on the geospatial distance nor based on the interior angle, cost capturing the string distance between the name t_j of the gazetteer entry g_j and the itinerary entry h_i is included into the expansion as long as the cost is not bigger than the lower bound distance $\check{d}_i$ plus some externally specified threshold δ^{str}. In the last line, the expansion list is sorted by ascending string distance and in case of ties, inspired from geocoding research, by descending population.

Figure 2 visualizes the filtering process. Starting from toponym $t_{2,1}$, moving to $t_{3,1}$ and from there onward we assume

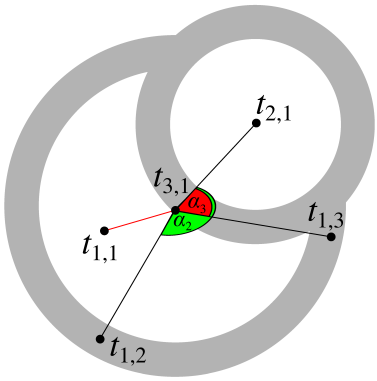

Figure 2: Pruning with geospatial filters.

three potential toponyms $t_{1,1}$, $t_{1,2}$ and $t_{1,3}$. By applying the geospatial distance filter $t_{1,1}$ is discarded since it is not contained in the gray sphere shell centered around $t_{3,1}$. However, this is the case for toponyms $t_{1,2}$ and $t_{1,3}$. Applying the filtering based on the interior angle calculated from the azimuths with an exemplary threshold value of $\alpha = 90°$ yields that also $t_{1,3}$ has to be discarded since $\alpha_3 < 90°$. Thus, $t_{1,2}$ is the only toponym which is integrated in the graph in this small toy example.

4. EVALUATION

There are several options on how to evaluate our retrieval task. They are discussed in Section 4.1. The input data to our algorithm is outlined in Section 4.2 before we present an evaluation in Section 4.3.

4.1 Finding Adequate Evaluation Measures

Let us first look at a single itinerary table I. The task of itinerary resolution is to predict the correct travel route—here in particular the stops of the itinerary table in the correct order. If we assume a table with three stops starting in location a and ending in location c by traveling through b for example, then the correct sequence $best$ = a-b-c has to be found. From a system perspective, at first glance, we have a scenario with a single relevant document which has to be ranked as good as possible. Thus, reciprocal rank (RR) is an obvious retrieval measure: $RR(I,k) = 1/rank(I,best,k)$. Here, $rank(I,best,k)$ returns the rank if $best$ is found on the first k ranks when resolving I. Otherwise, the RR is zero.

From a user perspective it is however difficult to apply RR. Given a solution from the algorithm, comprehensive and time-consuming research studies may follow—undertaken by users in order to check the given solution. Usually, users are unable to determine how relevant the given result document is without further investigations. Thus, it is essential that the solution at rank 1 or at the first k ranks is sufficiently good. If not all but only some of the stops are identified, it is important in our application context that the partial solution at least identifies stops close to the correct locations. Thus, another obvious evaluation measure is the mean sum of squared error ($msse$) of the geospatial distance between the stops given in the $result$ at rank r and the ground truth $best$:

$$msse(I,k) = \frac{1}{k}\sum_{r=1}^{k}\sum_{i=0}^{N-1} geodist(result_r[i], best[i])^2$$

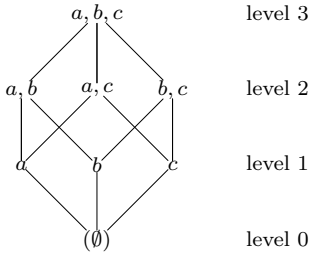

Figure 3: Example lattice.

Here, *geodist* is the Haversine distance, r the current rank, and the index i the position in the itinerary table I, as before. We do not discount the ranks because without doing so the *msse* remains interpretable. If we analyze results at rank one only, we use the sum over all tables of the sum of squared error of the stops of a table ($\sum sse$). When only a single table is analyzed at rank one, the sum of squared error (sse) is given.

We have so far assumed binary relevance. Assuming graded relevance does not make sense as will be shown by the following example. We could model the number of correctly identified locations as the relevance grade gr, so $gr = 3$ in case of *a-b-c*, $gr = 2$ in case of *x-b-c* for example, and so on (x represents a non-matching location). However, it would be a problem that $gr = 2$ is assigned to both *x-b-c* as well as *a-x-c* for example. With many evaluation measures for graded relevance both solutions $\langle$*x-b-c*, *x-b-c*$\rangle$ and $\langle$*a-b-x*, *x-b-c*$\rangle$ result in the same evaluation score (lists $\langle . \rangle$ indicate ranked results), although the second solution seems preferable since at rank two all three stops have at least once been identified. Thus, those schemes are inappropriate here. If however only the solution at rank one is analyzed, counting the number of correctly identified stops for determining the accuracy at rank one (acc@1) of course makes sense.

To define an adequate evaluation measure, we take a look at the evaluation of subtopic retrieval [29] and question answering [8]. Here, evaluation measures are based on the retrieval of subtopics or so called information nuggets. Let us first assume that each stop represents such a nugget, e.g. a, b, and c in case of $S = \{a, b, c\}$ (the set of stops). Measures from subtopic IR assume that a traditional measure such as RR is calculated first per subtopic and later the scores are aggregated. If we take a look at subtopic reciprocal rank of two rankings $\langle$*a-x-x*, *x-b-x*, *x-x-c*$\rangle$ and $\langle$*x-x-x*, *a-b-c*, *x-x-x*$\rangle$, the corresponding scores become 1+1/2+1/3 $\approx$ 1.8 versus 1/2+1/2+1/2 = 1.5 clearly preferring the first ranking. However, this is undesired in our scenario, too.

Thus, we decide to model the nuggets differently. Besides the individual occurrence of a stop such as in *a-x-x* or *x-b-x*, we model also combinations such as *a-b-x* indicating that both stops a and b have been correctly retrieved as an information nugget. Thus, the set of information nuggets is given by $Nuggets = \mathcal{P}(S) \backslash \emptyset$, the power set of the stops ignoring the empty set. In total, we thus have $2^N - 1$ nuggets to find for an itinerary table of size N. Figure 3 shows a lattice with arcs indicating the desired behaviour, that if a nugget is found, also the subsets are found which is a desired behaviour in our case. As an example: if *a-b-x* is retrieved at a certain rank, also *a-x-x* and *x-b-x* are found.

To sum it up, we use the following evaluation measures to decide about the quality of different parameterizations of our algorithm: i) accuracy, that is, the number or fraction of correctly identified stops, ii) *msse* of the Haversine distance of the retrieved locations, and iii) the measure based on the power set (ps)[3].

4.2 Input Data

Let us first describe the itinerary we use (Section 4.2.1) before we focus on the gazetteer (Section 4.2.2).

4.2.1 Gail's travel itinerary

We evaluate our approach based on travel routes taken from an itinerary first published in 1563 by Jörg Gail who lived in Augsburg/Germany. A facsimile of this itinerary is given in [16]. Gail's itinerary is made up of 272 pages. It is said to be the first autonomously printed travel itinerary in the German language [16, *p.*1]. Routes are placed in continental Europe. The German travel metropolises of the Late Middle Ages Augsburg and Nuremberg form the centers of the spider-like route net. Augsburg is part of 21 routes whereas Nuremberg is named in 13 routes. These two hotspots are also part in several of the routes we use.

We transcribed 15 more or less randomly selected German routes with 218 stops in total into textual representations. By doing so, we assume an optical character recognition step upfront transforming the images into text and adequately parsing it. Our routes are focused around Southern Germany because this is the hot spot of Gail's work.

4.2.2 Geonames gazetteer

In this study, only itinerary tables with stops in Germany are analyzed although the approach is by no means restricted to this particular geographic region. We apply the German part of the Geonames gazetteer[4] with β_1 = 78,882 populated places (denoted as G_1). In addition, we use a reduced gazetteer G_2 with only β_2 = 11,969 places, that is G_1 restricted to the locations with population information stated in the gazetteer. Usually, population counts are only given for cities. Villages are not labeled with this information. The use of this second, smaller gazetteer G_2 is motivated because for some cases it is very likely that the route stops in the 16th century have evolved to bigger towns and cities in the present. Using the smaller gazetteer G_2 offers the benefit that the runtime of the algorithm is reduced.

4.3 Evaluation Results

Section 4.3.1 discusses how we preselect distances. Runtime aspects are considered in Section 4.3.2 and the core of the evaluation is given in Section 4.3.3.

4.3.1 Preselecting string distances

The entire evaluation is not done with all distance measures from Section 3.2.1. Instead, only some promising distances are selected. To do so, we compute the distances from an itinerary table entry to all gazetteer entries from G_1 and sort the resulting list by increasing string distance. Then, we determine for each itinerary table entry the rank of the matching gazetteer entry. Figure 4 shows the distributions of those ranks. Table 1 provides additional statistics such as

[3]Of course, computing the measure based on the power set is in $\mathcal{O}(2^N)$. However, for small itinerary table sizes N this is feasible.
[4]http://www.geonames.org/

dist	*lev*	*jar*	*2gr*	*3gr*	*1s2*	*2s2*	*2s3*	*das*	*col*	*sdx*	*ph1*	*ph2*	*nys*
time in min.	32	11	9	8	22	27	40	11	44	38	3298	3280	38
∅rank	1029.4	1995.6	1754.1	4343.6	1108.7	935.1	1788.4	5908.6	1110.8	5708.2	1161.7	1318.6	6164.2
ρ	29.0	6.3	39.3	155.6	46.5	45.6	268.9	32.2	12.0	10.6	27.0	20.3	12.6

Table 1: String distance statistics using G_1.

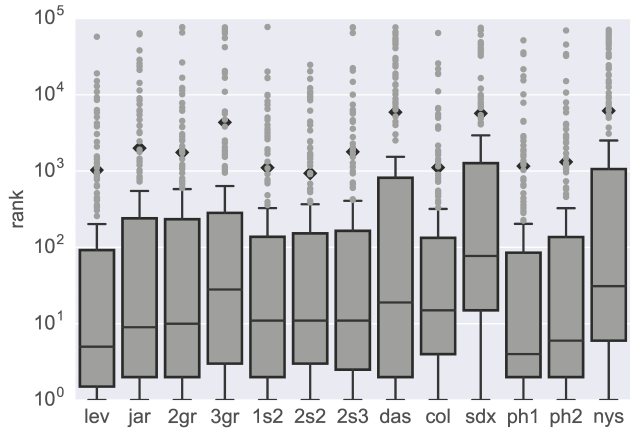

Figure 4: Boxplot of the ranks (means as diamonds).

execution times (Intel i7-860@2.8GHz, Win7, Python 3.5.1), average ranks, and intrinsic dimensions $\rho = \mu^2/2\sigma$. The intrinsic dimension [5] is used as an indicator for the selectivity of the distance measure with μ representing the average distance and σ the variance.

Clearly inappropriate are measures such as *das*, *sdx*, and *nys*. The latter two are designed for the English language and also *das* seems inappropriate for ancient German toponyms because focusing with the matching on the exact positions as well as the immediate predecessor and successor falls short. It can also be observed that the q-gram measure *3gr* is outperformed by *2gr* and the skip-gram measures. *1s2* and in particular *2s2* seem applicable with *2s2* having the best average rank. The Levenshtein distance offers the second-best average rank. As another measure, especially with the intrinsic dimension in mind, we select *col*. Also the Phonet measures perform quite well. However, because of the impracticable runtimes we do not consider them further. As a sixth measure besides *2gr*, *1s2*, *2s2*, *lev*, and *col*, we select *jar* because of its good runtime performance and its low intrinsic dimension according to the dataset. The p-values from pairwise Student t-tests back our decision.

4.3.2 Runtime aspects

The worst case runtime complexity of DFBnB is $\mathcal{O}(\beta^N)$ and thus exponential in terms of the search depth N, that is in our case the number of stops of an itinerary table. This also holds for the average case if the number of same-cost children is on average below one [23]. Same-cost children refer to the number of children of a node with the same cost as their parent. For our scenario with the cost defined as a string distance between historic place names $h_i \in H$ and toponyms from a gazetteer $t_j \in G$ this can quite often be the case because usually the historic names are only in rare cases contained in the gazetteer with $dist(h_i, t_j) = 0$. In our case, the string distances $dist(h_i, t_j) = 0$ can be cached to reduce runtime. However, load imposed by the geospatial filtering is still present.

It becomes obvious that the exact algorithm is for many scenarios rather impracticable. Runtimes with less than five minutes for example only work for very short itinerary tables when using fast distance measures such as the Jaro distance and the small G_2 gazetteer with only cities. However, DFBnB suits as an approximation algorithm being able to find good solutions early within its execution [30].

There are different possibilities on how to design an approximate DFBnB algorithm. First, a truncated algorithm can be designed by simply stopping the depth-first search as soon as the first or the first k goal nodes are reached [30]. However, in our scenario even this is sometimes runtime expensive. As a second option, epsilon-transformation can be applied. It is shown in [23] that artificially boosting the number of same-cost children by setting all distance values in the expansion step smaller than a threshold ϵ to zero can lead to a polynomial runtime performance. Instead of implementing this second solution, we decided to use a rather simplistic approach which stops execution after a certain time period. The reason for doing so is that we plan to design a user-centric web application with reasonable response times first. In-depth, exact analysis are left-out to an offline usage of the algorithm which is however not the focus of this work here. We want to answer the question which distance measures and parameterization already offer satisfactory results within given time bounds.

DFBnB can be applied in a one-step as well as a two-step approach. For the first, G_1 or G_2 can be used together with a time threshold θ_{end}, that is the final time threshold which, if it is exceeded, terminates the algorithm. For the latter, we use the small gazetteer G_2 first to compute a list of up to k initial results. If k results and thus a useful upper bound (i.e. the cost value of the k^{th} solution) are found or the time threshold θ_{1st} is exceeded, we move on to the second phase where we try to improve the top-k list by applying the full gazetteer G_1. As final stopping criterion, we again set the other time threshold θ_{end}.

4.3.3 DFBnB evaluation

In the experiments δ^{str} is set to 0.5 and thus sufficiently large since the distances we analyze in more depth provide values between zero and one. The fine-tuning of δ^{str} is part of future work. We do not feed the full gazetteers into the DFBnB algorithm but only the locations placed within a bounding box around the first and the last stop of the itinerary table with some further extend in each direction of 0.5 decimal degrees. This is done since often researchers roughly know the area where to search for the route or even the start and the end is known (see e.g. [14]).

The assumption that a village being a route stop in ancient times has evolved to a town nowadays and thus offers a population count bigger than zero in the gazetteer is however only justified for 168 out of 215 stops in the Gail dataset (Three of the 218 stops are not resolvable and un-

dist	acc@1	$\varnothing acc$@10	$\varnothing ps$@10	$\sum msse$@1	$\sum msse$@10
lev	53.7%	**51.1%**	3.7%	66.0 tkm	**70.6 tkm**
jar	45.9%	43.7%	1.9%	**65.7 tkm**	73.3 tkm
2gr	41.7%	39.8%	1.6%	91.5 tkm	87.4 tkm
1s2	40.4%	37.8%	1.3%	80.9 tkm	89.3 tkm
2s2	40.8%	38.2%	1.3%	96.9 tkm	96.7 tkm
col	**54.1%**	50.7%	**6.3%**	69.7 tkm	80.4 tkm

Table 2: Two-step approach querying for $k = 10$ results (1 tkm = 1000 km).

known.). Using only G_2 and thus a filter on the population might be an option in some special cases. In our dataset 47 stops cannot be identified this way. Thus, we use the two-step approach applying G_2 and G_1 in the following and set $\theta_{1st} = 120$s and $\theta_{end} = 360$s.

Table 2 shows results aggregated over the 15 tables of our dataset for different distances querying for the top-10 solutions without geospatial filters. With respect to accuracy *acc*, the *ps*-measure, and the sum of *msse*, *lev*, *jar*, and *col* perform best. In case of *col*, at rank one 54.1% of the 218 toponyms can be resolved by our algorithm without applying geospatial filters. For *lev* the good performance was expected (see Table 1 and Figure 4). Interestingly, *jar* is the measure with the fastest runtime and *jar* and *col* have a low intrinsic dimension (see Table 1). In the following, we consider only these three distance measures *lev*, *jar*, and *col*.

Now, we test the three different modes: *mode 0* processes the stops in order, *mode 1* reversely, and *mode 2* corresponds to the algorithm outlined above selectively picking the next stop to be processed. Hereby, we apply the two geospatial filters. For the filter on geospatial distance we test the following pairs (δ_{lo}^{geo}, δ_{hi}^{geo}): (0 km, 20 km), (1 km, 15 km), (4 km, 12 km), and (6 km, 10 km). The filtering based on the interior angles calculated from azimuths is either turned off or turned on with $\alpha = 90^\circ$. Thus we have eight parameter combinations per mode. Table 3 compares the different modes and evaluates the accuracy in terms of correctly identified stops as well as the sum of squared error *sse* summed over the 15 itinerary tables of the dataset. Each table entry in the columns *mode 0*, *mode 1*, and *mode 2* is the best value obtained from eight runs resulting from combining the aforementioned four geospatial distance filter parameterizations and the two settings based on the interior angles calculated from azimuths. The column *best* aggregates the best results from different modes. It can be observed that *col* is the winner in terms of *acc* in all cases and in terms of $\sum sse$ for *best*, *mode 1*, and *mode 2*. Only in case of $\sum sse$ and *mode 0*, *lev* performs best.

At this stage it is promising to delve into the individual itinerary tables. Table 4 shows results in a similar manner as before however individually for every itinerary table. We select *col* as the most promising distance measure. N denotes the number of stops of an itinerary table. 180 from 215 resolvable locations are correctly identified. This gives 83.7%. The average squared distance of a wrongly resolved toponym at rank 1 is 543.3km/38 = 14.3 km.

It can be observed from Table 3 and in particular Table 4 that all modes are worth considering with no clear winner. Analyzing *acc* and as a secondary criterion *sse*, in eight cases (including ties) *mode 0* and *mode 2* perform best, in nine cases *mode 1*. Whereas *mode 1* gives the best accuracy,

	best		**mode 0**		**mode 1**		**mode 2**	
dist	acc	$\sum sse$	acc	$\sum sse$	acc	$\sum sse$	acc	$\sum sse$
lev	155	9.7 tkm	132	7.2 tkm	112	22.7 tkm	131	10.9 tkm
jar	155	1.7 tkm	121	16.4 tkm	97	49.1 tkm	132	6.5 tkm
col	180	0.5 tkm	145	21.0 tkm	153	17.4 tkm	137	5.5 tkm

Table 3: Effectiveness for distances *lev*, *jar*, and *col* when querying for $k = 1$ results by mode and best.

		best		**mode 0**		**mode 1**		**mode 2**	
name	N	acc	sse[km]	acc	sse[km]	acc	sse[km]	acc	sse[km]
I01	10	8	33.6	8	33.6	8	123.5	8	112.4
I09	11	11	0.0	11	0.0	11	0.0	11	0.0
I10	8	7	0.1	7	0.1	7	0.1	7	0.1
I10b	8	5	38.4	5	38.4	5	38.4	5	38.4
I17	15	14	1.3	14	1.3	14	1.3	14	1.3
I23	16	14	2.0	14	4.0	14	2.0	14	2.0
I25	22	20	7.5	4	7509.5	20	7.5	13	148.9
I27	8	6	45.8	3	1975.6	3	1975.6	6	45.8
I37	23	18	24.7	11	1515.9	18	24.7	11	45.8
I45	20	18	2.1	18	2.1	10	1225.2	8	1399.5
I46	20	18	2.4	18	3.2	18	2.4	5	3306.7
I49	14	12	42.4	12	42.4	12	42.4	10	88.2
I4950	12	10	14.1	9	48.7	5	5630.3	10	14.1
I51	19	10	263.2	10	1416.5	2	5550.5	6	263.2
I83	12	9	65.7	1	8490.2	6	2817.9	9	65.7
$\sum$	218	180	543.3	145	21081.5	153	17441.8	137	5532.1

Table 4: Results per itinerary table for *col*.

followed by *mode 0*, *mode 2* wins in terms of sum of squared error leading the user closer to the real locations. This is similar for *lev* and *jar*. Thus, we plan to integrate all three modes into a user interface with *col* as the default distance.

For two of the 15 itinerary tables (I37 and I51) the best solution was obtained by disabling the filtering based on the interior angle of the azimuths. In all other 13 cases the filtering with $\alpha = 90^\circ$ seems suitable since the best results are obtained with this configuration. An analysis for I37 shows that in the itinerary table two stops are given in the wrong order and thus actually good filters can prevent a better solution because of this zig-zag navigation. Analyzing I51 reveals that two stops are alternative stops as stated by [16] also leading to a zig-zag path which the pruning based on the interior angle wrongly discards. Future extensions of the algorithm might deal with such cases (route stops switched by mistake and alternative stops given).

It should also be noted that longer runtimes do not influence the outcome of the algorithm much. Moving from θ_{1st} = 120s and θ_{end} = 360s to θ_{1st} = 1200s and θ_{end} = 3600s does not significantly change the results. We tested this for one particular parameter combination and here the total number of resolved stops remains the same. A more detailed evaluation of the time-thresholds will be part of future work.

5. CONCLUSION AND OUTLOOK

This paper presents a DFBnB algorithm for resolving the stops of historic itinerary tables characterized by many unknown toponyms. The algorithm was evaluated on a set of 15 itinerary tables from an itinerary from 1563. We showed that the algorithm can point users to the correct locations. Depending on the characteristics of the itinerary tables (they

are often heterogeneous within larger itineraries according to the interpretation of distance information or their directionality) no global configuration is best for all cases. Instead, researchers have to adequately choose the parameters. This seems possible since researchers often only focus on individual tables or table parts.

In the future, besides analyzing the parameters of the approach in more depth and a merging of results from different modes, we will test other languages and tables from different time periods. In addition, we work on the route finding between stops as well as on the definition of specific string distances for historic toponyms in the German language and a combination of string distances using supervised learning.

6. REFERENCES

[1] M. D. Adelfio and H. Samet. Itinerary retrieval: Travelers, like traveling salesmen, prefer efficient routes. In *Proc. of the 8th Workshop on Geographic Information Retrieval (GIR'14)*, pages 1:1–1:8, Dallas, TX, USA, 2014. ACM.

[2] E. Balas and P. Toth. Branch and bound methods. In E. L. Lawler, J. K. Lenstra, A. H. G. R. Kan, and D. B. Shmoys, editors, *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*. John Wiley & Sons, 1985.

[3] M. L. Berman and J. Ahlfeldt. Historical Gazetteer System Integration: CHGIS, Regnum Francorum, and GeoNames. *www.fas.harvard.edu/ ~chgis/ gazetteer/ AAG_GazIntegration_23feb2012.pdf*. Version: 23.02.12.

[4] D. Blank and A. Henrich. Geocoding place names from historic route descriptions. In *GIR'15*, pages 9:1–9:2, Paris, France, 2015. ACM.

[5] E. Chávez, G. Navarro, R. Baeza-Yates, and J. L. Marroquín. Searching in metric spaces. *ACM Comput. Surv.*, 33(3):273–321, Sept. 2001.

[6] W. W. Cohen, P. Ravikumar, and S. E. Fienberg. A Comparison of String Distance Metrics for Name-Matching Tasks. In *IJCAI-03 Workshop on Information Integration*, pages 73–78, August 2003.

[7] G. A. Croes. A method for solving traveling-salesman problems. *Operations Research*, 6(6):791–812, 1958.

[8] H. Dang, J. Lin, and D. Kelly. Overview of the TREC 2006 Question Answering Track. In *15th Text REtrieval Conference*, Gaithersburg, MD, USA, 2006.

[9] C. Derungs and R. S. Purves. From Text to Landscape: Locating, Identifying and Mapping the Use of Landscape Features in a Swiss Alpine Corpus. *Int. J. Geogr. Inf. Sci.*, 28(6):1272–1293, June 2014.

[10] S. Edelkamp and S. Schrödl, editors. *Heuristic Search*. Morgan Kaufmann, San Francisco, 1 edition, 2012.

[11] O. Eide. *The area told as a story: An inquiry into the relationship between verbal and map-based expressions of geographical information*. PhD thesis, King's College London, 2013. uk.bl.ethos.628286.

[12] A. Ernst-Gerlach and N. Fuhr. Retrieval in text collections with historic spelling using linguistic and spelling variants. In *JCDL*, pages 333–341, Vancouver, BC, 2007.

[13] R. Hurtienne. Ein Gelehrter und sein Text. Zur Gesamtedition des Reiseberichts von Dr. Hieronymus Münzer, 1494/95 (Clm 431). *Erlanger Studien zur Geschichte*, 8:255–272, 2009.

[14] S. Kempgen. The mysterious place named Suri on Afanasij Nikitin's return journey through India. Preprint: https://opus4.kobv.de/opus4-bamberg/files/26227/SK_AfanasijNikitin_SuriseA1b.pdf, 2015.

[15] D. Kilinc. An accurate toponym-matching measure based on approximate string matching. *Journal of Information Science*, 42(2):138–149, 2015.

[16] H. Krüger. *Das älteste deutsche Routenhandbuch. Jörg Gails Raißbüchlin.* 1974. Akad. Druck- und Verlagsanstalt, Graz.

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

[18] F. Melo and B. Martins. Automated geocoding of textual documents: A survey of current approaches. *Transactions in GIS*, 2016. DOI: 10.1111/tgis.12212.

[19] J. Michael. Doppelgänger gesucht: Ein Programm für kontextsensitive phonetische Textumwandlung. *c't, Heft 25/1999*, pages 252–261.

[20] L. Moncla, M. Gaio, J. Nogueras-Iso, and S. Mustière. Reconstruction of itineraries from annotated text with an informed spanning tree algorithm. *Int. J. Geogr. Inf. Sci.*, 30(6):1137–1160, June 2016.

[21] L. Moncla et al. Geocoding for texts with fine-grain toponyms: an experiment on a geoparsed hiking descriptions corpus. In *GIS*, pages 183–192, Dallas, TX, 2014.

[22] M. K. Odell. The profit in records management. *Systems*, 20(20), 1956.

[23] J. C. Pemberton and W. Zhang. Epsilon-transformation: exploiting phase transitions to solve combinatorial optimization problems. *Artificial Intelligence*, 81(1):297–325, 1996.

[24] H. J. Postel. Die Kölner Phonetik. Ein Verfahren zur Identifizierung von Personennamen auf der Grundlage der Gestaltanalyse. *IBM-Nachrichten, 19. Jahrgang*, pages 925–931, 1969.

[25] G. Recchia and M. Louwerse. A Comparison of String Similarity Measures for Toponym Matching. In *Intl. Workshop on Computational Models of Place*, Orlando, FL, 2013. articleno. 54.

[26] H. Southall, R. Mostern, and M. L. Berman. On historical gazetteers. *International Journal of Humanities and Arts Computing*, 5(2):127–145, 2011.

[27] R. L. Taft. Name search techniques. 1970.

[28] W. E. Winkler. String Comparator Metrics and Enhanced Decision Rules in the Fellegi-Sunter Model of Record Linkage. In *Section on Survey Research Methods*, pages 354–359, 1990.

[29] C. X. Zhai, W. W. Cohen, and J. Lafferty. Beyond independent relevance: Methods and evaluation metrics for subtopic retrieval. In *SIGIR*, pages 10–17, Toronto, Canada, 2003. ACM.

[30] W. Zhang. Depth-first branch-and-bound versus local search: A case study. In *Proc. of the 17th Nat. Conf. on AI*, pages 930–935, Austin, TX, 2000.

[31] X. Zhang et al. *Disambiguating road names in text route descriptions using Exact-All-Hop Shortest Path algorithm*, volume 242 of *Frontiers in Artificial Intelligence and Applications*, pages 876–881. 2012.