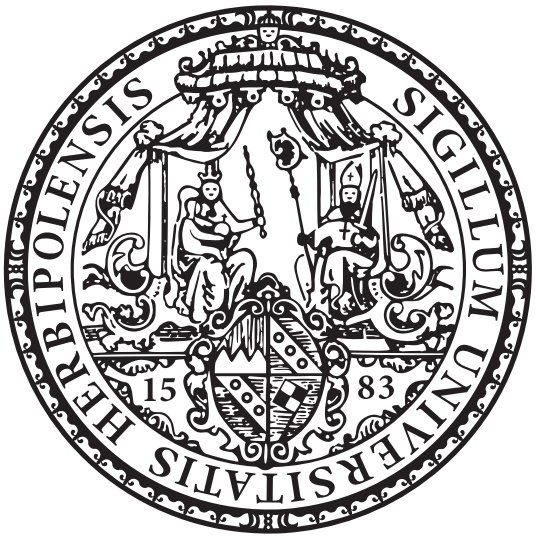
Master Thesis

Aligning Polylines to Line Features on Bitmap Images of Historical Maps

Tim Hegemann

Date of Submission: 16 November 2021
Advisors: Jun.-Prof. Dr. Thomas van Dijk
Prof. Dr. Alexander Wolff



Julius-Maximilians-Universität Würzburg
Lehrstuhl für Informatik I
Algorithmen und Komplexität

Abstract

With progress in digitalization at libraries and archives, large collections of scanned historical maps became accessible. Due to their nature as image data, they are difficult to automatically search, sort or otherwise extract information from. In order to apply artificial intelligence and machine learning annotated training data is usually required, but this is costly to produce. One solution would be crowdsourcing, but this leads to concerns about data quality.

We develop and evaluate a number of algorithmic approaches (based on computer vision, graph search and hidden Markov models) to create accurate annotations for line features like roads, waterways or contour lines based on degraded data sets or sketched human contributions. Those imprecise data are algorithmically optimized to align to the respective structural features in the scanned map with increased geometric accuracy. We compare the results to high-quality data annotated by hand (both qualitatively and quantitatively) and discuss strategies for optimal parameter estimation. Finally, we propose an interactive process for human contributors to quickly and accurately annotate historical map sources.

Zusammenfassung

Dank der fortschreitenden Digitalisierung wird immer mehr historisches Kartenmaterial aus den Sammlungen großer Bibliotheken und Archive als Scans verfügbar gemacht. Gescannte Karten, also Bilddaten, sind naturgemäß schwierig automatisiert zu durchsuchen, zu ordnen oder anderweitig auszuwerten. Ansätze, dies mit künstlicher Intelligenz und Machine Learning zu lösen, erfordern in der Regel annotierte Trainingsdaten, was sie aufwendig und damit teuer macht. Oft wird dann auf Crowdsourcing zurückgegriffen, womit man aber nicht immer eine zufriedenstellende Datenqualität erreicht.

Wir stellen eine Reihe von Algorithmen vor, basierend auf Bildverarbeitung, Suchalgorithmen und Hidden Markov Models, und zeigen deren Anwendbarkeit für die Erstellung hochwertiger digitaler Repräsentationen von Linienobjekten wie Straßen, Wasserwegen oder Höhenlinien aus unpräzisen Datensätzen oder skizzenhaften Benutzereingaben. Diese Rohdaten werden algorithmisch aufgearbeitet, sodass die in den gescannten Karten erkennbaren Strukturen geometrisch akkurat nachgebildet werden. Weiter vergleichen wir die gewonnenen Resultate sowohl qualitativ als auch quantitativ mit handoptimierten Referenzen und leiten daraus Strategien für die optimale Parameterwahl ab. Abschließend skizzieren wir einen interaktiven Prozess, der Anwendern eine schnelle und präzise Annotation historischer Karten ermöglicht.

Contents

1. Introduction 1

2. Preliminaries 3
   - 2.1. Map Matching . . . 3
   - 2.2. Fréchet Distance . . . 4
   - 2.3. Hidden Markov Models for Map Matching . . . 6
   - 2.4. The Viterbi algorithm . . . 8
   - 2.5. Local Optimization . . . 9
   - 2.6. Bresenham's line drawing algorithm . . . 11
   - 2.7. Related Work . . . 12

3. A Hidden Markov Model for Polyline-to-Raster Matching 15
   - 3.1. Emission Probabilities . . . 15
   - 3.2. Transition Probabilities . . . 17
   - 3.3. A Polyline-to-Raster Matching Algorithm . . . 19
   - 3.4. Super-Sampling the Polyline . . . 22
   - 3.5. A Fast Heuristic for Promising States . . . 23

4. Polyline-to-Raster Matching Using Uniform-Cost Search 27
   - 4.1. Uniform-Cost Search . . . 27
   - 4.2. Designing the Algorithm . . . 29
   - 4.3. Parameter Estimation . . . 32
   - 4.4. Simplifying Pixel Trails . . . 35

5. A Uniform-Cost Search Based Transition Probability Estimator 39

6. Implementation 45

7. Evaluation 47
   - 7.1. Ground Truth Data . . . 47
   - 7.2. Degraded Data . . . 49
   - 7.3. Quality Measures . . . 50
   - 7.4. Super-Sampling . . . 51
   - 7.5. Emission and Transition Probabilities . . . 52
   - 7.6. Color and Distance Scores . . . 56
   - 7.7. Linewalk: Viewport Size As Single Parameter . . . 57

7.8. Alternative Approaches for Handling Pixel Luminosity 59
7.9. Running Times 61

8. Strategies for Interactive Matching 63

9. Conclusion 67

Bibliography 69

A. Supplemental Figures 73

1. Introduction

> I wisely started with a map,
> and made the story fit.
>
> (*J. R. R. Tolkien*)

Historical maps are a great subject of study. Besides their information content, many of them, especially older ones, are magnificent pieces of art. Huge effort has been invested into creating silhouettes of the world packed with information that is both valuable by itself and in what it tells us about the look on the world their creators and their clients had. With access to affordable digitalization instruments, large inventories of historical maps have been scanned and often made available in digital libraries. As sources of information they lack the accessibility of machine readable modern maps, so acquiring the capabilities to query features of historical maps automatically or in a computer-assisted way is a major step in developing those sources.

Scanned maps are *raster data* typically (but not necessarily) stored as a two-dimensional array of pixel values. Features in modern maps such as roads or contour lines are represented as *vector data*, parametric descriptions of geometric shapes or metadata. Those features are also present in the raster data, visible to the human eye but mostly unusable for computerized processing. The *polyline-to-raster matching* problem seeks to identify a vector data representation of a feature that matches a feature in the raster data, given a close vector data feature for reference.

Many features like road and path networks, rivers and other waterways, borders, or contour lines would be represented as line strings in contemporary maps. Sequences of points that when linked together follow the intricate lines that interconnect locations and show the course of a river or the rise of a hill. These features are particularly difficult to obtain from the raster data of scanned old maps.

Example. Figure 1.1a shows a small detail of a historical map that has an apparent road feature leading in a wide arc around the city in the center from north west to south east. In Figure 1.1b the dashed gray line string shows an inaccurate representation of the road feature. Imagine a contemporary list of old milestones or a human that vaguely identified this road by three mouse clicks. The red line string is a more precise representation matching the feature on the map. Obtaining this by an algorithm with only the map and the gray line as inputs would be considered a good solution for the polyline-to-raster matching.

Above quote is from Tolkien's letter to the novelist Naomi Mitchison written in April 1954 [car00].

(a) The plain raster data.

(b) Inputs and output of a sample application.

Fig. 1.1.: Detail of a historical map with a raw sample feature (dashed gray) and a more refined one matching the raster data (solid red).

This thesis presents several algorithms to provide a simple and reliable toolkit for the polyline-to-raster matching. Whereas feature extraction is a task nowadays primarily assigned to machine learning, the lack of large annotated corpora for historical maps lets the traditional methods seem more promising. Nonetheless, the results of our semi-automatic approach can lead to the necessary conditions for machine learning to evolve on this topic.

Our tool uses a combination of image processing and map matching approaches with focus on geometric precision and runtime. The main attention lies on the optimization of vague feature descriptions. Feature identification is out of scope of this work and will be substituted by human contribution and simple heuristics. The algorithms are embedded in an intended interactive framework for supporting feature annotation and providing semi-automated solutions for complicated features.

After an introduction of the problem and brief overview over some preliminary topics, three algorithms are presented that implement the major approaches hidden Markov models, uninformed searching, and a combined method using both. Some focus falls on practical considerations in their implementation and the evaluation of their performance on historical maps as well as their running time for realistic tasks. Concluding, a strategy is proposed to combine multiple algorithms in an interactive setting and guide the user to concentrate their efforts to critical points.

2. Preliminaries

For the polyline-to-raster matching we will first look into the problem of traditional map matching. Here, both geometrical and statistical solutions have shown good results. These will be influential to our approach in solving polyline-to-raster matching. Later, we look into local optimization in the context of scanned documents and related work in the area of line features on historical maps.

2.1. Map Matching

In the classical *map matching* problem there is a sequence of points and a road map (a planar graph embedding). The problem asks for the likeliest path on the roads that the point sequence has been sampled from. The point's coordinates may contain noise such that the point does not precisely conform to a road. A path is more likely if it is close to the input points and it is valid by the means of the map topography. This is a very common problem that often has to be solved in real time, for example on navigation systems.

Example. Trajectories captured from GPS sensors are noisy by nature. The black dots in Figure 2.1 show the measured points on an imagined drive through an area of dense road crossings. Independently matching the points to the nearest road is not instrumental in finding the original path (the blue line). Although all points perfectly overlay roads, a map matching algorithm with knowledge of the underlying road network would assign them to other roads in order to resemble a valid path trough the graph induced by the map data (either the original, blue colored, or the parallel lane for the opposite direction).

There is a wide variety of solutions for this problem, especially due to the practical applications. We will further elaborate two major classes of algorithmic approaches for the map matching problem because they show relevant principles for the line-to-raster matching.

The naïve approach to this problem is a nearest neighbor matching. Each point in the input sequence is assigned to the nearest line feature of the road network. Those are connected by a shortest path algorithm or similar strategies. There is no guarantee that one can find a valid path connecting the ascribed positions. There is, of course, no guarantee that such a path existed in the first place. For incomplete or severely distorted

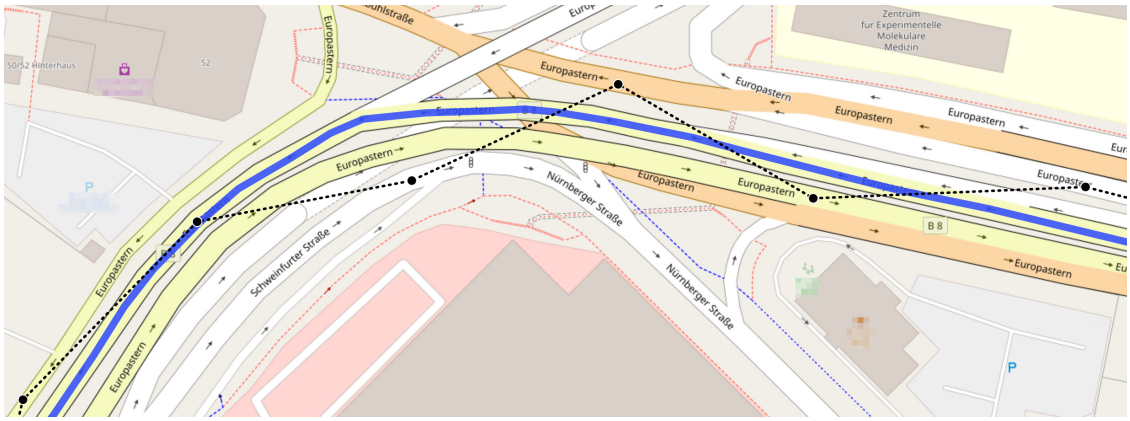


Fig. 2.1.: The measured locations (black dots) match a consistent route through the road network (blue) that can be identified by advanced map matching algorithms.

data sets even a skilled expert cannot recover the exact route due to ambiguities. One can easily imagine situations where this simple strategy fails on feasible input data like the example shown in Figure 2.1 and the roads selected as most likely are not connected or only via large detours.

More advanced methods use geometric measures of similarity to weight the possible solutions. Brakatsoulas et al. [BPSW05] introduce two such algorithms for map matching. One uses an incremental approach that chooses the next road segment based on distance and orientation with a local look-ahead to achieve a more global decision. This strategy is simple and with basically linear complexity fast and usable for online applications where input data is not available from the start but successively when the position changes.

2.2. Fréchet Distance

The other algorithm they developed based on original work by Alt et al. [AERW03] is a global approach that lists all possible paths in the road network and compare them to the path introduced by the input point sequence using the (weak) Fréchet distance [Fré06]. The original algorithm by Alt runs in $O(mn \log^2(mn))$ time where n denotes the number of sampled points and m the number of line segments in the road map. The improved algorithm proposed by Brakatsoulas is faster by a log factor, which still is significantly slower than the incremental solution.

Definition. For two curves $f, g : [0,1] \to \mathbb{R}^2$ in the plain the *Fréchet distance* δ_F is defined as follows.

$$\delta_F(f,g) := \inf_{\substack{\alpha:[0,1]\to[0,1] \\ \beta:[0,1]\to[0,1]}} \max_{t\in[0,1]} \|f(a(t)) - g(b(t))\|,$$

with α, β being continuous, surjective, and non-decreasing; $\alpha(0)$ and $\beta(0)$ both be 0, $\alpha(1)$ and $\beta(1)$ be 1. Here, $\|\cdot\|$ denotes the Euclidean norm.

The same definition without the requirement for α and β being non-decreasing is called *weak Fréchet distance*.

A common illustration for the Fréchet distance is a person walking their dog. The two curves are the paths the person and the dog walk. The parameter t is time and the functions α and β allow them to control their speed as long as they do not go backwards. (In the case of weak Fréchet distance they are allowed to go backwards). The Fréchet distance then is the length that the leash between them must have at least.

The algorithm for the computation of the Fréchet distance briefly described below was proposed by Alt and Godau [AG95]. First consider the decision variant of the global map matching problem that asks if there exists a path in the road network that has a Fréchet distance smaller than ε to the input trajectory for a fixed $\varepsilon > 0$. The optimization variant can then be solved by parametric search.

Definition. For two curves $f, g : [0,1] \to \mathbb{R}^2$ the set

$$F_\varepsilon(f,g) := \{(s,t) \in [0,1]^2 \mid \|f(s) - g(t)\| \leq \varepsilon\}$$

is called the *free space* of f and g with respect to ε. For two polylines P with n and Q and m segments the partition of $[0,n] \times [0,m]$ into regions of parametrizations in or not in the free spaces of the respective segments of P and Q is called the *free space diagram* of P and Q.

For two polylines shown in Figure 2.2 (on the left) their free space diagram with respect to ε is drawn on the right. White patterns show parametrizations in the free space for those in dark areas the distance between the segments is greater than ε. The Fréchet distance of P and Q is smaller than ε if and only if there exists a monotone path in the free space from the lower left corner $(0,0)$ to the upper right corner (n,m) of the free space diagram. Such a path is drawn exemplarily in Figure 2.2.

The concept free spaces for polylines can be generalized to planar graph embeddings with straight lines. This allows efficient calculation of the Fréchet distance for the complete road network.

The evaluations by Brakatsoulas et al. [BPSW05] show that the global matching approach based on the Fréchet distance produces better quality at the cost of runtime compared to simpler algorithms. In contrast to the naïve algorithm, the sequential character of the input samples is incorporated better into the algorithm and the space between two samples becomes relevant for the assessment of possible matchings.

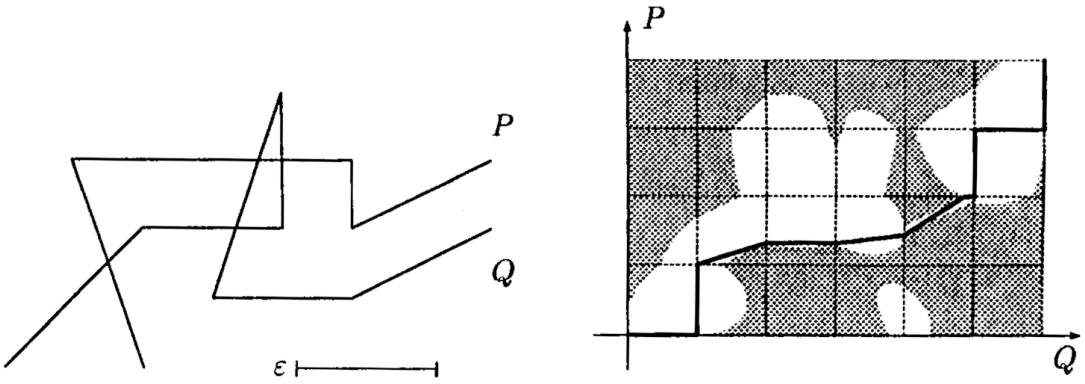


Fig. 2.2.: Free-space diagram for two sample polylines P and Q taken from the original paper by Alt and Godau [AG95], revised based on their corrections from 2003 [AERW03].

2.3. Hidden Markov Models for Map Matching

At low sampling rates the geometric similarity between the input line segments and the the original road becomes smaller. Two consecutive samples then often no longer lie on the same road segment so especially corner points produce large mismatches. In this case one can incorporate other means than geometrical similarity to find likely road combinations. Especially the transition between road sections can be a valuable source of information because some transitions are much likelier than others. For example U-turns from one road segment back on the same segment are uncommon. Also road segments that are not directly connected are less likely to explain consecutive samples.

Many algorithms use statistical methods to model problems with fuzzy success criteria. For the map matching problem Newson and Krumm [NK09] introduced an algorithm to solve this problem based on Hidden Markov Models. They improve on prior work on this approach by Hummel [Hum06] and Krumm et al. [KHL07], inasmuch as they introduce a more natural derivation for the transition probabilities from geometric characteristics of the road network. Their evualuations show the effectiveness of this approach.

Hidden Markov Models (HMM) assume a process as sequence of unobservable ("hidden") variables that each with a given probability produce an observable event and, with a known probability, pass on to the next state in the sequence. They have been widely adopted into different fields of data processing like signal processing, speech and handwriting recognition, or natural language processing. This thesis focus on their practical applications especially for map matching. A more profound introduction by Rabiner and Juang can be found at [RJ86].

The basic structure of the model architecture is shown in Figure 2.3. The hidden states are the road segments r_1 to r_n that may be chosen for the final path. The Markov process is the sequence of input samples p_1 to p_n. The model assumes that the occurrence of the road segment for any point p_i depends solely on the road segment matched for

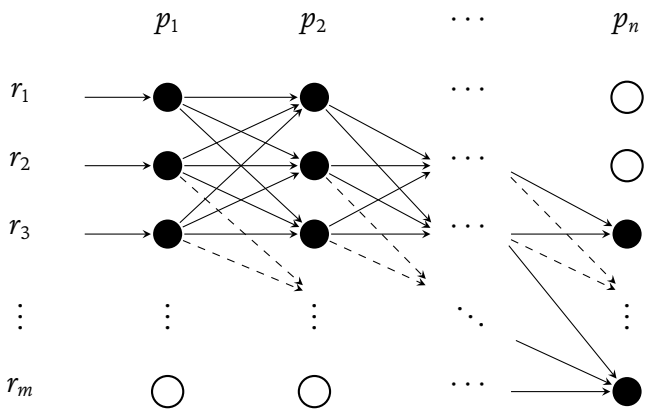


Fig. 2.3.: For each observed point p in the input sequence of length n the Hidden Markov Model considers all sensible choices for the matching road segment r out of the road network of size m. Transitions with zero probability are omitted. States that cannot be reached are white.

point p_{i-1}. This has to be taken into consideration for the *transition probabilities* that are visualized as black arrows between states in Figure 2.3. They tell how likely a subject on one road segment will continue its way on another road segment, maybe via one or more segments in between. The loose arrows on the left account for the probability of each state to occur as start of the sequence, called *start probability*. Each state additionally has an *emission probability*, that is not in the figure, and states how likely a state will occur assumed a given point has been sampled.

In Figure 2.3 not every possible transition is populated. In practice to maintain performance, infeasible transitions get a probability of zero. Those are omitted in the figure. Also road segments that are significantly distant to a sample point result in emission probabilities of zero. States that are therefore unreachable are white in the figure whereas vivid states are black. If for any sample point all states become unreachable a break in the HMM lattice occurs. For such input sequences the algorithm cannot compute a matching. Newson and Krumm list possible reasons for HMM breaks which are missing roads in the map, low probability routes, and GPS outliers. Haunert and Budig [HB12] propose a variation of the algorithm that is capable of handling incomplete road data.

For Newson's concrete map matching algorithm the probabilities are implemented as follows:

Emission probability tells how likely a road segment explains the sampling of a given point. Road segments farther from the measured point are considered more unlikely. The distance is transferred into a probability via a zero-mean Gaussian distribution with the standard deviation of GPS noise estimated by experiment.

Start probability tells how likely a road segment is at the start of a measured route. Other implementations usually assume no previous measurement and assign a uniform distribution. This algorithm instead starts at the second sample point and takes the emission probability of the first point and the current road segment as start probability.

Transition probability tells how likely matching a road segment explains the path from the previous road segment. The intuition is that the length of the path on the road network is close to the linear distance of the corresponding measurements. The shortest path therefore is considered likelier than any detour, which is explainable by the relatively short distances between consecutive points of measurement. In this case the distance is transferred into a probability via an exponential probability distribution established by a histogram over the ground truth data.

2.4. The Viterbi algorithm

Now, to find the most likely explanation for the latent variables, or concretely the most likely sequence of road segments, one must calculate the probabilities for any possible sequence. Then, the one that maximizes these probabilities can be obtained. This can be done efficiently using dynamic programming. The algorithm has been invented multiple times for different problems including famous ones for example by Needleman and Wunsch [NW70] for finding similarities between amino acid sequences or by Wagner and Fischer [WF74] for string edit distance. In the context of HMMs the formulation by Viterbi [Vit67], that originally has been intended for signal decoding, is used and the algorithm is therefore be known as Viterbi algorithm.

The method is shown in Algorithm 1. The algorithm populates two tables. One holding the probabilities of the partial sequences and one holding pointers to the most likely previous state. Typically for dynamic programming the tables are filled by a recursive formula. Each entry is either the product of start probability times emission probability, for the first column of states; or the maximum of the product of transition probability times emission probability of all previous states, for all other entries. The formulation in Algorithm 1 uses the natural variant filling the table column by column. The solution then is recovered using backtracking over the table of back pointers starting at the largest value of the last column, where the probability of the likeliest sequence is found.

The runtime of Viterbi's algorithm is dominated by the loop filling the dynamic programming table. Assuming we can look up the probabilities in $O(1)$, the total runtime is in $O(nm^2)$ for n input points and m road segments. In practice, only a minority of the road segments is considered for each state and the runtime can be further improved using multi-threading (e. g. by Song et al. [SLS+12]). Koller et al. [KWDG15] further improve the runtime of Newson and Krumm's method by replacing the Viterbi algorithm with a bidirectional Dijkstra search.

Algorithm 1: Viterbi algorithm for the map matching HMM

Input: Points $p_1, \ldots, p_n$, road segments $r_1, \ldots, r_m$, functions $f_{start}(\cdot)$, $f_{emit}(\cdot, \cdot)$, and $f_{trans}(\cdot, \cdot)$

Output: Sequence of matched road segments

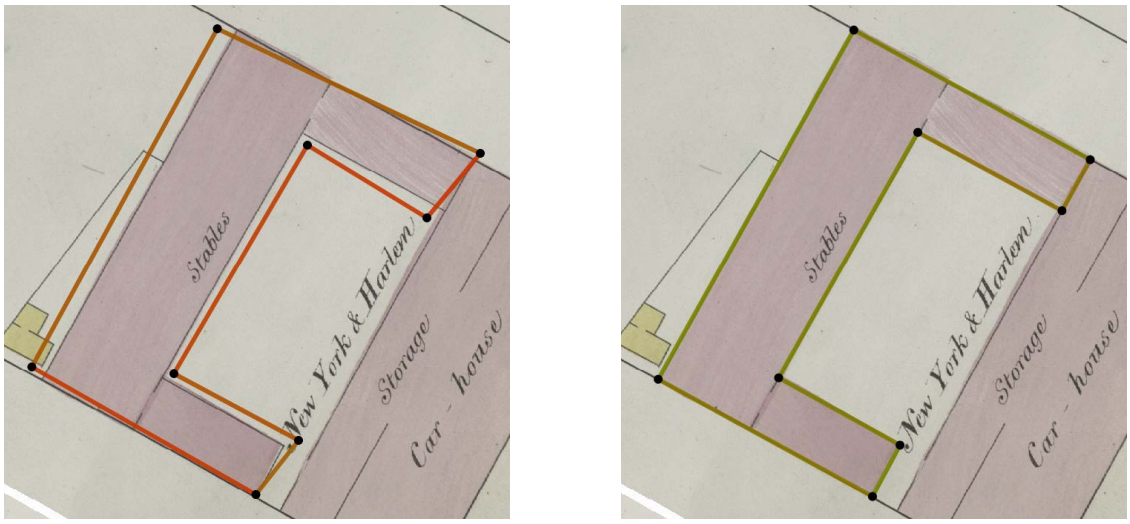
$T, B \leftarrow$ tables $m \times n$
foreach road segment r_i **do**
    $T[i, 1] \leftarrow f_{start}(r) \cdot f_{emit}(r_i, p_1)$
    $B[i, 1] \leftarrow$ **null**
foreach point $p_j, j \geq 2$ **do**
    foreach road segment r_i **do**
        $T[i, j] \leftarrow \max_{1 \leq k \leq m} \left(T[k, j-1] \cdot f_{emit}(r_i, p_j) \cdot f_{trans}(r_k, r_i) \right)$
        $B[i, j] \leftarrow \arg\max_{1 \leq k \leq m} \left(T[k, j-1] \cdot f_{emit}(r_i, p_j) \cdot f_{trans}(r_k, r_i) \right)$
$X \leftarrow$ sequence $x_1, \ldots, x_n$
$x_n \leftarrow \arg\max_{1 \leq k \leq m} (T[k, n])$
for $j = n, n-1, \ldots, 2$ **do**
    $x_{j-1} \leftarrow B[x_j, j]$
return X

Although Newson's algorithm is globally optimizing, new data can be incorporated incrementally reusing the old weights in the Viterbi table. Therefore, the algorithm can be used in online situations with only partially available data. In combination with its increased precision and reliability, these are the main reasons for its popularity.

2.5. Local Optimization

For many applications there are overall correct but inaccurate data sets that can be algorithmically optimized by unsupervised systems. One such data set, that happened to be influential for this thesis, are building footprints from a set of historical insurance atlases. Those are part of the collection of the New York Public Library and have been scanned and made publicly available[1]. The building footprints have been identified by volunteers (often called "crowdsourcing") and suffer from typical errors like geometric looseness or semantic misinterpretations. Budig et al. [BvDFA16] investigated this data set and used clustering algorithms to improve instances where multiple representations had been created—a common approach to increase data quality is to have the same entity processed by different volunteers. Van Dijk et al. [vDFH20] proposed an algorithm

[1] See `http://buildinginspector.nypl.org/`



(a) Volunteer's contribution.

(b) Algorithmically optimized version.

Fig. 2.4.: Algorithmically optimized digitalized building footprints from a crowdsourced project. Taken from van Dijk et al. [vDFH20].

based on classic local optimization techniques to improve the digitalized building footprints with and without relying on multiple representations.

Figure 2.4 shows their results. Figure 2.4a on the left shows the volunteer's contribution that correctly identifies the building's footprint but fails to accurately position the polygon's vertices on the corners of the footprint. The second Figure 2.4b on the right shows an algorithmically optimized version of the polygon on the left. The vertices are repositioned in order to have the edges line up with the outline of the building footprint.

Suppose the human contribution is semantically correct, so position and dimension of the building footprint are identified and the polygon has the correct number of vertices. The algorithm's task is now to optimize geometrical accuracy. Algorithm 2 details the optimization process. For a small environment around the vertices different positions are tried for an improved fitting by a given function. This is often called *local search* due to locality and testing different possibilities are the defining characteristics of this approach. Deliberately, a quite simple algorithm has been chosen because a globally optimizing procedure could easily run into semantically distinct structures that rank better in the rather unsophisticated measure of quality (the darkness of the pixels under the polygon outline).

Definition. The *average luminosity* (or *average darkness*) of a line segment on a bitmap image is the average luminosity (or darkness) of all pixels in the drawing of that line segment. A *line drawing* is the set of pixels out of a raster that should be selected to closely approximate a straight line.

The building footprints are printed ink on paper so one can reasonably argue that

Algorithm 2: Randomized Hill Climbing for Polygons

Input: A polygon P, a fitness function $f(\cdot)$ (higher is better), a radius r, a number n

Output: The optimized polygon

```
t ← 0
b ← f(P)
loop
    t ← t + 1
    v ← choose a random vertex from P
    Q ← P with v moved at random on a disk of radius r
    x ← f(Q)
    if x > b then
        (t, b, P) ← (0, x, Q)
while t ≤ n
return P
```

polygon edges matching the outlines should cross mostly darker pixels. For each edge in the polygon the average luminosity of pixels is calculated using Bresenham's line drawing algorithm (see Section 2.6). Then, the unweighted average over those values is subtracted from 1. Therefore the measure has a value between 0 and 1 with values closer to 1 being better.

In order to support convergence, van Dijk et al. propose some basic image manipulation. Concretely, a Gaussian blur filter applied at 50 % to the original image seems to improve both performance as well as runtime of the algorithm. The blurring enables partial solutions that do not darken the polygon on the original data but are effectively closer to the outline to get higher ranked. Therefore, they are accepted as intermediate solution and act as a bridge head to an overall better solution.

2.6. Bresenham's line drawing algorithm

Determining whether or not a pixel is covered by a straight line segment is equivalent to drawing that line on the raster. Bresenham's algorithm [Bre65] is a contribution from the very early days of computer graphics. Consequently it is simple, fast, and reliable. It takes a line segment whose start and end points must lie on the grid. Then, it walks the line pixel by pixel along the main axis direction, that is the one with the greater coordinate difference, using an integer error measure to determine when to step along in the minor axis direction.

Example. Figure 2.5 shows an example drawing the line from $(1, 1)$ to $(11, 5)$. The error measure e is drawn green above each pixel. It is all positive because the example is in

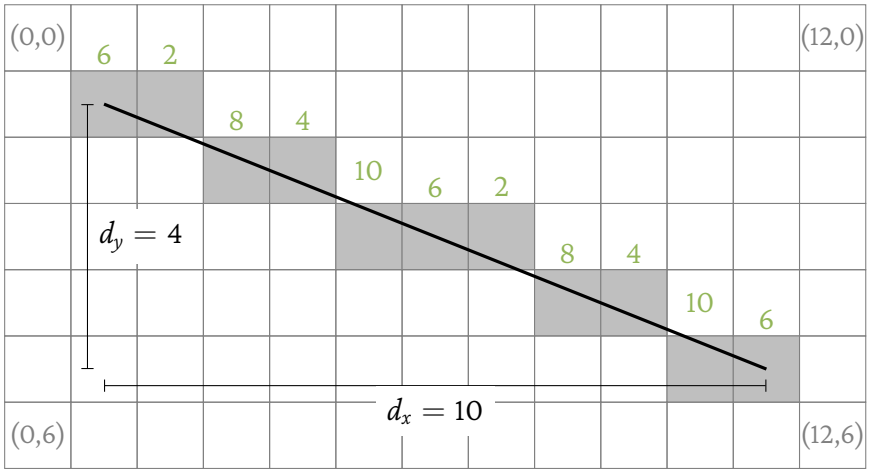


Fig. 2.5.: Line segment from $(1, 1)$ to $(11, 5)$ drawn using Bresenham's algorithm.

the first octant but can get negative if d_y is larger than d_x. The critical points are $d_x/2$ and $-d_y/2$. Because e is always greater than $-d_y/2$ in the example we do a step in x-direction for every pixel. This decreases e by d_y in every step. Each time e is greater than $d_x/2$ we do a step in y-direction and increase e by d_x. The algorithm terminates when the end of the line is reached.

Unlike the original, this thesis uses a variation of Bresenham's algorithm from a report by Zingl [Zin16]. It has a slightly different definition of the error measure and therefore can be implemented very compact, as Algorithm 3 shows. Notice that the algorithm takes the drawing function as an input. While it is intended to draw a pixel as determined by the algorithm, the information about the pixel positions can be useful beyond that. In the polygon optimizer (see Section 2.5) each invocation of *draw* initiates a query of the luminosity of the image pixel at the given position, averaging those after the whole line segment has been processed.

Bresenham's algorithm calls *draw* exactly once for each step in the major axis direction. Because $max(d_x, d_y)$ is always smaller or equal than the length of the segment the number of *draw* calls is in $O(\|s\|)$ for every segment s.

2.7. Related Work

Many of the topics covered by the previous sections must be considered related as they form the tool belt for map matching, the vector data analogy to the polyline-to-raster alignment problem. Concerning the raster part, extracting vectorized features from image data has its applications for example in road network recognition for aerial or satellite images. Hu et al. [HRF+07] use traditional image processing. After finding promising points as seeds, their algorithm tries to expand the network in different angles until

Algorithm 3: Bresenham's line drawing algorithm

Input: A line on the grid from (x_1, y_1) to (x_2, y_2), a function $draw(\cdot, \cdot)$

```
(d_x, d_y) ← (|x_2 − x_1|, −|y_2 − y_1|)
(x, y) ← (x_1, y_1)
e ← d_x + d_y
loop
    draw(x, y)
    if x = x_2 and y = y_2 then break
    if d_y ≤ 2e then
        e ← e + d_y
        x ← x + sgn(x_2 − x_1)
    if d_x ≥ 2e then
        e ← e + d_x
        y ← y + sgn(y_2 − y_1)
```

they hit road boundaries. In a concluding step the networks are pruned from implausible components. More recent approaches often focus on machine learning. A comprehensive overview of those and related techniques has been brought together by Hossain and Chen [HC19].

Chen et al. [CSV14] combined these techniques with contemporary map data to improve the junction matching performance. For the road network extraction task junctions are critical, the remaining network then can then be derived with minor effort. This is not limited to images of the earth's surface but also true for historical maps. Saeedimoghaddam and Stepinski [SS19] use convolutional neural networks (CNNs) to identify road junction points in historical maps. Despite they show impressive accurateness, those methods due to the trained machine learning models, are often hard to adapt to other features and other styles of maps.

Duan and Chiang [DC18] work on a CNN based automatic system for extracting a broader range of vector features from old maps, especially polylines of railroad descriptions. Their system shows promising preliminary results but has not yet been tested very profoundly. Duan et al. [DCL+19] later published a method for guided training data generation that uses reinforcement learning and solves a task very similar to this thesis. Their method was evaluated using comparatively accurate input data. Our method on the other hand focuses on data of lower quality. Another major difference is, that we can give concrete models, in terms of whom our algorithms are optimal.

3. A Hidden Markov Model for Polyline-to-Raster Matching

The classical map matching and the polyline-to-raster matching have much in common. Both seek to align a sequence of measured points on a planar structure that provides information about possible paths. The Hidden Markov Model based solution for map matching presented in Section 2.3 can be used as a blueprint to solve the polyline-to-raster matching. The design presented in this chapter is based on the *lineman* tool by van Dijk et al. [vDCD20]. For our toolkit we reimplemented the algorithm and proposed several improvements.

One main concern is the tradeoff between the location facts of the input polyline and the plausibility of the path given the underlying historical map. Even though the input is inaccurate (otherwise no matching would be necessary), it is of major importance as the sole source of high level information of course. The map data on the other hand is precise (we cannot undergo the map's resolution) but has conflicting features that must be differentiated. For example a path has junctions or multiple paths cross. Even more problematic, the line feature is likely overlaid with labels and other symbols. Fundamentally, feature classification is beyond the scope of this thesis. We cannot expect to avoid matching a letter as part of the path without having a classifier that sorts out such areas. Even then, the path might actually run along exactly under the label. The Hidden Markov Model can enable us to balance those information.

The logic model for the HMM is as follows. The input polyline, handled as a sequence of points, act as the observed values. The hidden variables are the real positions of the polyline's vertices that should get aligned to the actual feature on the map. There are many possible strategies to define the discrete state set for the latent variables. Given that the resolution of typical scans is not drastically finer than the printed features, a regular grid with a gap size of one pixel is a reasonable choice. So in this model each state represents a pixel.

3.1. Emission Probabilities

The emission probabilities give the likelihood that a sample resulted from a given state, based on the properties of that sample alone. For the lineman tool a pixel in the state space box explains a vertex in the polyline the better the closer it is to that sample point. So this is the probability that the vertex should actually be on the position of the pixel.

Without further knowledge about the cause of inaccuracy of our input polyline, finding a probability distribution for the displacement is difficult. As such, a one-fits-all solution seems implausible. So because we cannot assume the probability distribution for the vertex displacement, it remains as a parameter of the algorithm. Our concrete implementation by default uses a simple zero-mean Gaussian distribution with configurable standard deviation. The probability density function is scaled by $\sigma\sqrt{2\pi}$ in order to balance emission and transition probabilities. To our algorithm this is irrelevant, but formally, this is no longer a valid probability distribution for example because the values do not sum up to one. We therefore use the term *pseudo probability distribution*. The described model results in a fixed balancing for the two distributions. Usually one would introduce a factor (often called α) to regulate two influences. For ease of implementation and parameter assessment, we rely only on the standard deviation as configurable balancing factor. All examples in this chapter use the value 15 for σ that has proved to produce solid results in our informal testing.

Recall that the discrete states of the HMM are the pixels of the map. Those scans have millions of pixels. Solving the HMM for that many states is infeasible under normal time constraints. Therefore the emission probability for the vast majority of states must be zero. So assuming there is a function that for every sample point defines a mask for all the states, this partitions the state space in a small group of feasible states and a very large group with zero emission probability. We can ignore the latter in the Viterbi table saving a significant portion of computational load. For example we could ignore all states with an emission probability below a given threshold. This heuristic whether or not a state is feasible makes another parameter of the algorithm.

The concrete implementation uses axis aligned boxes of configurable size. The grid size is a tradeoff between the size of the expected error and the runtime of the algorithm. In our experiments, the area of about 25 pixels around the vertices of the polyline has proven feasible. We refer to this length as the *viewport*. Consequently, the boxes have a side length of 50 pixels. Figure 3.1 illustrates an example configuration. For each vertex the pixels with non-zero emission probabilities are in the blue framed boxes. Where the boxes overlap, our implementation considers the pixels for multiple vertices. One other idea is to cut the viewport at half the distance between the states.

For the initial states there is the start probability left to be defined. In this context the start probability says how likely the first vertex of the sequence has to be shifted to a pixel position. Again, there are many possible heuristics. For example the first vertex could be placed by color to a pixel in a neighborhood of some size. The original lineman tool assumes the first vertex to be correctly placed and therefore applies a probability of one to the pixel closest to the vertex' coordinate and zero to any other pixel. In our reimplementation we use the neighborhood defined by the state space heuristic and the pseudo probability distribution for color values that Section 3.2 introduces.

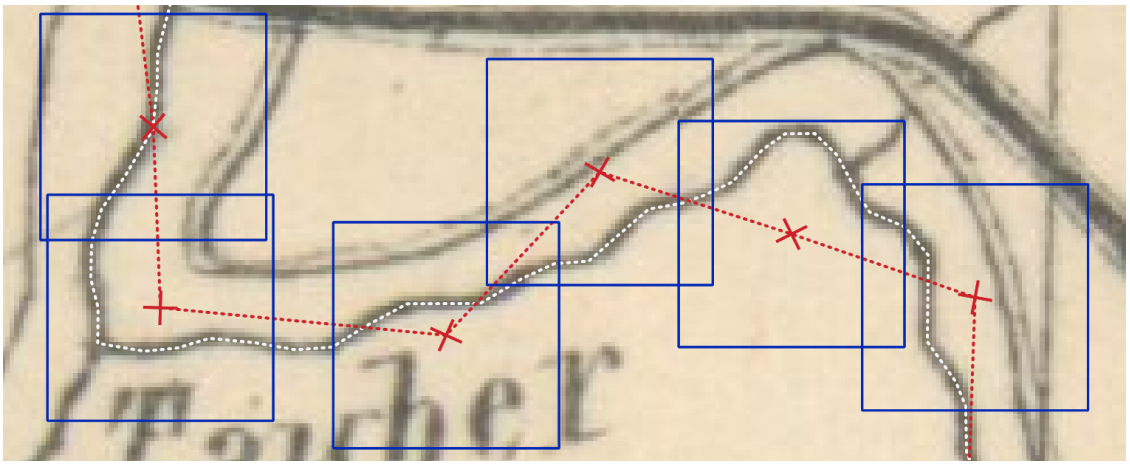


Fig. 3.1.: State space of the lineman HMM (blue boxes) for an input sequence (red). The target feature is the river Tauber (white).

3.2. Transition Probabilities

For each vertex in the input polyline there is a list of pixels considered for its final placement. The transition probabilities say how likely one pixel has a path in the map from it to any pixel in the list of the next vertex. Describing these probabilities is a complex task including more or less recognizing the feature in the map. To handle this complexity, like in the case study presented in Section 2.5, we rely on local optimization and rather simple heuristics. As long as the distance between two vertices is small, local optimization should lead to decent results. The intuition is, that the input polyline already is close to the desired feature and the local optimization will converge to this feature without erroneously switching to another undesired feature.

Therefore, we need to sketch two heuristics. One for the path that we assume between two vertices. Another for the probability of each pixel to be part of the feature or a feature of the same category. For the lineman we approximate the path using straight lines, that for small distances give a good guess and are easy to handle. Analogously to the building footprints, we assume the line feature is ink on paper so the luminosity of pixels is a good indicator whether the pixel is part of a feature or blank paper.

Figure 3.2 shows a histogram of the pixel luminosity for a sample map. It is overlaid with the density functions of three pseudo probability distributions that we designed to classify pixels. Each of them maps a luminosity value of a pixel to a probability between 0 and 1. The histogram shows a considerable negative skew. This is expected because the majority of pixels are background showing only blank paper. The pseudo probability distributions show the probability of a pixel being foreground and thus, need a turning point where to switch from paper to ink. As one can see, for this map the turning point can be far on the light side of the spectrum. Our experiments show that this is common for both historical and contemporary maps.

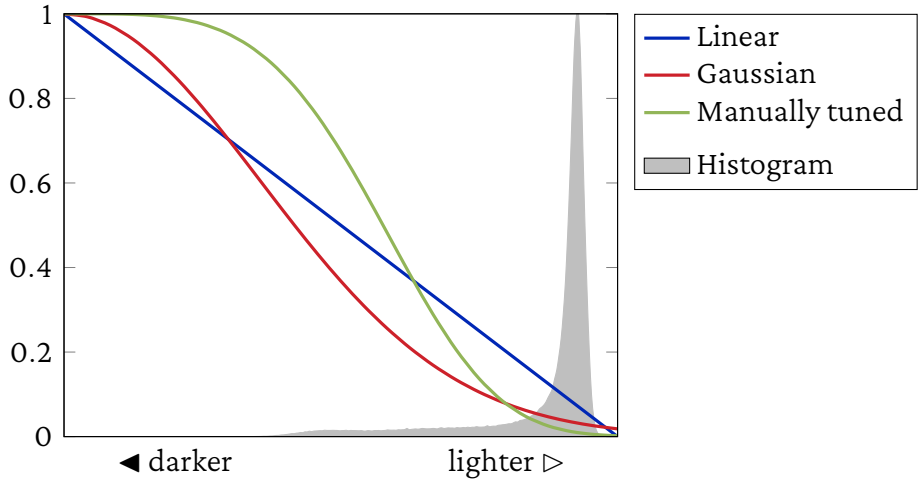


Fig. 3.2.: Different pseudo probability distributions overlaid with the histogram of pixel luminosity values for a sample map. The belly of the histogram shows the luminosity of the background pixels. Line features should be darker.

The probability density functions f of the pseudo distributions shown in the figure are defined as follows.

$$
\begin{aligned}
f_{\langle\cdot\rangle} &: [0,1] \to [0,1] \\
f_{linear}(x) &:= 1 - x \\
f_{Gaussian}(x) &:= \frac{\sqrt{\pi}}{2}\mathcal{N}(x, \mu = 0, \sigma^2 = \frac{1}{2\sqrt{2}}) = \exp(-4x^2) \\
f_{manually\ tuned}(x) &:= \exp(-2\pi x^4)
\end{aligned}
$$

with $\mathcal{N}$ being the probability density function of the normal distribution.

Note that those functions lack several critical properties of probability distributions. For our use case this is irrelevant as they are only loose descriptions of probabilities that we cannot determine exactly. Their magnitude is also of minor importance because we balance transition and emission probabilities mostly via the standard deviation parameter of the latter. Whereas the original lineman tool uses the linear form, for our experiments we use the manually tuned option, except where explicitly noted otherwise.

The final transition probability is estimated as follows. Given two states, for the line segment between them the average luminosity is calculated by drawing with Bresenhams's algorithm as described in Section 2.6. For each position the luminosity of the pixel in the underlying image is queried and they are averaged. The result is transferred into a probability using one of the aforementioned functions.

Algorithm 4: Lineman: HMM based algorithm for polyline-to-raster matching

Input: Polyline $p_1, \ldots, p_n$, functions $states(\cdot), f_{emit}(\cdot), f_{trans}(\cdot,\cdot)$
Output: Aligned polyline

$T \leftarrow$ empty list
append $\langle$pos: p_1, score: 1, ptr: **null**$\rangle$ to T
$i \leftarrow 0$
foreach point $p_j, j \geq 2$ **do**
  $\tilde{T} \leftarrow$ empty list
  foreach state s **in** $states(p_j)$ **do**
    $y \leftarrow \max_{i<k\leq T.\text{length}} \left(T[k].\text{score} \cdot f_{emit}(\|p_j - s\|) \cdot f_{trans}(T[k].\text{pos}, s)\right)$
    $k_{max} \leftarrow \arg\max_{i<k\leq T.\text{length}} \left(T[k].\text{score} \cdot f_{emit}(\|p_j - s\|) \cdot f_{trans}(T[k].\text{pos}, s)\right)$
    append $\langle$pos: s, score: y, ptr: $k_{max}\rangle$ to $\tilde{T}$
  $i \leftarrow T$.length
  append $\tilde{T}$ to T
$X \leftarrow$ empty list
$z \leftarrow \arg\max_{i<k\leq T.\text{length}} (T[k].\text{score})$
prepend $T[z]$.pos to X
while $T[z]$.ptr **not null do**
  $z \leftarrow T[z.\text{ptr}]$
  prepend $T[z]$.pos to X
return X

3.3. A Polyline-to-Raster Matching Algorithm

For the polyline-to-raster algorithm we use Viterbi's algorithm (see Section 2.4) to get the likeliest sequence for the HMM. Because of our sparse state space, this formulation uses a flat array for the Viterbi table. Algorithm 4 has the details.

The inputs are the polyline, the function that decides which states should be considered, and the functions that calculate the emission and transition probabilities. The main array can be separated into buckets, of possibly varying sizes, each containing those states of a column that have a non-zero emission probability. Each entry holds the pixel coordinate, their partial score, and a pointer to their likeliest predecessor, which saves us an additional table. Apart from that, the Viterbi algorithm is mostly unmodified.

Given the implementations described in the previous sections, the runtime can be estimated as follows. Suppose the input polyline contains n vertices and has a length of N pixels. The boxes of the *states* heuristic have a side length of m and all pairs of consecutive boxes have a distance of at most k each from their farthest corners.

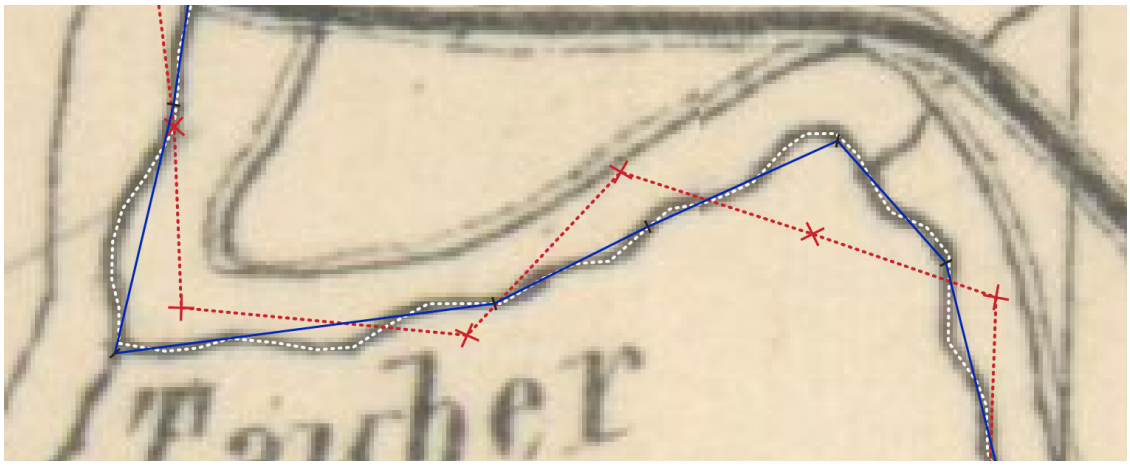


Fig. 3.3.: The HMM based optimizer applied to a historical map of the river Tauber region. For an imprecise input (red) following the river (white) an algorithmically optimized version (blue) has been established.

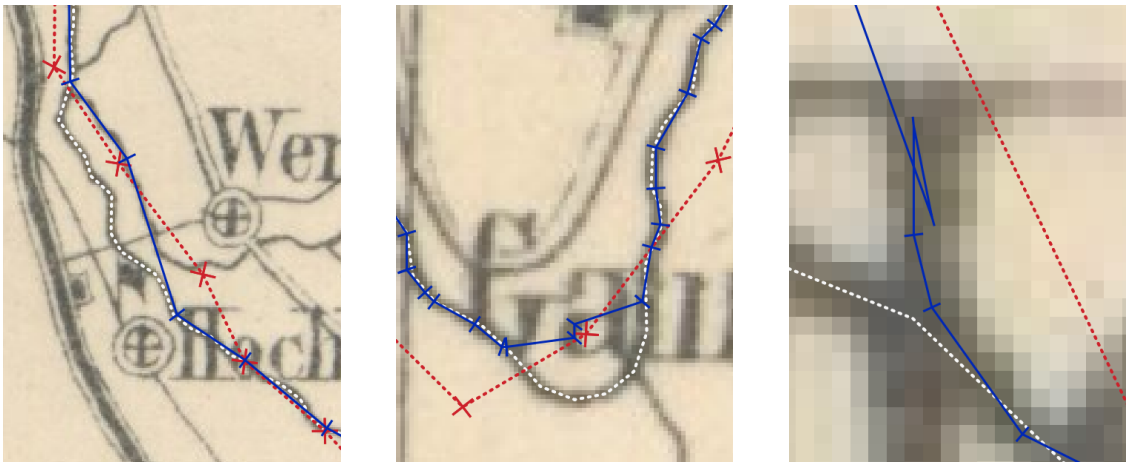
The backtracking loop building the output polyline runs at most $O(n)$ times because there are at most n buckets and each back pointer points into the preceding bucket. Therefore, it is negligible.

The main loop runs $n - 1$ times. Each iteration runs through m^2 possible states. For each possible state and each of the m^2 directly preceding states, the average luminosity of the path between them is calculated issuing k pixel queries. The emission probability can be estimated in $O(1)$. So for the complete algorithm, this gives a runtime in $O(n \cdot m^4 \cdot k)$.

This can be simplified when we look at the length of the path that we operate on. First, operating n times an operation of weight k, in this case, is equal to operating once an operation of weight $n \cdot k$. Second, the average distance of pixels in the boxes is N/n. There are as many pixels closer to the preceding box as farther away. By replacing k with N/n, we get an amortized runtime in $O(m^4 \cdot N)$.

Figure 3.3 shows the algorithm in action. Apparently, the blue line is mostly aligned with the river feature on the map. Although the detail is quite small, as we will show in Chapter 7, the algorithm works well for long input polylines as well. Looking closer into this example, one can see that the feature on the map has details that cannot be reproduced by the aligned polyline. Many river bends are cut because the algorithm cannot adapt to the level of detail of the raster feature, but is stuck at the resolution of the input polyline.

A more in-depth analysis will raise some more shortcomings of the plain algorithm outlined in the previous sections. As already mentioned, the algorithm cannot distinguish between conflicting features as long as they are not separated by background pixels. Even then, a very dark feature may overpower a short period of white pixels compared to an unsteady or only partially matched feature continuation. Figure 3.4b shows



(a) (b) (c)

Fig. 3.4.: Several examples where the algorithm fails to correctly align with the underlying feature.

an example where the letter in combination with an offset input vertex gives a better match than the correct feature. In Figure 3.4a the feature splits and the algorithm takes the wrong path later rejoining at an opportune position.

If consecutive input vertices are close together and therefore the respective state space boxes overlap, the algorithm leans to put several vertices in the same favourable position. Figure 3.4c shows this case where several vertices linger over the dark area in the middle. This can lead to undesirable loops and large straight segments with inferior alignment.

Depending on the guess for the standard deviation σ of the emission probability, the algorithm may under- or overvalue promising groups of dark pixels against the distance to the input vertex. Therefore, the correct path may be missed because the input line is too far away or a wrong path will be taken because it contains a very dark object that in fact is not part of the feature, even if it is unreasonably far off. Some parts of the feature might not get considered at all, when they are farther away form the input polyline than the boxes allow.

Growing the boxes is often not an option because of the quartic dependency between runtime and the boxes' edge length. Boxes larger than 100 pixels often result in a running time of several hours on typical consumer hardware for longer input polylines. Even for smaller boxes, as our evaluations shows, the algorithm is not fast enough for interactive settings.

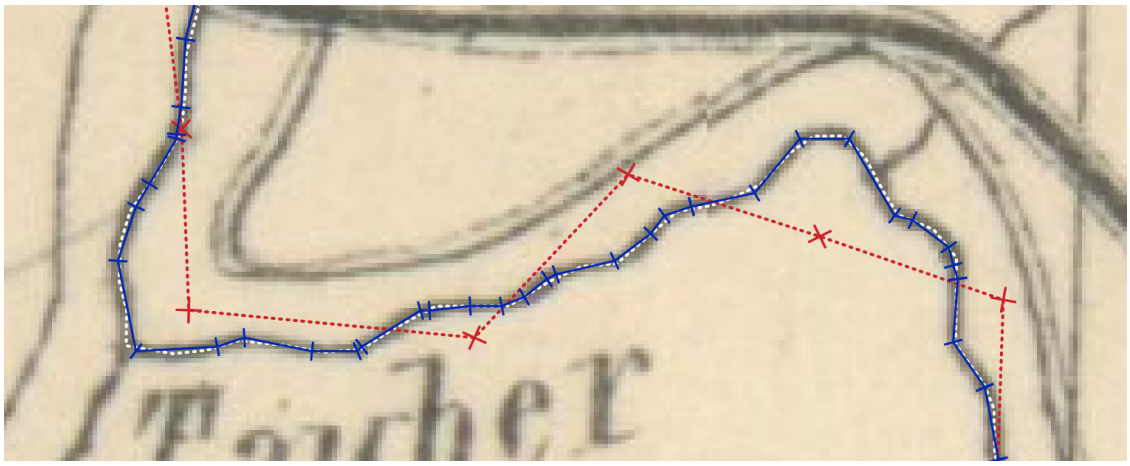


Fig. 3.5.: Invocation of the algorithm using a super-sampled input polyline that has a vertex every 7.5 pixels on average (blue). The red line shows the original input that has no artificial vertices.

3.4. Super-Sampling the Polyline

In order to precisely match the feature outline in the image the number of vertices in the input polyline often is insufficient. The original lineman tool offers a setting to add a fixed number of additional vertices at every edge. Because the edges of the input polyline may have varying lengths, we prefer to add vertices to achieve a given target edge length. For a target length k every edge that has a length l greater than $1.5k$ gets $\lfloor {}^{l}/_{k} + 0.5 \rfloor$ additional vertices evenly spaced over the line segment.

As shown in Section 3.3 the runtime of our algorithm does not depend on the number of vertices but only on the total length. Therefore, the number of additional vertices added by the subdivision procedure does not affect the runtime. Regardless, edges that are significantly smaller than the viewport become unprofitable because they will probably be placed on the same very short very dark sequence of pixels. The fitness function rewards short edges the same as long edges, so having multiple very good edges overpowers some mediocre ones.

Subdividing the input polyline in most instances gives smoother results and a better alignment. Figure 3.5 shows an example applying the lineman algorithm to the same input as in Figure 3.3 but with all edges subdivided to a target length of 7.5 pixels. This value has produced good results in development. A detailed evaluation follows in Section 7.4. The measure of 7.5 pixels apparently is fine enough to precisely reproduce the river but already shows areas where vertices are placed too close to be meaningful. This behavior seems to correlate with the distance the input vertex is off the target line. This is plausible when the vertex is in the inner side of a bend and the correct path is therefore a detour that due to is length needs more vertices than expected. This parameter obviously depends on the characteristics of the target feature. So it should be determined

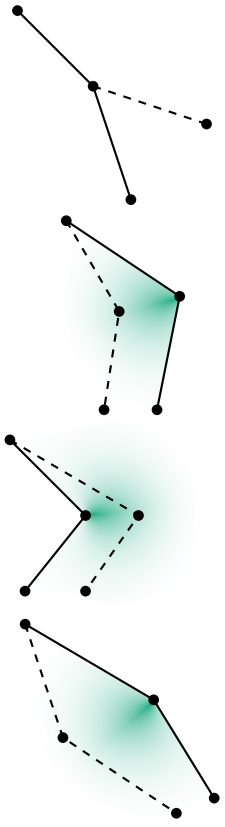
with the curvature of line on the map in mind. Nonetheless, our experiments show that it is rarely harmful to underestimate the target length of the subdivided edges.

3.5. A Fast Heuristic for Promising States

One major aspect this thesis seeks to improve is runtime performance. Recall that the runtime of lineman is in $O(m^4 \cdot N)$ for a polyline of length N and a square viewport of size m. The problematic part of this is the m^4 term. We will improve this by designing a heuristic that considers fewer states while not excessively deteriorating the alignment.

Therefore have a look at the candidates, the pixel positions the input vertices might be moved to. Given that the input polyline is sufficiently subdivided, we can assume that the segments of the optimal target polyline have almost the same lengths as the segments the algorithm is working on. As shown in Section 3.3 it is undesirable to move along the input line segments because we do not change anything—additional vertices on a straight line are superfluous—and the fitness function does not handle them well.

Assume that the previous vertex is already aligned. In the following drawings this will be the top left vertex. For the next vertex of the input polyline (solid) there are several positions it can have relatively to the optimal target polyline (dashed). The most favourable directions to move are indicated by a green shade.

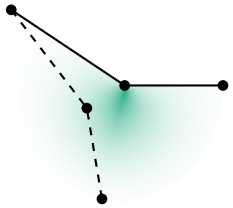


First and simplest case, the Vertices overlap. This illustrates that the pixel position of the input vertex itself should be considered a candidate. Optimally, it is not moved at all.

Both curves bend in the same direction, the inner angle of the input line faces the optimal vertex position. The area of high probably for favourable moves is in the inner angle keeping a distance to the line segments.

Both curves bend in the same direction, the inner angle of the input line opposes the optimal vertex position. The vertex has a large angle to move to always hitting the target line. Optimally it is moved towards the point of highest bending, that is most likely in the middle of the cone.

The curves bend in opposite directions, their inner angles face each other. Probably the vertex should be moved perpendicularly and again, the area of good positions is a cone tending to its middle axis.



The curves bend in opposite directions, their inner angles face different sites. This case is problematic because both lines go in different directions. Likely this will not happen because the feature should be roughly approximated by the input polyline. Still, widening the arc could improve the alignment so optimally a point outside the bow near the vertex is chosen.

Of course for the heuristic, we cannot determine which case we are in. Therefore a common denominator has to be identified. Also the heuristic should not depend on the alignment of previous point because then is has to be recalculated for each previous candidate, likely polluting the state space.

Condensing the observations for the cases outlined here, the vertex position itself is a good candidate as well as a cone both sides the vertex each with a bias towards their middle axis. Further simplifying this, for each line segment the bisector line of the angle between this and the consecutive segment seems to be both of limited complexity and covering the areas of most promising vertex positions. We will refer to this as the *bisector heuristic*. Figure 3.6 shows the viewports of this heuristic for an example application.

The analysis suggests, and our experiments confirm this, when using the bisector heuristic the algorithm significantly profits from a highly subdivided input. In the example shown in Figure 3.7 the alignment quality is on par with the original algorithm. There are only a few places where a placement of the vertex outside the bisector line would support the alignment. In exchange, the bisector heuristic is more tolerable to small subdivision, which can be beneficial on fine features.

Given a viewport of size m the bisector heuristic only considers $O(m)$ states. So the runtime of the alignment algorithm is in $O(m^2 \cdot N)$ when using this heuristic. For the recommended viewport size of 50 pixels this means the running time should decrease

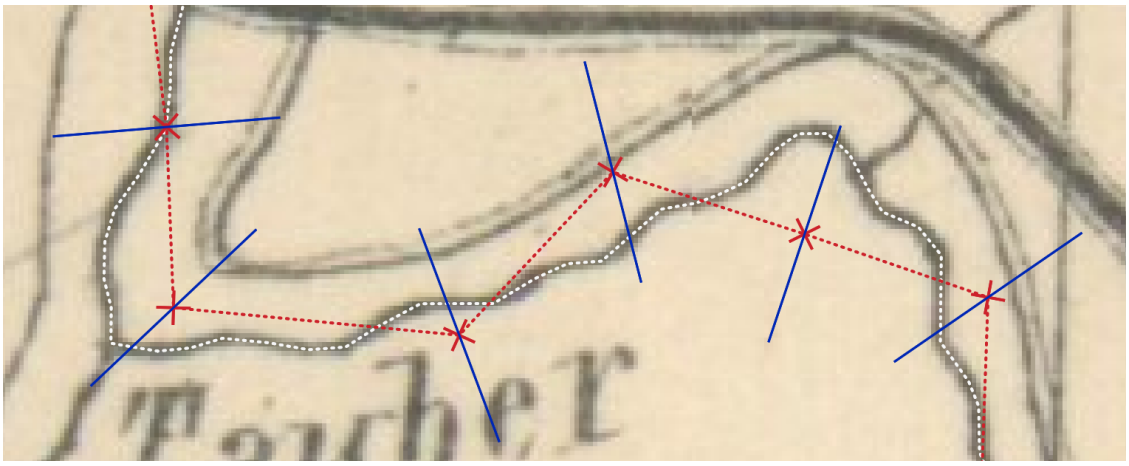


Fig. 3.6.: State space created with the faster bisector heuristic. Only pixels on the blue lines are considered as possible vertex positions.

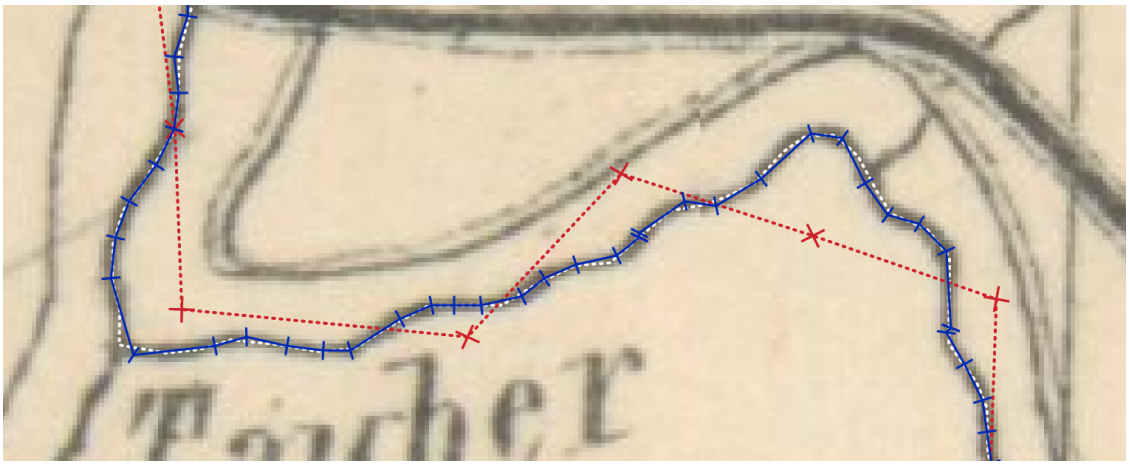


Fig. 3.7.: Alignment using a subdivided input and the bisector heuristic (blue).

by three orders of magnitude.

Assumed the manually tuned distribution for the transition probabilities works well on that particular map, for practical use there are three more parameters that we have to estimate. The target length for the input line subdivision should be chosen rather small at about twice the width of the line feature. The viewport size should be set as small as possible to maintian runtime performance but has to be greater than the expected displacement of the input polyline. The standard deviation σ of the emission probabilities should be similar to the viewport size. On instances with rather accurate input data a smaller σ will help to avoid alining to unwanted nearby features.

4. Polyline-to-Raster Matching Using Uniform-Cost Search

Our intuition is, and our experiments in Section 7.4 confirm, that subdividing the input polyline leads to significant improvements in the alignment. Both the classic and the bisector heuristic—to its credit the bisector heuristic is more reliable in this issue—show undesirable behavior when the subdivision is too eager, and the line segments become too small. Pointless accumulation of vertices and arbitrary loops being the most common side effects, that can bee seen in Figure 3.4c and at some places in Figure 3.5 and Figure 3.7 in the previous chapter.

For optimal results the features in the raster data should be reconstructed at pixel precision. Additionally, tuning the subdivision parameter is tedious and it is clearly an advantage if we could drop it. Recall what the HMM based algorithm optimizes: It looks for a repositioning of all vertices that has maximum average darkness while maintaining locality to the original polyline. At pixel precision this can be reformulated as finding a path of pixels that is dark on average and follows a polyline. Path finding is a well known field in algorithms.

4.1. Uniform-Cost Search

Finding a path through a large—or possibly infinite—number of virtual states is a fundamental task in classical artificial intelligence. Those algorithms that do not rely on information beyond identifying their goal state and expanding adjacent states are called *uninformed search strategies*. Of those, *uniform-cost search* (UCS) is a generalized form of graph search based on Dijkstra's algorithm [Dij59], although some authors like Felner [Fel11] argue that both describe the same algorithm. The polyline-to-raster matching algorithm proposed in this chapter will be a UCS variation augmented for the concrete problem domain.

Algorithm 5 has a formulation of UCS based on the on in Artificial Intelligence A Modern Approach [RN09]. Uninformed search algorithms mainly differ in the order they expand states. To *expand* here means, the state is examined whether or not it is a goal state, that successfully terminates the algorithm, and what adjacent states, also referred as *actions* that lead to other states, are accessible from this state.

Beginning at an initial state *start* the algorithm will expand states based on their *path cost*, the sum of all *step costs* for every state passed on the path beginning at *start*. This

Algorithm 5: Uniform-cost search

Input: Initial state *start*, functions *goal_test*(·), *actions*(·), and *step_cost*(·,·)
Output: The goal state or **null** if no goal state is reachable

```
1  node ← ⟨state: start, path_cost: 0⟩
2  frontier ← priority queue ordered by path_cost
3  frontier.push(node)
4  explored ← empty set
5  loop
6      if frontier is empty then return null
7      node ← frontier.pop_lowest
8      if goal_test(node.state) then return node.state
9      explored.add(node.state)
10     foreach action in actions(node.state) do
11         child ← ⟨state: action.apply(node.state), path_cost: node.path_cost +
             step_cost(node.state, action)⟩
12         if child.state not in explored or frontier then
13             frontier.push(child)
14         else if child.state in frontier with higher path_cost then
15             frontier.update(child)
```

is assured by a priority queue data structure that allows accessing the best element by the means of a given measure. When a state is expanded and it is not a goal state, the actions applicable from this state are used to generate new states that are added to the priority queue. Notice that a state is only assessed to be a goal state on expansion not on creation. This ensures that if a better path to this or any other goal state is found while expanding other states, the path costs are updated and the correct state is returned. In order to reproduce the path itself, one must introduce another data structure that for each state holds a back pointer to its preceding state. Many implementation instead achieve this by adding a third field to the records saved in the *frontier* data structure.

Provided that all step costs are positive, Uniform-cost search is optimal in general, meaning that out of any goal states reachable from *start*, UCS finds the one with lowest path costs (and therefore the shortest path) or terminates if no such state exists. Any non-positive cycle in the state graph will cause the UCS algorithm to loop indefinitely. According to Russell and Norvig [RN09], the algorithm has a space and time complexity of $O(b^{1+\lfloor C^*/\varepsilon \rfloor})$, where b is the *branching factor*, that is the maximum number of actions taken in any expansion, C^* is the path cost of the best path and ε is the minimum step cost of any state. As one can easily see, the runtime does not depend on the total number of states, which is possibly infinite anyway.

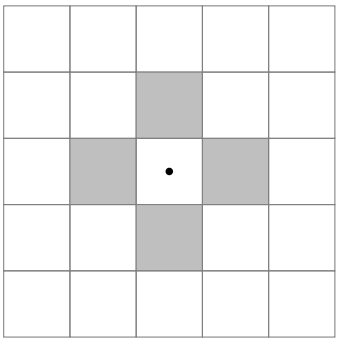
4.2. Designing the Algorithm

Suppose we search for a pixel of a given color on a bitmap image. A UCS instance for this problem might be structured as follows. The states are the pixels of the image. The initial state is not important, we use the point of origin or the upper left edge of the bounding box. The goal test for any pixel assesses if the pixel is of the wanted color. For the step costs we choose a constant function (for a constant greater than zero). When expanding a pixel, the actions are walking to any connected pixel. Thanks to the *explored* set we never expand a pixel twice so the same pixel may be suggested as new state multiple times without causing problems.

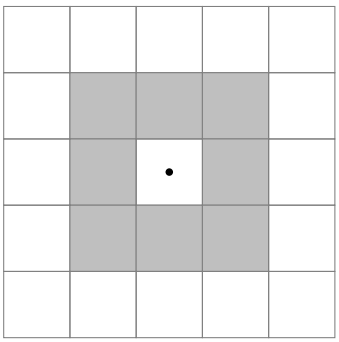
There are several concepts of connected pixels. We will account for two of them, *4-connected* or N_4 and *8-connected* or N_8. They can be defined as follows

$$N_4(x,y) := \{(x+i, y+j) \mid -1 \leq i,j \leq 1, |i| \neq |j|\}$$
$$N_8(x,y) := \{(x+i, y+j) \mid -1 \leq i,j \leq 1, \neg(i = j = 0)\}$$

for i and j being integers. Figure 4.1 illustrates those for a square set of pixels. For the pixel in the middle all the 4-connected (Figure 4.1a) respectively all the 8-connected (Figure 4.1b) pixels are highlighted.



(a) N_4



(b) N_8

Fig. 4.1.: Some types of connectivity for a regular grid of pixels

This algorithm will find a pixel of the wanted color if any such pixel exists, potentially expanding all pixels in the image. It would also find a shortest path from the initial pixel to the colorful one with respect to either the Manhattan distance (in the N_4 case) or the chessboard distance (in the N_8 case).

Add a single line segment to the picture. We seek to find the darkest path that connects start and end point of the segment without accepting too obscure detours. Change the step cost to the luminosity of the pixel plus the distance to the line scaled by a balancing factor. Now, the algorithm finds a path along the line segment without collecting more light than strictly necessary.

This strategy can be generalized to polylines. For each segment of the polyline an extra layer is introduced. The search space, so far the pixel plane, therefore becomes three-dimensional. An additional action is introduced that allows to change the layer to the next higher layer. Switching the layer means the segment of reference changes and the distance score is now calculated with respect to that segment. For the step cost of a layer change refer to Section 4.3 where all configurable parameters are discussed in detail. Figure 4.2 illustrates the search space and

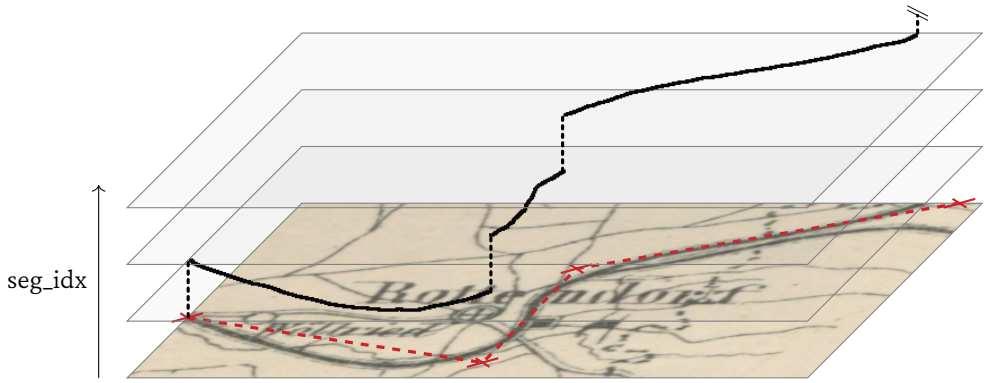


Fig. 4.2.: Schematic of the imagewalk algorithm walking through three segments (red) with layer changes (dotted black lines). The solid black line shows the produced pixel trail per layer.

how the shortest path switches layers at critical points around the vertices. The output of the algorithm is a polyline with segments of length 1 pixel (or up to $\sqrt{2}$ pixels for the N_8 case). We will call such polylines a *pixel trail*.

Algorithm 6 has the detailed instructions. The algorithm starts at the first vertex of the polyline and at the first layer. Then step by step the search space is expanded both the connected pixels and the next layer. Of course, thus far away from the next segment states in the next layer are not attractive due to their hight step costs, so the algorithm will first expand in this layer until the states are closer to the next segment. Then a layer change is beneficial. The algorithm terminates when in the last layer the last vertex of the polyline is reached and returns the pixel trail by backtracking through the states' back pointers.

Note that using different layers is crucial to the correctness of the alignment. Assume we would only use the distance to the polyline and the pixel luminosity as step costs. Then, the resulting polyline would still follow the input polyline but there is no guarantee that all segments are followed. Take a feeder road that often forms a loop with the highway. To the error model without layers it is beneficial to cut the loop because this is shorter and at least as close to the polyline as taking the loop. Even if the input does not cross itself but only has a dominant u-turn. Without the layers in between as guidance, the algorithm probably would take the short segment of light pixels in order to avoid the darker but much longer bend.

As outlined in Section 4.1 the runtime of uniform-cost search is determined by the branching factor, the path cost of the shortest path, and the minimum step cost. While this analysis is handy for infinite state spaces, this case is different. The branching factor is 9 (or 5 if we consider only 4-connected pixels). But the cost of a shortest path could be very high when large detours are necessary and the step costs can be very low for dark pixels on the polyline. With naïve definitions of path and step costs, this analysis likely

Algorithm 6: Imagewalk: UCS based polyline-to-raster matching

Input: Polyline $p_1, \ldots, p_n$, function $step\_cost(\cdot,\cdot)$
Output: The aligned polyline

```
1  node ← ⟨pos: p_1, path_cost: 0, seg_idx: 1, parent: null⟩
2  frontier ← priority queue ordered by path_cost
3  frontier.push(node)
4  explored ← empty set
5  loop
6      if frontier is empty then return null
7      node ← frontier.pop_lowest
8      if node.pos = p_n and node.seg_idx = n then return best_path(node)
9      explored.add(⟨node.pos, node.seg_idx⟩)
10     foreach step in N_8(node.pos) ∪ {CHANGE_LAYER} do
11         if step = CHANGE_LAYER then
12             child ← ⟨pos: node.pos, seg_idx: node.seg_idx + 1, parent: node,
                 path_cost: node.path_cost + step_cost(node, step)⟩
13         else
14             child ← ⟨pos: step, seg_idx: node.seg_idx, parent: node,
                 path_cost: node.path_cost + step_cost(node, step)⟩
15         if ⟨child.pos, child.seg_idx⟩ not in explored or frontier then
16             frontier.push(child)
17         else if ⟨child.pos, child.seg_idx⟩ in frontier with higher path_cost then
18             frontier.update(child)

19 Function best_path(node) :
20     path ← empty list
21     path.prepend(node.pos)
22     while node.parent not null do
23         node ← node.parent
24         path.prepend(node.pos)
25     return path
```

results in an exponential runtime. Luckily, the search space is finite.

Because no state is expanded more than once, their total number can be simply estimated by the number of pixels times the number of layers, respectively line segments. From the analysis of the traditional Dijkstra algorithm we know that on a graph of vertices V and edges E its runtime is in $O((|E| + |V|) \log |V|)$. So for a map with $w \times h$ pixels and an input polyline with n segments, $|V|$ is in $O(w \cdot h \cdot n)$. As our state space is a layered square grid graph with five edges per vertex (or nine if we consider 8-connected vertices), four (eight) edges to connected pixels plus one into the next layer, $|E|$ is in $O(|V|)$. Therefore, the runtime can be expressed as $O(x \log x)$ for x equals $w \cdot h \cdot n$.

Map scans usually have a very high resolution, so this quickly becomes unwieldy in practice. Like with the lineman algorithm, for a better runtime performance it is reasonable to limit the viewport, meaning the area where pixel are considered connected, for every segment.

If one limits the search space per layer to pixels that are in a distance of at most m around the segment, the number of states is bounded by the number of pixels in those areas. Each segment with length k counts with a rectangular area of $2m \times k$ pixels and the "line caps" with a combined area of πm^2. Summing the areas for all segments, the number of pixels is in $O(m^2 \cdot (N+n))$ for a polyline of total length N. So in the constrained case, the total runtime is in $O(x' \log x')$ for x' equals $m^2 \cdot (N + n)$. This is worse than the runtime of lineman with the bisector heuristic but considerably better than lineman with the classic heuristic.

4.3. Parameter Estimation

The previous section outlined the algorithm but left out concrete definitions for some crucial parameters. Our main concern are the step costs for the UCS algorithm. As already mentioned, these must be positive real numbers that are calculated based on a state transition. We consider three aspects. The color of a pixel that the algorithm is entering, the distance to the current line segment, and the cost we charge for a layer change.

Color Score

The component of the step costs that honors the pixel color (or more precisely pixel luminosity) is quite similar to the transition probabilities discussed in Section 3.2, so we reuse the pseudo probability distributions. They are tuned to gradually reward pixels below a threshold that is suitable for most maps, as our experiments show. Because they only produce values between 0 and 1, that should make balancing the scores easier. The manually tuned option—as well as the one derived from the normal distribution—also has the advantage that it stays positive even for completely white pixels. Therefore, even when the other partial costs are zero the step cost remains positive which is a re-

quirement of the UCS. As for the lineman algorithm we stick with the manually tuned option.

Contrary to the Viterbi algorithm that seeks to maximize probabilities, the UCS tries to minimize costs. Whereas a high probability is beneficial, high step costs are considered adverse. In order to use the pseudo probability functions for step costs, we use the negative logarithm of the probability as the edge weight. For the Viterbi algorithm, as outlined in Chapter 6, we use a similar trick (without negation) to achieve numerical stability, where we borrowed this idea.

Distance Score

The second component of the step costs weights the distance to the input polyline in order to avoid detours and switching to undesired features. It can be seen as a loose analogy to the emission probabilities discussed in Section 3.1, except that it is calculated based on the line distance between the pixel position of the state and the current segment.

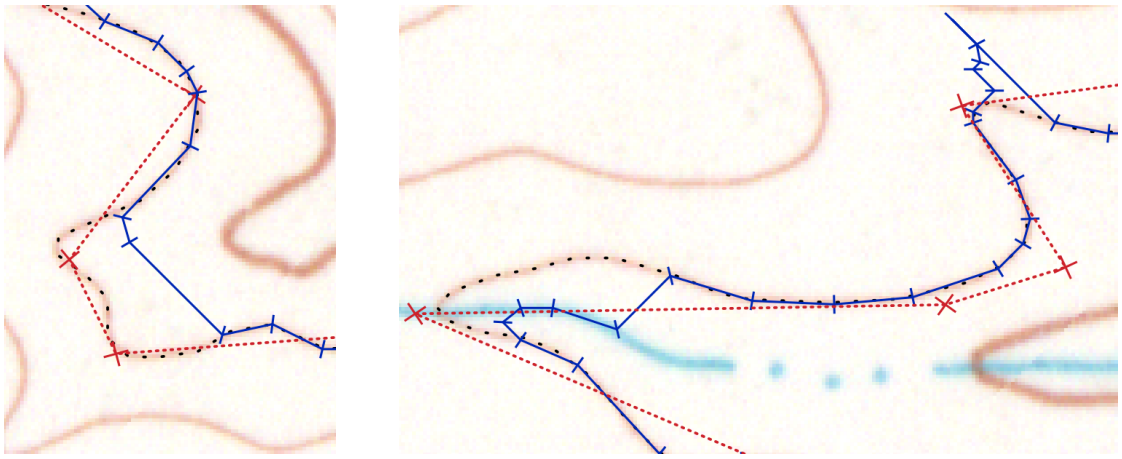
Definition. Due to its importance for some of the following algorithms, we briefly recall the definition we use for the distance between a point P and a line segment s, also referred to as *line distance*. It is the minimum of the Euclidean distances between P and Q for any point Q on s. This is equivalent to the distance to the line except for when the projection of P on the line would be outside s. Then it is the distance between P and the respective endpoint of s.

Again as we cannot assess the precision or behavior of the input polyline, a zero-mean Gaussian distribution is assumed as a first guess. To balance with the pseudo probability distribution for the color the Gaussian is scaled by $\sigma\sqrt{2\pi}$. Beside that, balancing is left to the choose for the standard deviation. As for the lineman algorithm a value of 15 has been used for all examples in this chapter. In Section 7.6 we will have a closer look at this parameter. In order to convert the probability to a weight, the same trick as for the color score is applied.

Layer Change Score

States that only change their pixel coordinates but stay within the same layer get step costs equal to the sum of color score and distance score. This corresponds to the product of the probabilities. Layer changes on the other hand are different.

Changing the layer is already considered in the distance score because when the segment of reference changes, the distance adapts correspondingly. Out of this logic an additional layer change score is optional. Ideas to have layer changes appear primarily at places where they are beneficial include adding a constant value to the distance score and adding the color score. The color score does not change when switching layers and



(a) An undesirable shortcut.

(b) Penalizing beneficial shortcuts can lead to loops (e.g. top right).

Fig. 4.3.: Two examples where the imagewalk algorithm (blue) fails to align with the underlying feature (dotted black) by taking undesirable shortcuts or loops.

including it penalizes switching layers on light pixels which follows the intuition of our fitness function.

Irrespective of which option is taken, the imagewalk algorithm has a fundamental weakness. Because each path cost is the sum of the step costs, the algorithm favors short paths. A short but light path can achieve the same score as a long, dark, and probably better one. This is not entirely bad. We do not want the algorithm to produce unnecessary detours. But it also favors unwanted shortcuts, especially on low contrast map images like the one shown in Figure 4.3. The path shown in Figure 4.3a is a good example for this. The shortcut is both light and distant to the reference line. But because the orange of the line feature—in the figure it is marked by black dots for clarity—is only slightly darker than the background, the shortcut is profitable.

The layer change score can be utilized to have the algorithm stick to a line for a number of steps that depends on the segment's length. Therefore, for every state the algorithm has to remember the number of steps along this particular line segment so far. Technically, holding this information in the state expands the state space, but we can consider states that conform in the three original coordinates as equal for the *explored* set, so the number of expanded states does not change.

With this change implemented, premature (and also late) layer changes are now penalized by the algorithm. This raises another problem. When a shortcut is beneficial, either because there are dark pixels on it or because the inaccurate input polyline likewise cuts short, the algorithm postpones consecutive layer changes until a cheap detour is possible. This leads to artifacts like the one shown in Figure 4.3b.

In our experiments the algorithm's preference for shortcuts appeared less bad than the substitutional loops when shortcuts were penalized. To the human perception shortcuts often are better than loops. Therefore, we implemented the layer change score by

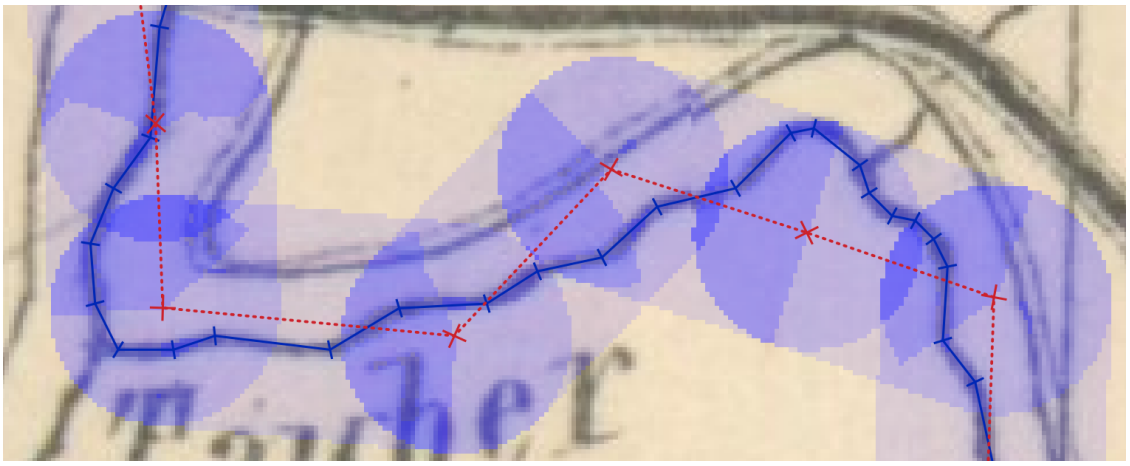


Fig. 4.4.: Heatmap of the constrained imagewalk algorithm. Pixels that are expanded are dyed blue with increasing opacity for a higher number of expansions.

a simple constant. In practice, the exact value does not seem to affect the accuracy by much so we went with a value of 1.

Cutting Costs

As discussed in Section 4.2, the runtime of the algorithm is rather high because it likely scans the whole image multiple times. In order to improve the running times, we constrained the algorithm such that for every layer only pixels that had a maximum distance of m to the line segment where considered connected. We also limited the layer changes to positions that had a maximum distance of m to the end of the segment, meaning the point where the next segment connects.

Figure 4.4 illustrates the multiple expansion of pixel states. States that get expanded are dyed blue. The color intensifies if the state is expanded multiple times. One can see half-circular artifacts around each input vertex where the algorithm "prepares" for a layer change.

Similar to the limited viewport of the lineman algorithm, alignments with the imagewalk algorithm get considerably faster when these constraints are applied. Also the limits can improve the precision of the alignment when it prevents the algorithm from jumping to unwanted features. All examples in this chapter are generated with m set to the value 25 which in pre-evaluation tests gave fast executions and visually pleasing results.

4.4. Simplifying Pixel Trails

The output of the imagewalk algorithm is a pixel trail which is not always a convenient representation. A polyline that should precisely reproduce a feature needs not neces-

Algorithm 7: Douglas-Peucker algorithm for polyline simplification

Input: Polyline $p_1, \ldots, p_n$, positive real number ε
Output: Simplified polyline

Function recurse(s, t) **:**
  $i \leftarrow \arg\max_{s<k<t}$ line_distance$(p_k, \text{Line}(p_s, p_t))$
  $d \leftarrow$ line_distance$(p_i, \text{Line}(p_s, p_t))$
  if $d > \varepsilon$ **then**
    $S \leftarrow$ recurse(s, i)
    $T \leftarrow$ recurse(i, t)
    return $S \cup \{i\} \cup T$
  else return $\varnothing$
$M \leftarrow$ recurse$(1, n)$
return $\{p_j \mid j \in (\{1\} \cup M \cup \{n\})\}$

sarily have segments of length 1 pixel. Given the nature of line features, most likely the orientation of the feature will not change often. So straight lines can resemble these without loss of accuracy.

Simplifying polylines is a well established task. For example when zooming out a digital map, the lines like streets or borders become more general and in the same way less detailed. The *line simplification* task is to identify vertices that can be removed without changing the course of the polyline by more than a given threshold. A simple but widely adopted algorithm has been discovered independently by Ramer [Ram72] as well as by Douglas and Peucker [DP73] and is traditionally named after the latter. Algorithm 7 has a detailed formulation.

Given a parameter ε, the Douglas-Peucker algorithm decides whether to keep a vertex or not based on their distance to a straight line that recursively converges to the original line. Starting with the first and the last vertex, for two vertices the algorithm defines a split node with greatest distance to the line between the two vertices. If the distance is greater ε the split vertex is fixed as part of the result and the algorithm recursively continues. Otherwise, it ignores all vertices in between. Those all have a distance below ε to the line and, because this line will be part of it, to the resulting polyline.

In the worst case the algorithm splits at every vertex and before every split looks at all vertices for the greatest distance. So for a polyline of length n the simplification can be computed in $O(n^2)$. In the best case, when the polyline is always split evenly, it finishes after $O(n \log n)$ steps.

The Douglas-Peucker algorithm is not optimal in the number of vertices it removes. One can construct instances where Douglas-Peucker retains an arbitrarily high percentage of unnecessary vertices. Think of a zigzag line of width 2ε where all vertices can be replaced by a straight line but with cleverly constructed start and end vertices, Douglas-

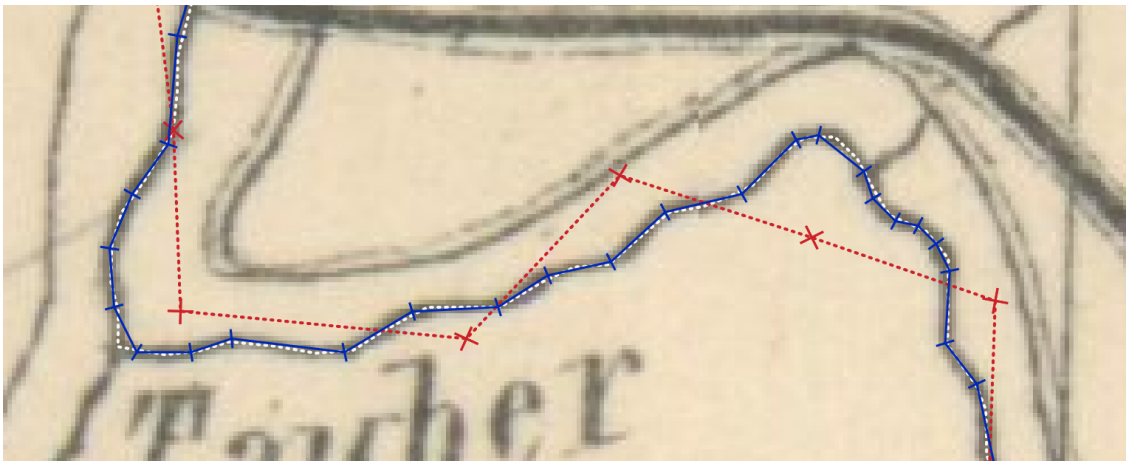


Fig. 4.5.: Simplified polyline (blue) after aligning the red polyline to the river feature (white).

Peucker will retain all but one vertices. In practical instances, this almost never occurs. Furthermore, a quick processing time for our use case is much more favorable over a optimally small number of vertices.

Because the simplification reduces the precision of the alignment, ε should be chosen as small as possible. In our experiments for the pixel trails, a relatively small value for ε is sufficient to drastically reduce the number of vertices. The example shown in Figure 4.5 uses a value of $\sqrt{2}$, the length of a pixel diagonal. While maintaining a comparable alignment quality, the number of vertices is reduced by about 30 percent compared to the subdivided lineman figures in the previous chapter.

5. A Uniform-Cost Search Based Transition Probability Estimator

Setting parameters by hand is an unappreciative task, especially when there are no intuitive features in the input that guide a good guess. The lineman algorithm needs four parameters, the luminosity distribution, the distance distribution, the size the input is subdivided into, and the viewport size. For the imagewalk instead of subdividing, a cost function for layer changes has to be identified. The probability distributions in particular are not very intuitively to select. In this section we seek to develop an algorithm based on the lineman and the imagewalk concepts that only requires one parameter that has an intuitive counterpart in the input data.

Out of the previously enumerated parameters in our perception the viewport size has the strongest connection to the dataset. Its practical implications are easy to comprehend and good to visualize, for example in a way like Figure 4.4. Particularly when the maximum error of the input polyline is known, this makes a straightforward candidate for the viewport.

A Hidden Markov Model For Choosing the Right Trail

The backbone of the algorithm is an HMM like the one in the lineman for an unchanged (that is not super-sampled) input polyline. For any polyline from a human contribution we can assume that those have a reasonable number of vertices which are strategically placed. Also for line feature from other sources these assumptions are reasonable. The hidden states again are pixels and the measurements are vertices in the input polyline. Going for an HMM has the advantage that we do not need to bother with a mechanism like the layer changes of the imagewalk. The HMM naturally fits sequential data. This is the use case it has been designed for.

For the lineman the emission probabilities of the hidden states were estimated by the distance to the input vertices. As we will use the viewport size to regulate the area of interest, the distance as a guiding attribute looses significance. Given the viewport is tight, the pixels inside should have very similar chances that they are part of the feature when their color matches. Therefore, a uniform distribution is assumed for all pixels inside the viewport. Previously, the start probabilities were estimated using the first vertex of the polyline. For uniformly distributed emissions this is the same as assuming no knowledge at all for the start probabilities. With this design, as it uses only uniform distributions, the formerly used σ parameter becomes superfluous.

Searching a Way Through Pixels

The lineman model for state transitions is inadequate for the new algorithm. Remember that the input does not get subdivided. Instead of straight lines each state transition is described by a pixel trail. In order to calculate these we modify the uniform-cost search with one line segment described in Section 4.2. Given the state transition from a state a into another state b for an input line segment s. The states are pixels. Note, that these are not the same states as the states of the HMM. We will refer to them solely as pixels to avoid confusion. We take the set of all pixels that have a line distance smaller or equal the viewport size to s as the viewport. Pixels outside the viewport are not considered connected.

The UCS starts at a; a state is a goal state if and only if it equals b. The step cost for entering a pixel is the luminosity of that pixel. Therefore, it is beneficial to enter dark pixels that are likely part of a feature. There are multiple options to define path costs. Analogously to the traditional UCS formulation one can sum up the step costs. Alternatively also averaging and taking the maximum are feasible. Analyzing the imagewalk has shown that summing the step costs often leads to alignments with undesired shortcuts due to the penalization of long trails. When averaging the pixel's luminosities, this rewards building long trails over darker, often unrelated features. Therefore the algorithm has been implemented to take the maximum with the distance to the end vertex b as a tiebreaker. All examples in this chapter use this option. Despite the particularly good results, this makes the algorithm unusable for interrupted features like dashed lines without preprocessing.

For this algorithm the transition probabilities are the only probabilities of the HMM that are not uniformly distributed. The Viterbi algorithm does not depend on the weights being probabilities; neither on their multiplication. In fact, the multiplication in practice is harmful as will be shown in Chapter 6. Therefore, the transition weights will simply be the path cost of the pixel trail that the UCS found and instead of the product their sum is optimized. For the best sequence of states, as determined by the Viterbi algorithm, the pixel trails are obtained as described in Section 4.1 and then concatenated. The resulting pixel trail can be simplified using the techniques shown in Section 4.4. Being a combination of the lineman and imagewalk concepts, we will refer to this algorithm as *linewalk*.

A Heuristic for Further Reducing the State Space

Using such an elaborate model for state transitions has a cost in complexity. To reduce the runtime not every pixel in the viewport is examined. To achieve a good running time, especially when in interactive environments, the complexity of the effective state space has to be constant. For the lineman a quadratic and a linear heuristic were presented. Consequently, both are unsuitable for this task. This one considers only five places in the viewport. The pixel closest to the input vertex as well as the darkest pixel for every

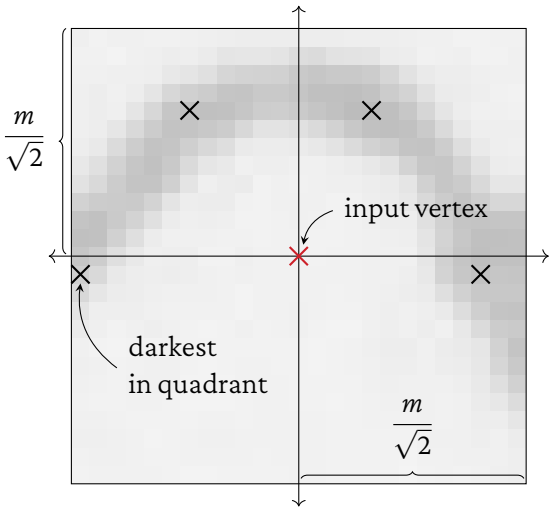


Fig. 5.1: The state space heuristic for the linewalk algorithm and a viewport of size m. For every input vertex only five states are considered.

quadrant in a coordinate originated at the vertex. Figure 5.1 illustrates the concept.

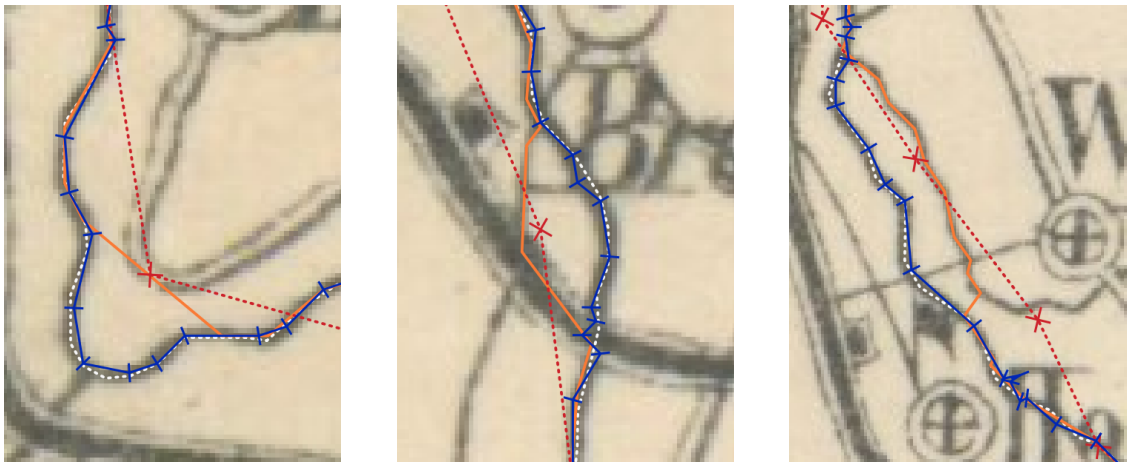
For the UCS we defined a viewport area of all pixels with constant distance m from a line segment. For the start point only this makes a circle. In order to ease implementation, only the axis aligned square inscribed into that circle is used. This is grouped into four quadrants by a coordinate system originated at the vertex. For each quadrant the darkest pixel is located. These plus the pixel closest to the vertex make up the effective state space. The four dark pixels can be computed in $O(m^2)$ once for every vertex in advance.

Complexity

For the UCS part the same calculations as for the constrained variant in Section 4.2 apply. For each segment with length k the number of pixels x is in $O(m^2 + m \cdot k)$ covering the rectangle around the segment and two halve circles on the ends. Therefore like for the imagewalk, the complexity of the UCS is in $O(x \log x)$.

Because with the proposed heuristic the effective number of states, meaning the number of states with non-zero weights, is constant, the number of state transitions is in the order of $O(n)$. As the transition costs—the replacements for the probabilities in the traditional HMM formulation—are calculated by UCS invocations, the Viterbi table only has $O(n)$ entries. With our simplified HMM structure considering only the transition costs no other calculations are necessary.

While the imagewalk additionally has layers that add up as additional factor of n, the Viterbi algorithm for the n vertices tries a constant number of states and therefore a constant number of darkest path queries. Because as stated above the runtime of the UCS per layer and per state transition is the same, both result in the same runtime complexity. Due to the repeated searches and its multi-step design, the linewalk probably has higher constant factors.



(a) (b) (c)

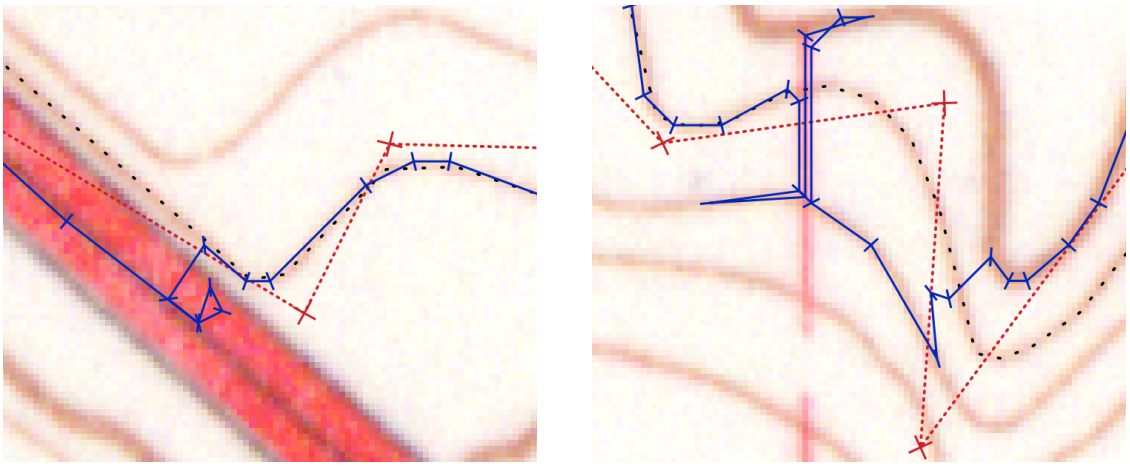
Fig. 5.2.: Example alignments by the linewalk (blue) and imagewalk (orange) algorithms of the river feature (white). The input polyline is marked red.

Results

While Chapter 3 already showed that we can improve the running times of polyline-to-raster matching by choosing proper heuristics, the problem remained that many parameters of the algorithm had to be estimated by a skilled expert with knowledge of the target map and feature. In a first step the target segment size, necessary for using super-sampling, could be replaced by introducing pixel trails. This can be seen as the limit of super-sampling but having a too detailed result is not a problem because, as showed in Section 4.4, simplification is an opportunity.

With the linewalk concept presented in this chapter, the parameters necessary to balance contrast characteristics of the map and locality of the search strategy, introduced by the emission and transition probabilities of the HMM formulation or the equivalent partial scores in the UCS, could be replaced by a simpler search strategy with only the search horizon or maximum displacement as a parameter. Despite this only works for continuous features, we consider it a substantial improvement especially because this parameter is an obvious part of the strategy and can be intuitively visualized for example as shown in Figure 4.4.

Compared to the other algorithms this one has a stronger intent to follow dark at the cost of giving up the locality score. In Figure 5.2 we see some areas where this strategy is beneficial compared to the imagewalk concept. As we can see in Figure 5.2a the imagewalk has a tendency to shortcuts that in the example is backed by the vertex position. For the lineman we deliberately chose another strategy that does not penalize detours and therefore follows the line better. Figure 5.2a show examples where the imagewalk accepts short light segments in order to follow unassociated lines that are closer to the input. Here, again the imagewalk follows the feature more consistently.



(a) (b)

Fig. 5.3.: Examples where the linewalk (blue) takes arbitrary detours over dark pixels while aligning with the contour line feature (black dots).

This strategy can lead to undesired behavior when a feature allows avoiding a light pixel by taking detours. The examples in Figure 5.3 show that. In Figure 5.3a a state has been chosen that is on a darker area on another more dominant feature and therefore the UCS forms a loop to reach it. When this happens for multiple vertices in a row like in Figure 5.3b, this can lead to multiple feature changes over the same area and therefore large detours or even loops. These cases are rare and an experienced annotator can anticipate them. For a precisely placed vertex the correct placement is always at least considered.

6. Implementation

Although the analytical considerations already gave plenty of insights into the fundamentals and properties of the presented polyline-to-raster alignment algorithms; in order to get a comprehensive comparison all algorithms were implemented as well as some auxiliary tooling to support fast and reproducible experiments. The data was prepared so that all raster data was available in the Portable Network Graphics (PNG) image format and for all vector data, particularly the polylines, the GeoJSON format was used. The latter supports polylines in the form of the `LineString` feature type. All tools were implemented in the Rust programming language, mainly because of its modern feature set and runtime efficiency. In order to support the handling of geographic primitives as well as the GeoJSON format the geo crate[1] was used. Additionally libraries for reading PNG files and for stochastic primitives were used. The source code and build instructions are provided alongside this thesis.

Implementing the algorithms that were presented or mentioned as foundational in the previous chapters mostly is a straightforward task. So this chapter in small notes will focus on those aspects where theoretical formulation and implementation diverge or where design decisions have been made that have not been previously addressed.

Bresenham's Algorithm

For the implementation of Bresenham's algorithm the simplified formulation shown in Section 2.6 is used. Traditionally the algorithm is implemented in a procedural style like the pseudocode in Algorithm 3. Instead, we use the iterator pattern to traverse the set of pixels of the line drawing. Consequently, the items of the iterator are pixel coordinates. When evaluated our iterator produces a sequence of pixels equivalent to the drawing of the input line segment. The line and therefore it's slope define the iterator instance that is constant over each invocation of the *next* function. The current position and error measure build the iterator state that changes with each invocation and is therefore used to calculate the next pixel coordinate alongside the next state and so on.

Viterbi Algorithm

In the previous chapters already different variations of the Viterbi algorithm are described. The different alignment algorithms that use HMM-like structures have most of the details presented in the respective sections. One common aspect is the handling

[1] in Rust libraries are called *crates*. For geo see `https://docs.rs/crate/geo`

of probabilities and therefore the multiplication of small numbers. For modern computation models using floating-point numerics, that has some problems with numerical precision. For the implementation of the Viterbi algorithm commonly the log probabilities are used instead the plain ones, as described for example by Slade [Sla13]. So instead of multiplying the emission and transition probabilities, for each state sequence the logarithm of each probability is taken and they are summed. Due to the monotony of the logarithm the most likely sequence in the Viterbi table using log probabilities is also the most likely one using plain probabilities.

Uniform-Cost Search or Dijkstra's Algorithm

Research into the implementation techniques of Dijkstra's algorithm, and therefore UCS, has a long history. As already mentioned, Felner [Fel11] studied the differences of the formulation known as Dijkstra's algorithm and the UCS as it is used mostly in artificial intelligence. He discovered that it is harmful to the performance when adding all vertices to the *frontier* data structure in advance. Instead he advises to only include the start vertex and add the rest on the fly as they are discovered as adjacent to expanded vertices.

Already in their early analysis of the algorithm Goldberg and Tarjan [GT96] showed that using binary heaps as a data structure for the *frontier* is expected to have a better practical performance than Fibonacci heaps even for sparse graphs. Chen et al. [CCR+07] further investigated that issue concluding that, especially for sparse graphs, one should ignore the decrease key or update operations and instead add the state another time with its updated cost and rely on the internal sorting of the heap structure. While this results in more push and pop heap operations, the overhead introduced by the decrease key (or update) operations has been larger in their experiments. Our implementation is guided by all these findings so we use a binary heap and never delete from it except by pop-ing the lowest item.

Shortest-Path Trees

While our implementation is single-source, single-target; Dijkstra's algorithm can be used to produce a *shortest-path tree*. A shortest-path tree of a graph and a source vertex is a spanning tree where every path is a shortest path in the graph from the source. Instead of using discrete Dijkstra invocations for every pair of states in the linewalk algorithm, a partial shortest-path tree can be built for every previous state such that all target states are in the tree. With the presented state space heuristic, instead of $5^2 = 25$ Dijkstra invocations only five would be necessary. This is not yet implemented in our tools, but could improve the running times for the future.

Douglas-Peucker Algorithm

Our implementation of the Douglas-Peucker algorithm is taken from the geo crate. See their *Simplify* trait for details.

7. Evaluation

7.1. Ground Truth Data

To evaluate the alignment algorithms we use scans of three different historical maps. These differ in resolution, contrast, and the colors used for the line features as well as colored parts of the background. As such, they cover a range of common styles and do not form a positive selection. High quality representations of line features as reference are difficult to obtain so a systematic review of the entire variety of map styles is a task for future research. Instead, our evaluation focuses on the effect of different algorithm parameters.

For each map a characteristic line feature was chosen that has a winding curvature in order to make the alignment more challenging. The maps are:

Würzburg The oldest map we test on is a map of the district Würzburg in Bavaria, Germany from 1885 [Wen85]. The scale of 1:200 000 is rather large and the resolution is only decent. The map is available as a digital reproduction by the Bayrische Staatsbibliothek Munich under the terms of the Creative Commons BY-NC-SA 4.0 licence.[1] The map shows mostly rural areas, except from the city of Würzburg itself that, due to the large scale, is not very present. Dominant features are larger paved roads as well as the river Main. Figure 7.1a shows a detail depicting one of the larger roads and a medium sized river. We will focus on medium sized features that are more difficult to identify, like smaller rivers and roads.

Louisville The second map is a topographical map of the Louisville quadrangle produced by the U.S. Geological Survey (USGS) [usg65] from 1965 named after the village of Louisville in Colorado, United States as part of their 7.5 minute series[2]. Digital copies of the series' maps were made available as public domain. The maps of this series have a scale of 1:24 000 and are very detailed containing many geographic features. We look particularly at the contour lines, as their curvature makes them challenging for polyline alignment. Additionally, this map is one of the older ones of the USGS and its contrast is particularly low. A detail can be found in Figure 7.1b.

[1] see `https://creativecommons.org/licenses/by-nc-sa/4.0/deed.en`

[2] Each map of the 7.5′ series is bounded by two meridians and two parallels spaced 7.5 arc minutes apart, hence the name.

(a) Würzburg district, GER 1885. (b) Louisville, Colorado, US 1965. (c) Bray, California, US 2001.

Fig. 7.1.: Details from the maps used to evaluate the alignment tools; all at the same resolution.

Bray The newest map is also a topographical map from the USGS 7.5 minute series that was made in 2001 [usg01]. It covers the Bray quadrangle, an area in the county of Siskiyou in the north of California. With the same 1:24 000 scale, it is very detailed and contains many features of excellent contrast as shown in Figure 7.1c. Again, we focus on contour lines of that plenty are present in this mountainous region.

The algorithms run on smaller parts of the maps that have been chosen representative for the complete map, because those are easier to handle especially in development and multiple evaluations can be done within a reasonable amount of time. For each map as a ground truth a line feature has been annotated by hand in a geographic information system. The features were selected by their assumed difficulty for the algorithms. They are, compared to other features on the map, relatively long and winding so the approximating polyline has to be accurately placed to cover them. In each case there is another feature nearby that is darker or more pronounced and the algorithms might be tempted to switch to that feature.

Some major parameters are outlined in Table 7.1. As discussed, the resolution of the map details is a tradeoff between covering complete features and running the evaluation within a manageable amount of time and memory. The table for each reference

Map	Resolution	Feature	Vertices	Width	Length
Würzburg	1656 × 1722	river	267	3–4 px	2792 px
Louisville	1347 × 1351	contour line	327	2 px	5367 px
Bray	800 × 790	contour line	150	1–2 px	1525 px

Tab. 7.1.: For each map a representative feature has been selected.

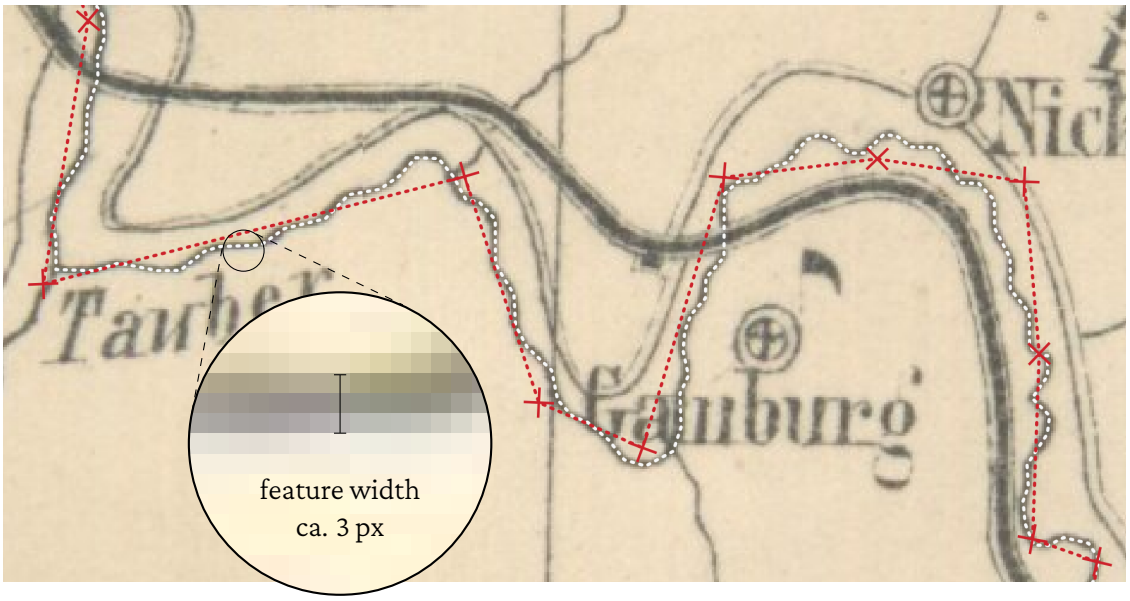


Fig. 7.2.: In order to create test inputs for the evaluation, the reference polyline (withe) was simplified with an ε of 15 px and Gaussian noise was applied (red).

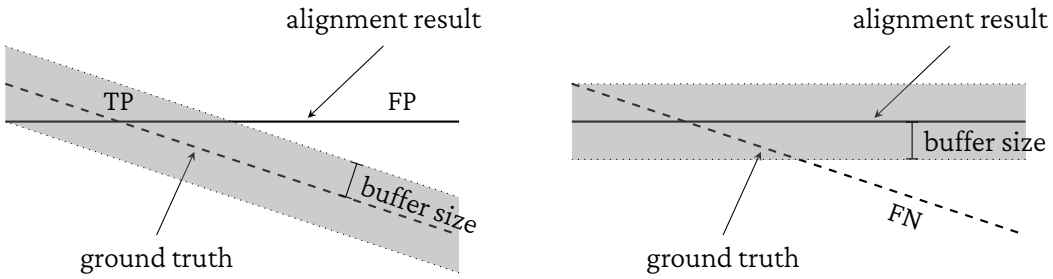
polyline has the number of vertices and the summed Euclidean length of all segments. The feature width is difficult to measure on the scans so the values are rather estimations and only apply in the average case because the features are not always painted precisely. Also only the darkest pixels were counted and about one pixel per side is of a luminosity in between the feature center and the background, so one could add another two pixels if those were counted in. Figure 7.2 has a zoomed in version of a feature on the Würzburg map where the feature width is highlighted.

7.2. Degraded Data

For the evaluation we need suitable input data to test our algorithms. While for the development a set of hand-crafted polylines is sufficient (and often instrumental in debugging), a more quantitative approach is required.

A common path towards worse data is dropping points of the polyline (e.g. used by Newson and Krumm [NK09]). Whereas in the setting of sampling GPS points at a given rate this is a plausible approach, we want to be able to quantify the accuracy of our input data. Therefore, instead of sub-sampling the reference lines, we use a simplification algorithm to get a degraded version of the polyline with constrained deviation. For convenience, the algorithm by Douglas and Peucker from Section 4.4 was reused for this task. Via the ε parameter we can regulate the degradation of the polyline per experiment.

For some instances it is undesirable to have the vertices placed perfectly on the feature because this benefits some of the algorithms but is implausible for real-world inputs.



(a) Matched alignment.

(b) Matched ground truth.

Fig. 7.3.: Matching principles based on Heipke et al. [HMWJ97].

Unfortunately, this is the case after applying the simplification algorithm. So for some experiments we additionally apply Gaussian noise to the coordinates of each vertex. Of course this further degrades the data but the parameters are chosen conservatively with $\vec{\sigma} = (0, 0)$ and $\vec{\mu} = (6\text{ px}, 6\text{ px})$. This ensures a displacement outside the feature for a significant number of vertices while avoiding a false attribution to another feature where possible. An example using noise generated with the described parameters and otherwise created by simplification of the reference polyline with an ε of 15 px can be found in Figure 7.2.

7.3. Quality Measures

In order to evaluate the quality of different methods, there have been many approaches to define numerical measures that make different strategies easily comparable. The rise of data driven science gave these an additional impetus. Especially the evaluation of binary classifiers can be an archetype for our task.

For defining the matching principles we rely on the work by Heipke et al. [HMWJ97] about evaluating road networks extracted from digital imagery. Alongside the evaluation of binary classifiers, they define *true positive* (TP) and *true negative* (TN) for values that have been correctly classified as part of the feature (*true*) and not part of the feature (*false*). Furthermore, there are two types of misclassification. *False positive* (FP) for items wrongly classified *true* and *false negative* (FN) for those wrongly classified *false*.

For an optimized polyline l and a ground truth line r, TP, FP, and FN are estimated as shown in Figure 7.3. First, a buffer of given size is sampled around r. The parts of l overlapping the buffer are considered correctly classified as part of the feature, and therefore TP. The parts outside the buffer were wrongly classified as parts of the feature, which is FP. Then, the buffer is sampled around l and the parts of r not overlapping the buffer were wrongly classified negative (FN). True Negatives are not covered by this model and therefore not considered in our evaluation.

As primary measures of quality the *correctness* (also called *precision*) and the *complete-*

ness (also called *recall*) are calculated as follows.

$$\textit{correctness} := \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}}$$

$$\textit{completeness} := \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}}$$

Instead of the measure called "quality" in Heipke's paper, for a comparison in one number we use the F_1-score [Chi92]. It was originally introduced for comparing machine text understanding approaches and is often used in information retrieval when the FN cannot be determined precisely—notice the parallels to the road extraction task. The F-score is defined as the weighted harmonic mean of precision and recall, where the weight factor of 1 means a balanced score calculated as follows.

$$\mathrm{F}_1 := \left(\frac{\textit{correctness}^{-1} + \textit{completeness}^{-1}}{2} \right)^{-1} = \frac{2\mathrm{TP}}{2\mathrm{TP} + \mathrm{FP} + \mathrm{FN}}$$

Evaluating polyline-to-raster matching is simpler than evaluating road network extraction, inasmuch as one does not need to consider roads junctions. Therefore, in contrast to Heipke et al. we calculate the buffer analytically instead of drawing the lines and counting pixels. Also, because our lines are continuous we consider the circles around every vertex into the buffer including the endings. With respect to the feature width presented in Section 7.1, we use a buffer size of 2 pixels for the evaluations in this chapter. Our reference polylines are aligned to the centerline of the feature so this covers a feature width of 4 pixels.

The polyline-to-raster alignment algorithms work on pixels. Any coordinate therefore has to be rounded to whole pixels, in our case by discarding all decimal digits. Hence, after the alignment the resulting polyline has integer coordinates that each represent a pixel. In order to emphasize that connection, before quality assessment the coordinates are moved to the pixel center by adding the vector (0.5, 0.5).

7.4. Super-Sampling

For the lineman algorithm in Section 3.4 we proposed to super-sample the input polyline. This increases the number of vertices available to the algorithm for constructing a more detailed alignment. In order to assess the effect we will look at both heuristics for the state space and different subdivisions of the input polyline. The settings of our experiment are simplified polylines with an ε of 15 and the default noise applied as described in Section 7.2. The two lineman variants use a σ of 15 and a viewport size of 25 pixels, meaning the square (or line) of the viewport has a (side) length of 50 pixels.

Table 7.2 has some details important for interpreting the results. Due to the simplification, the number of vertices is decreased compared to the reference path. Together

Map	Vertices	Average segment length		Baseline
	simplified	reference	simplified	F_1-score
Würzburg	33	10.46 px	84.60 px	24.11 %
Louisville	52	16.41 px	103.21 px	24.77 %
Bray	14	10.17 px	108.92 px	19.96 %

Tab. 7.2.: Baseline measurements for the super-sampling experiment.

with the noise applied, this makes the difficulty for the algorithms. The average segment length for the reference is between 10 and 15 pixels. So for the super-sampling we expect best results when the line is split into segments of about this length. For the simplified input the segments consequently are much longer with about 100 pixels on average. Because super-sampling at a lower rate than the input is meaningless, we stop the experiment at 233 pixels targeted segment size. At this rate no additional vertex is added.

The baseline for the algorithm is the quality rating for the degraded input lines. By chance the input overlaps parts of the reference line. Therefore the quality measures defined in Section 7.3 can be measured for these. The algorithms are expected to outperform the plain input by a large margin.

Figure 7.4 has the results. Because the baselines for all three settings were close together, only their average is included into the plot. As expected, the alignment quality increases with finer super-sampling until a maximum near the average segment length of the reference path. Then, especially on the Louisville map, the alignment quality decreases with even shorter segments. This phenomenon can be explained by the accumulation of points on very dark foreign features. This results in lots of very short but positively rated segments that outperform the correct alignment. The alignment quality decreases because rather far points are moved on the dark area leading to long segments outside the target feature.

Also the data show an advantage of the classic heuristic over the bisector heuristic when the targeted segment size is large. This advantage shrinks when the segments become shorter. As expected, the bisector heuristic is overall weaker than the classic heuristic with a notable exception for the Louisville map and short segments. With a well chosen super-sampling the differences between the heuristics are smaller than the differences between the three features.

7.5. Emission and Transition Probabilities

The major parameter for the lineman algorithm is the standard deviation of the emission probabilities. We use this to balance the influence of the locality and the luminosity of underlying pixels. For the assessment we tested the alignment quality for different

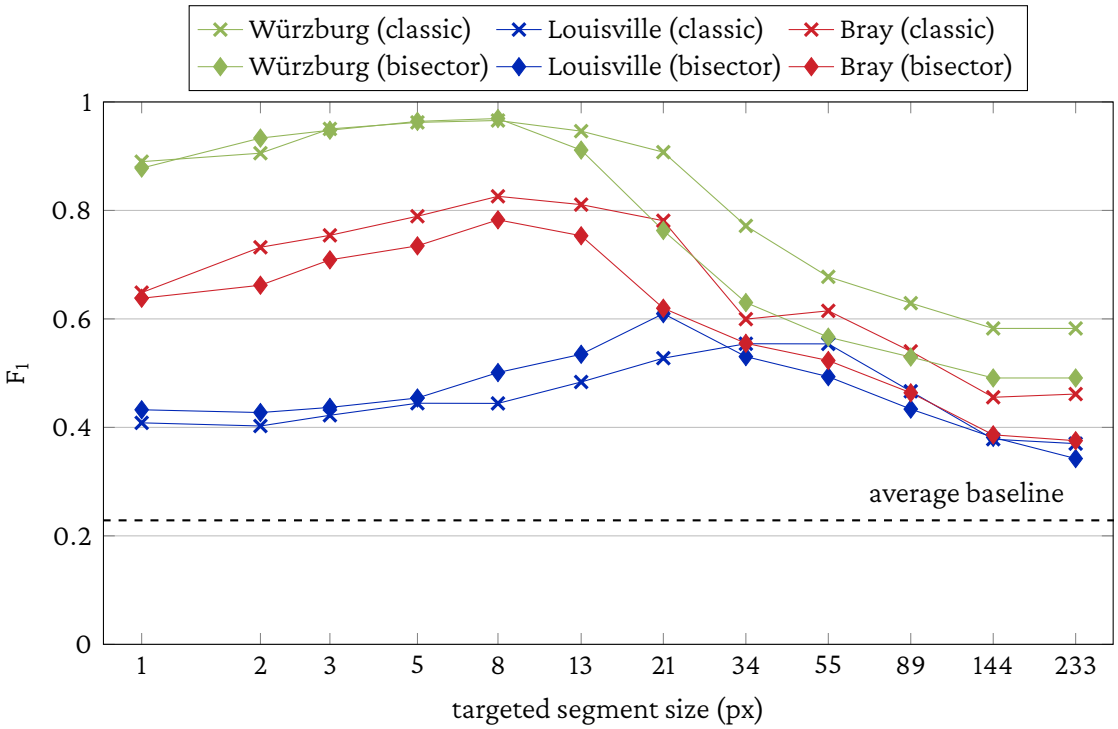


Fig. 7.4.: Alignment quality of the different lineman heuristics for super-sampled input polylines.

levels of degradation over a range of different values for σ and the viewport size. For each level of degradation we tested with and without noise and used super-sampling for a target segment length of 7.5 pixels. Using a normal distribution for modeling the locality, we scaled the viewport size along the standard deviation at a rate such that viewport size equals 2σ. Therefore, we expect that more than 95 % of all points sampled from the respective distribution would overlap the viewport. Due to its poor running times, for the classic heuristic we capped the viewport scaling at 26 pixels (therefore at $\sigma = 13$). For larger values of σ the viewport size was kept constant.

As in Section 7.4 a baseline quality can be established for all inputs. With each feature,

ε	Würzburg		Louisville		Bray	
	w/o noise	w/ noise	w/o noise	w/ noise	w/o noise	w/ noise
3	94.09 %	30.34 %	93.27 %	28.45 %	90.31 %	33.27 %
13	48.24 %	26.50 %	40.20 %	25.12 %	42.74 %	35.12 %
55	18.07 %	9.95 %	14.53 %	13.63 %	11.99 %	14.55 %

Tab. 7.3.: Baseline F_1-scores for all experiments that use different levels of simplification with and without noise.

each value for ε, and the potentially applied noise, a different input polyline is generated. For all combinations the baseline F_1-scores can be found in Table 7.3. These demonstrate the spectrum of degradation we test.

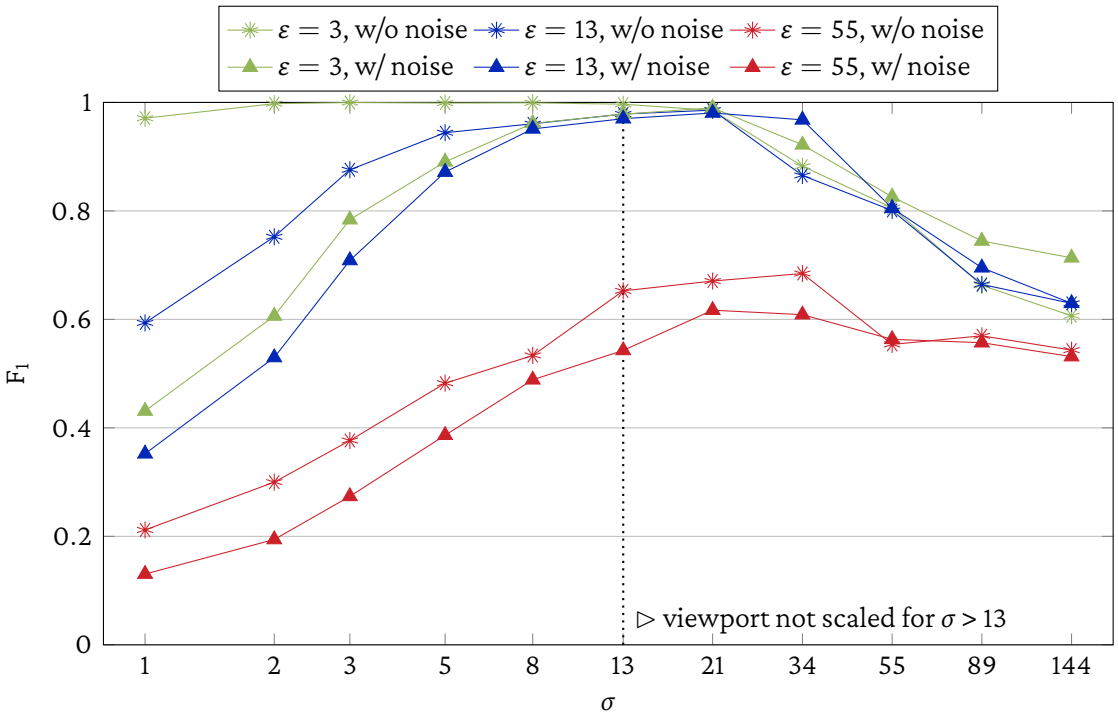
Figure 7.5a has the results for the classic heuristic and Figure 7.5b for the bisector heuristic. For the first setting with very minor displacement of at most 3 pixels by the simplification, we see, as expected, a very high alignment performance. Also we see that applying the default noise with a standard deviation of 6 is completely healed by the algorithm at a σ of 21. On the other side when σ is chosen too high the performance decreases because the algorithm chooses nearby features without honouring locality. The bisector heuristic here shows a more massive collapse which might be because at these values we already stopped scaling the viewport of the classic heuristic, due to runtime issues. Therefore, in the setting using the classic heuristic the algorithm could not choose features that far aside.

For more degraded input data the results are similar. A maximum displacement of 13 pixels could still be aligned perfectly with a similar standard deviation. The even worse setting with at most 55 pixels displacement was uncorrectable but with massive improvements when choosing appropriate parameters. Also apart from the very high values for σ, the performance characteristics were similar for both heuristics. The data show that the displacement of the input polyline has a rather small impact on the optimal value for σ. Large displacements need a higher σ than smaller but the alignment quality equally suffers from high σ values.

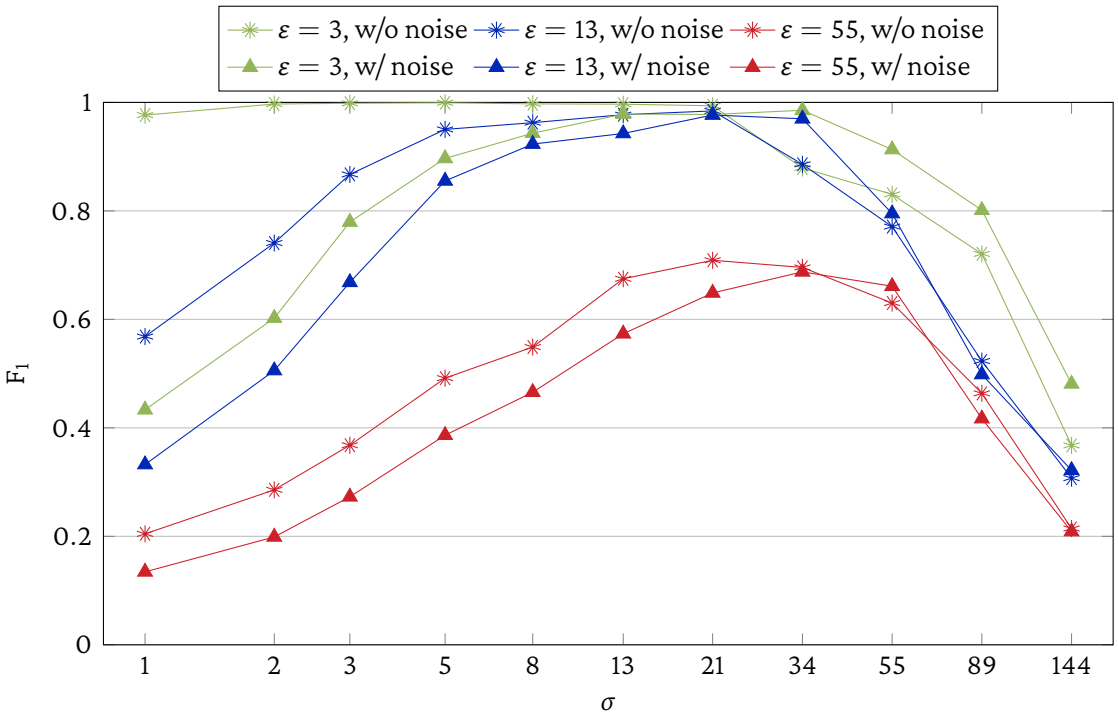
On the other hand, the point of optimality highly depends on the map. While on the Würzburg map optimal values for σ are at about 21, on the Bray map the range 8 to 13, and on the Louisville map the range 5 to 8 showed better alignments. For details see Figures A.1 to A.4 in the appendix. This can be explained by the feature density of the maps. When there are other features in range, locality becomes more important and especially contour lines, the features we focus on for the Bray and Louisville maps, by their nature are close together. But also the contrast of the maps influences the optimal choice of σ because the transition probabilities vary based on the luminosity difference of feature and background and therefore the emission probabilities become more influential when contrast is low.

Another observation is that there is a relatively large plateau of equally high quality. This makes it plausible that a close-to-optimal choice for σ can be achieved with few guesses. Especially for small displacements a user-selected value for this parameter likely results in decent alignment quality when it is chosen close to the maximum error.

Concluding, we can say that the quality difference between the two heuristics are negligible. The classic heuristic overall produces a better quality but the lead usually is below 5 % and with some settings, especially on the Louisville map, the restricted viewport of the bisector heuristic even benefits the alignment.



(a) Lineman using the classic heuristic.



(b) Lineman using the bisector heuristic.

Fig. 7.5.: Alignment quality of lineman on the Würzburg map for different values of σ in the emission probability distribution.

7.6. Color and Distance Scores

In Section 4.3 the construction of the step costs for the uniform-cost search has been introduced based on the emission and transition probabilities of the hidden Markov model based lineman algorithm. Analogously, evaluating their influence on the quality of the alignment is based on the experiments in Section 7.5. In both cases the problem is balancing locality and pixel luminosity, the measure we use to identify affiliation with a map feature. With a constant layer change score and the manually tuned distribution as color score, the only parameter influencing the step costs is the standard deviation σ of the distance score.

We test different levels of input data degradation, namely simplification with an ε of 3, 13, and 55 pixels as well as applying additional noise as described in Section 7.2. Therefore, the baselines from Table 7.3 apply. Again, the viewport size was linked to the standard distribution of the locality model at 2σ. The running times allowed a scaling over the complete experiment range of σ. The imagewalk algorithm does not need super-sampling of the input polyline, so this was not a concern. In a real-world application pixel trails are unwieldy, so we simplified the output with an ε of $\sqrt{2}$ as described in Section 4.4.

Figure 7.6 has the results for the Würzburg map. The other two setups can be found in the appendix (Figures A.5 and A.6). We see some recurring patterns. The setting without noise is, except for one outlier, consistently better than with noise, which is expected because the noise obviously lowers data quality and increases the difficulty to align the feature. The results mimic the behaviour of the lineman, that uses the same error model, with a nearly optimal alignment at a σ of 13 for the slightly and medium distorted inputs. The strongly distorted input could be aligned with decent quality and best results at a σ of about 21.

One apparent difference is the performance at higher σ values. The imagewalk completely looses track on this map as of a σ of 89. On purpose we choose a feature that is not as dominant as other features on the map so this behaviour is expected at extreme parameters. Still, the focus switch happens more often than with the lineman and once switched the algorithm tends to stay on the wrong feature for longer.

Both observations hold for the other two maps. When looking at the Louisville map, the alignment quality is better with smaller values for σ so emphasis on locality (meaning low σ) here is important. On the Bray map it is mostly the same except for the severely distorted setting where the imagewalk algorithm finds an alignment that is significantly better than with other algorithms. Here, the global search can show its strength.

Still, it is difficult to recommend a strategy for choosing σ. Like for the lineman, due to the overall high quality, especially with low quality inputs, at small displacements a value close to the expected displacement is a good guess. For maps with low contrast or features that are only lightly painted σ should be bit smaller.

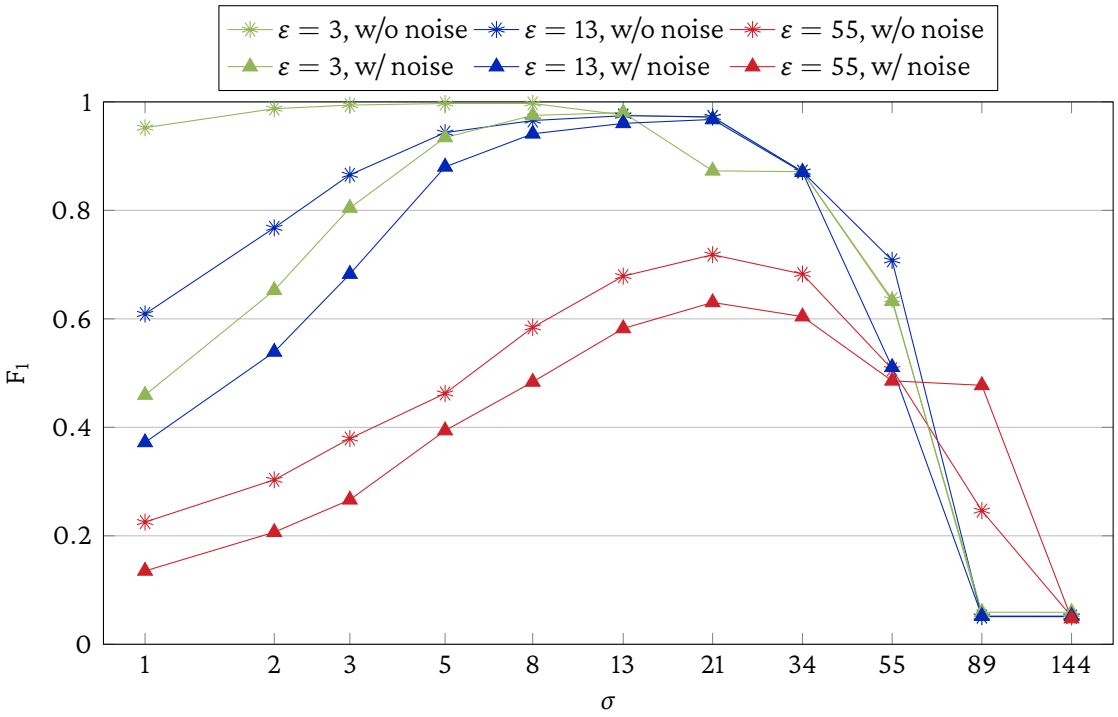


Fig. 7.6.: Alignment quality of imagewalk on the Würzburg map for different values of σ in the distance score.

7.7. Linewalk: Viewport Size As Single Parameter

As outlined in Chapter 5, we designed the linewalk algorithm to reduce the necessity of finding good sets of parameters for different settings. In the same setting as for the experiments in Sections 7.5 and 7.6 the algorithm proved a competitive alignment quality, while relying only on a single parameter that has a strong connection to measurable properties of the input. The viewport size as single parameter designates the scope we allow the input displacements to have. In our experiment this scope is limited by the ε parameter of the simplification algorithm used to create the artificial inputs. Therefore, we expect the optimal alignment quality when ε and viewport size match, or at a slightly greater viewport when additional noise is applied.

Because the input polylines are the same, the baselines from Table 7.3 can be consulted for reference. As in Section 7.6 the output pixel trail was simplified with an ε of $\sqrt{2}$ to get more convenient polylines. Figure 7.7 has the results for the Würzburg map. Again, for the other maps the respective plots can be found in the appendix (Figures A.7 and A.8).

We see that the alignment quality is more sensitive to the viewport than with the other algorithms. Considering that this is the only tunable parameter, this seems acceptable. At the respective sweet spots the alignment quality is comparable to the lineman and imagewalk algorithms, except for the severely distorted input on the Würzburg

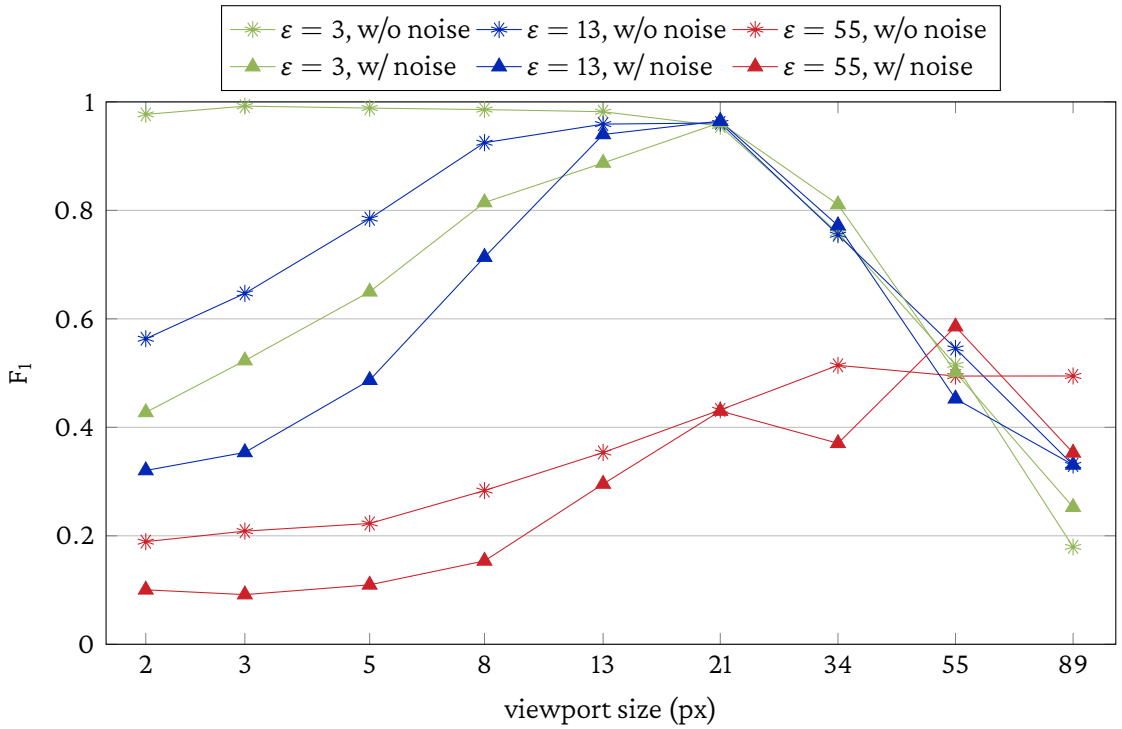


Fig. 7.7.: Würzburg map linewalk epsilon vs viewport

map. In this setting the other algorithms reach higher scores at smaller viewports. The linewalk concept does not produce good partial solutions when a continuous path of dark pixels is not present in the viewport. On the other maps the severely distorted setting where ε is 55 is better aligned by the linewalk than on the Würzburg map and it no longer falls behind the other two algorithms.

When recalling the experiment in this section as well as those from Sections 7.5 and 7.6, we often see similar results. Considering the error model those algorithms have—they all model misalignment in a similar way—this is as sign that our implementations consistently solve the task. Looking at the maximum alignment quality we see that, given an optimal selection of parameters as precisely as we can estimate them from our experiments, there are still differences. Table 7.4 shows the results when from the experiments with varying σ and viewport size parameters we choose only the best by alignment quality for two given settings on each map.

In order to select a realistic level of degradation, we opted for simplified inputs with 13 px and 55 px maximum displacement and Gaussian noise applied as described in Section 7.2. The relevant parameters can be found below the respective scores. We see every algorithm favours another map. On the Würzburg map the lineman is a little stronger. On the Louisville map the linewalk massively profits from its incentive not to leave path and on the Bray map the imagewalk leads the field.

One important observation therefore is, the lineman and imagewalk algorithms do

	Würzburg		Louisville		Bray	
ε	13	55	13	55	13	55
Lineman classic	**98.06 %**	61.69 %	68.22 %	36.58 %	85.82 %	30.21 %
σ	21	21	5	13	8	8
Lineman bisector	97.68 %	**68.76 %**	69.91 %	38.82 %	84.46 %	33.54 %
σ	21	34	5	13	8	34
Imagewalk	96.76 %	63.02 %	66.96 %	41.37 %	**96.96 %**	**73.63 %**
σ	21	21	5	13	13	55
Linewalk	96.46 %	58.56 %	**79.91 %**	**67.52 %**	87.34 %	62.58 %
viewport size	21	55	21	55	21	55

Tab. 7.4.: Alignment quality using the best parameters from the experiments from Sections 7.5 to 7.7.

not generally profit from tuning more parameters. Even when choosing the best of all runs the linewalk, that only has the viewport size as an adjustable input, does not drastically fall behind the other two algorithms. We see a very strong standing with the low displacement setting and good results with an ε of 55 on the Louisville map, where the performance overall was class leading, as well as on the Bray map where the lineman could be surpassed by a large margin. On the Würzburg map the results were not as good but with small manual adjustments to the input the results improved, so this is not a general problem with this map.

7.8. Alternative Approaches for Handling Pixel Luminosity

For all other experiments in this chapter we used a fixed setting for the transition probabilities (respectively color score). In Section 3.2 we introduced two alternative pseudo probability distributions. The one with the linear characteristic line has been used in the original lineman formulation by van Dijk et al. [vDCD20]. The second based on the Gaussian distribution is an attempt to establish a theoretic foundation for this parameter.

In order to evaluate the differences, we set up an experiment using a simplified input polyline with noise for the three maps with the default parameters of 15 for ε and (6,6) for $\vec{\mu}$. All algorithms used a σ of 15 and a viewport sized 25 pixels. For the two lineman variants the input was super-sampled to a target segment size of 8 pixels taking some values from the super-sampling experiment in Section 7.4. Having the same input parameters, the baselines from Table 7.2 also apply here.

Table 7.5 has the results. The differences are larger than expected by looking at the

Map	Algorithm	Linear	Gaussian	Manually tuned
Würzburg	Lineman classic	95.12 %	**96.81 %**	96.58 %
	Lineman bisector	94.35 %	**97.71 %**	96.95 %
	Imagewalk	88.79 %	91.01 %	**96.28 %**
Louisville	Lineman classic	**62.74 %**	53.63 %	44.42 %
	Lineman bisector	**64.19 %**	57.97 %	50.11 %
	Imagewalk	**80.55 %**	57.39 %	59.92 %
Bray	Lineman classic	69.97 %	64.25 %	**82.60 %**
	Lineman bisector	75.10 %	66.18 %	**78.27 %**
	Imagewalk	81.76 %	74.69 %	**83.63 %**

Tab. 7.5.: Alignment quality using different pseudo probability distributions for the transition probabilities or color score.

plot in Figure 3.2. We see that the manually tuned variant performs good on both the Würzburg and the Bray map, but the Louisville map again is a difficult candidate. Especially the combination of the imagewalk algorithm and the linear pseudo distribution is an outlier, to the positive but still, that is very sensitive to parameter changes.

From the experiments in Sections 7.5 and 7.6 we know, that the value of 15 for σ is not a good choice and the alignment quality benefits from lower values. When repeated with a σ of 8, the F_1-scores settle between 67 % and 79 % and the large discrepancies no longer occur.

When designing the linewalk we used the maximum luminosity instead of average luminosity. Therefore staying on a path becomes more important for the score than the absolute darkness. This approach can be used also for the lineman's state transitions. Figure 7.8 has the results as difference in the F_1-score between using maximum and average luminosity. Positive values mean the maximum luminosity and negative values mean the average luminosity resulted in a better alignment quality.

Although the negative effect clearly dominates, it highly depends on the map and the length of the segments. For very small segments the differences become insignificant because the scores can easily be compensated by other segments. When in the supersampling the target segment size is chosen longer, this changes. While alignments on the Bray map become significantly worse, this effect is not as distinct on the other maps. On the Louisville map, and only when the bisector heuristic is used, taking the maximum luminosity increases alignment quality for segments of small to medium length. For longer segments maps with high contrasts seem favour average luminosity more than low contrast maps. For shorter segments the differences are negligible with the bisector variant on the Louisville map as an outlier.

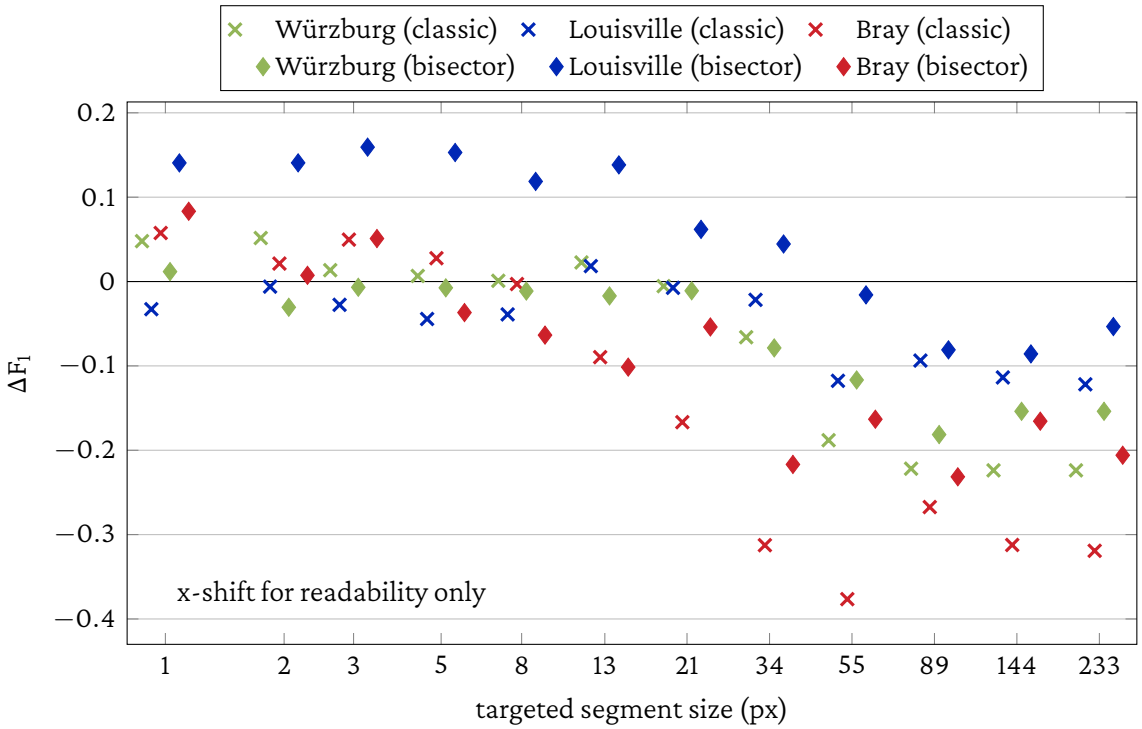


Fig. 7.8.: Difference in alignment quality when using maximum luminosity instead of average luminosity for scoring state transitions.

7.9. Running Times

One major concern of the original lineman was that it is rather slow, so improving on that was a critical design objective for all the algorithms. All performance testing was done using a workstation running Fedora Linux 34 with Kernel 5.14.11 on an AMD Ryzen 9 3900X with 64 GiB of memory. In order to use independent tooling, we measured the complete time from starting the executable until its termination. This includes overhead for loading the map which was 63.4 ms for the Würzburg map, 37.4 ms for the Louisville map, and 16.6 ms for the Bray map. For consistent results each experiment has been repeated at least 10 times in a row (or for at least 1s when the running time was very short) and the average of all runs has been taken. We paid special attention to the variance of the runs rejecting all experiments that may have been influenced by external events.

As we now from the theoretical considerations, the runtime of the algorithms, due to its asymptotic behaviour, mainly depends on the viewport size. So for our experiments we will focus on settings that vary this parameter. The runtime of our algorithms also depends on the length of the input polyline and the number of vertices in it. For our map samples the total length can be found in Table 7.1. We used the same simplified polylines as inputs as in Section 7.4 so the number of vertices can be found in Table 7.2. For the lineman we super-sampled to a target segment size of 7.5 px, so in this case the samples have 347 (Würzburg), 702 (Louisville), and 193 (Bray) virtual vertices.

Map	Algorithm	Viewport sized 26 px	Maximum viewport s.t. $t \leq \ldots$ 125 ms	250 ms	500 ms	1 s
Würzburg	lineman classic	2 m 29 s	4 px	5 px	7 px	8 px
	lineman bisector	104 ms	32 px	49 px	67 px	88 px
	imagewalk	365 ms	8 px	19 px	33 px	54 px
	linewalk	764 ms	< 2 px	3 px	16 px	34 px
Louisville	lineman classic	4 m 52 s	3 px	5 px	6 px	7 px
	lineman bisector	115 ms	28 px	41 px	55 px	73 px
	imagewalk	545 ms	7 px	14 px	25 px	40 px
	linewalk	1 400 ms	< 2 px	2 px	5 px	19 px
Bray	lineman classic	1 m 20 s	5 px	6 px	8 px	9 px
	lineman bisector	38.5 ms	50 px	66 px	85 px	110 px
	imagewalk	145 ms	23 px	40 px	66 px	168 px
	linewalk	294 ms	4 px	21 px	41 px	67 px

Tab. 7.6.: Running times for a representative viewport and maximal viewport sizes for chosen response times in an interactive environment.

In our experiments we used two settings. First, we chose a viewport of 26 pixels that is representative for our free hand sketches. Second, we tested how large the viewport may be at most to finish in a given time. Because our algorithms are meant for interactive settings, the response time should be imperceptible or at least not annoying. The results can be found in Table 7.6.

The running times of the classic lineman have been addressed several times. Here the numbers confirm that this algorithm is not a match for interactive applications. For our average setting the running time of over a minute is clearly too long. The bisector heuristic as expected shows massive improvements. Also the two other algorithms finish after a reasonable amount of time.

In the other setting also the scaling is tested. We see that when increasing the time limit for all algorithms, except for lineman with the classic heuristic, significantly larger viewports become feasible. The linewalk implementation as outlined in Chapter 6 has potential for improvements. Especially the baseline running times are rather high, even higher than the classic lineman which seems unnecessary and often implies a suboptimal implementation. With more time budget the linewalk supports larger viewports, so the scaling as expected based on the runtime analysis is better.

Another observation is the total length of the input polyline for lineman is not as influential as for the other two algorithms that work on a pixel level. This makes the lineman with bisector heuristic very lightweight and suitable for low latency applications. On the other hand when the features are short, the linewalk concept is fast enough even for larger viewports and needs less setup.

8. Strategies for Interactive Matching

Reading and understanding maps is a complex task even for humans. For the foreseeable future algorithms will have a supporting role in that process. While we rely on humans to identify a feature of interest, more sophisticated tools may automate this. But a human reviewer will have to check the results and take actions on wrongly attributed elements. This task should be made as simple and as efficient as possible.

Our algorithms are designed for a batch-processing setup. For a given set of maps input polylines have to be present and the algorithms optimize them by moving their vertices. Considering that the alignment is not always perfect, the setting should instead expect the process of generating inputs and reviewing outputs to be interleaved, so feedback loops allow continuous adaption. Human annotators mark features for transcription by clicking rough polylines along the course of the lines. Then an alignment algorithm performs optimization steps, followed by an examination of the results by the annotator. If the alignment is not adequate the input can be modified followed by another round of optimization and examination.

There are two straightforward points where this feedback loop can be enhanced. First, the examination process may be guided by a measure of confidence for the segments of the polyline, so the reviewer can focus on points where there are doubts that the algorithm performed well. Second, in case an annotator found an area of misalignment the algorithm should not just be rerun with a refined input but the annotator might add additional points to the input polyline that lie on the feature and must not be moved by the algorithm.

Incorporating Fixed Points

The algorithms presented in this thesis produce the alignment by moving (potentially super-sampled) vertices according to given parameters. In order to support fixed vertices some changes are necessary. Suppose that, as part of the input, we know which vertex should be fixed as an augmentation of the polyline.

For the algorithms based on hidden Markov models we already introduced the concept for example for the start vertex in Section 3.1. A vertex that should be fixed induces a probability of one for the closest pixel and of zero for all other pixels. The Viterbi table than has only one non-zero entry in the corresponding column and the optimal sequence has to contain exactly this entry. So for the lineman and linewalk algorithms we can use this to ensure a given vertex is not moved in the output—to be precise, it will be moved to pixel closest to the vertex but no further. Both implementations have been

extended to support fixed points as inputs so we can test the processes presented in this chapter.

The imagewalk algorithm is based on uniform-cost search. Here, the concept of fixed points is more difficult to incorporate. Our approach is to split the input at each fixed point and run the alignment independently. We know that with this algorithm the input and output polyline share the start and end vertex. Therefore, we can obtain a consistent output by concatenating the separate polylines. With respect to our assumptions in the definition of the imagewalk algorithm, this is optimal. Despite this is not much more effort than adopting the HMM based algorithms and because for a first evaluation we do not need all the algorithms, due to time constraints the implementation has been postponed.

Measuring Alignment Confidence: The Naïve Approach

In order to provide guidance for finding areas of misalignment, we need to provide a measures of alignment confidence. One naïve approach is to find the lightest segment of the output polyline. Using the average luminosity defined in Section 2.5 this can be implemented with already established techniques. For each segment the line drawing is calculated for example using Bresenham's algorithm as suggested in Section 2.6. Averaging the luminosity of the pixels from the drawing gives a ranking where the darkness of a segment means we are more confident that this segment aligns with the feature. Where dark segments show confidence, light segments probably show cases where there is a misalignment or a change to another feature.

This approach gives us the points on the polyline where the algorithm made a wrong decision or corrected a former mistake by changing back to the feature. Where we seek to identify areas of misalignment, the segments that are light because the algorithm switches to another feature (or back from one) only point at the start (or end) of such an area. As a feedback to the annotator, one could argue the center of the misaligned area is preferable. With the luminosity only, we cannot determine where and if we align with the target feature versus an undesired one. So we cannot find the center of that area.

Measuring Alignment Confidence Via Disagreement

Another approach is to use the fact that our toolkit provides us with more than one algorithm for the task. Assuming that those make different mistakes, we can run two algorithms with the same input and find areas where the results diverge. For finding the largest gap between two polylines the Fréchet distance can be used. As discussed in Section 2.2, the Fréchet distance is difficult to calculate. Therefore we use a simpler formulation, the discrete Fréchet distance, as proposed by Eiter and Mannila [EM94].

Other than the original, the discrete Fréchet distance only looks at the vertices of the polylines and ignores the segments in between. Therefore, for polygonal curves the original Fréchet distance is always smaller or equal the discrete variant. If the leash is long

Algorithm 8: Discrete Fréchet distance

Input: Polylines $s_1, \ldots, s_n, t_1, \ldots, t_m$
Output: Indices i, j of the two gap vertices

```
C ← table n × m
for 1 ≤ i ≤ n do
    for 1 ≤ j ≤ m do
        d ← ‖s_i, t_j‖
        if i = 1 and j = 1 then C[i, j] ← d
        else if i = 1 then C[i, j] ← max(C[i, j − 1], d)
        else if j = 1 then C[i, j] ← max(C[i − 1, j], d)
        else C[i, j] ← max(min(C[i − 1, j − 1], C[i, j − 1], C[i − 1, j]), d)
return arg min_{1≤i≤n, 1≤j≤m} i + j such that C[i, j] = C[n, m]
```

enough to span the distance between the endpoints it is also long enough to span any minimal distance in between. Eiter and Mannila also show that the difference is not larger than the length of longest segment of the polyline. For our use case the polylines have rather small segments either at the length they were super-sampled at or at one pixel length. Because of that, we do not expect consequences on the geometrical precision of our method by using the discrete Fréchet distance instead of the exact Fréchet distance.

Algorithm 8 has a detailed description of discrete Fréchet distance. Notice that compared to the formulation by Eiter and Mannila we also backtrack the dynamic programming table to find the location of the gap responsible for the distance. That is the position in the table where the first time the final distance appears. Apart from that the algorithm is a straightforward application of dynamic programming. As a result we get the indices of two vertices. The midpoint of the segment between them is our approximation for the center of the gap.

Evaluation

In order to quantify the effect of interactive alignment we set up a small experiment that is a simplified fully autonomous version of the interactive process. We use a degraded input like in Chapter 7 using the Doublas-Peucker algorithm with ε equals 34 to simplify a hand-annotated reference and apply Gaussian noise as described in Section 7.2 with the zero vector as mean and a standard deviation of (6 px, 6 px). This input then is aligned to the map by applying the Lineman algorithm with the bisector heuristic as well as the linewalk algorithm—those we prepared for incorporating fixed points.

Then, in order to find the worst segment we started using the naïve approach separately for each algorithm. In another setting we used the combined strategy based on

		Separate		Combined	
		Lineman	Linewalk	Lineman	Linewalk
Würzburg	Start	81.89 %	69.98 %		
	Steps	3	4	10	
	End	82.63 %	74.38 %	85.10 %	83.42 %
Louisville	Start	49.27 %	59.65 %		
	Steps	5	8	10	
	End	54.91 %	69.70 %	55.53 %	63.69 %
Bray	Start	40.01 %	64.58 %		
	Steps	4	6	13	
	End	48.54 %	78.72 %	84.34 %	88.11 %

Tab. 8.1.: F_1-scores before and after several steps of automated refinements.

the discrete Fréchet distance. With the coordinate of the center of lowest confidence we take the closest segment and the closest point in the reference. This segment then is patched with the reference point to produce a refined input.

The refined input is again aligned and we find new areas of low confidence. The above procedure is repeated and a more and more patched input is produced of that we expect better alignments. The alignment quality is determined for every step and algorithm as described in Section 7.3. We repeat the procedure until either the alignment is perfect (meaning an F_1-score of 99 % or more in our case) or the same reference point is selected twice which would result in an infinite loop. The results are presented in Table 8.1.

For each map the table has the two settings (*separate* and *combined*) and the baseline F_1-score before the first patching step (which is the same for both settings), the number of steps before we stopped, and the resulting score after the last patch. We can see none of the experiments stopped with a perfect alignment but the quality could be increased.

Recall that the second condition under that the procedure stops means the patch did not improve the alignment and the same area is still likely poorly aligned. So having more successful steps is positive and usually results in a better alignment. As we can see the combined method leads to better results than only evaluating the segment luminosity. When reviewing the results, the recommendations for badly aligned areas to us appeared useful. Other than our autonomous setup, a human would have strategically chosen better patches from the reference. So this has to be considered as a weakness of the experiment that might have had a negative influence on the results. In practice with a real human selecting the fixed points, we expect an overall better alignment after the refinement steps.

9. Conclusion

Aligning polylines to line features on raster data of historical maps can be done with good geometric accuracy using the tools we presented. On real scans of historic maps from different centuries we showed that our algorithms are ready for application. Based on prior work we implemented an algorithm that uses hidden Markov models (HMMs) and local optimization techniques. Our evaluation shows that this is a match for the task and displacements of 3–4 times the feature width can be reliably compensated. With super-sampling also inputs with low complexity can be processed. For this algorithm we found good heuristics that reduce the running time by three orders of magnitude for real-world instances compared to prior implementations.

Because the existing algorithms, including our first approach, depend on multiple critical parameters we introduced enhanced algorithms that provide fast and accurate alignments while requiring less domain knowledge and tuning. The first algorithm uses uniform-cost search (UCS) to find good paths through pixels alongside the input maximizing the achievable geometric precision to one pixel without depending on additional user input for super-sampling.

Combining those approaches, we designed a third algorithm that uses HMMs for the large scale and UCS for the fine adjustments. This realizes a feasible tool with only one adjustable parameter—the search range or maximum displacement—that can be ergonomically integrated in an annotator interface for example by highlighting the covered area. In future versions the algorithm can be implemented with variable viewports to enable users to choose the maximum displacement per line segment. Our evaluation shows that alignment quality and performance is comparable to the other approaches as long as the line feature is continuous. An enhancement to enable aligning discontinuous features like dashed lines or similar patterns is one important task for future research. We also presented an idea how in future work the running time can be improved using search strategies that cover multiple targets. Parallelization is a promising approach to further reduce running times that, as shown in the literature review, has been successfully implemented for related tasks.

Besides the running time, the alignment quality is the most important property of these algorithms. We implemented a benchmarking harness based on established guidelines to assure and compare the quality of different implementations. Our experiments show how the different parameters affect the alignment depending on different properties of the maps and targeted features. Also we could evaluate how critical the precision of the input is and for what grade of line displacement an accurate reconstruction of the line feature can be expected.

Our algorithms use the luminosity as a measure for whether a given pixel belongs to a feature on the map or not. For colored maps a multispectral approach seems more promising. When the color of the target feature is known, the algorithms can use a distance measure in the color space instead. More research into this approach is necessary. Especially the automatic deduction of the feature color from imprecise input is an open question that overlaps the greater question of automated feature detection. Also other visual properties like saturation could be taken into account to enhance the classification step. Even more enhanced techniques based on machine learning when available can be installed that use our algorithms to create the vector representation from pixel classification.

To get an intuition for real-world usage we simulated an interactive process for producing annotations with local optimization by our algorithms. The results show that measures of confidence can guide users to areas of misalignment and reduce the effort necessary for obtaining data of highest quality. These findings can be used as best practices for future designs of interactive feature annotation software.

Bibliography

[AERW03] Helmut Alt, Alon Efrat, Günter Rote, and Carola Wenk: Matching planar maps. *Journal of Algorithms*, 49(2):262–283, November 2003, 10.1016/S0196-6774(03)00085-3.

[AG95] Helmut Alt and Michael Godau: Computing the Fréchet distance between two polygonal curves. *International Journal of Computational Geometry & Applications*, 05(01n02):75–91, 1995, 10.1142/S0218195995000064.

[BPSW05] Sotiris Brakatsoulas, Dieter Pfoser, Randall Salas, and Carola Wenk: On map-matching vehicle tracking data. In *31st International Conference on Very Large Data Bases*, VLDB '05, pages 853–864. VLDB Endowment, August 2005.

[Bre65] Jack E. Bresenham: Algorithm for computer control of a digital plotter. *IBM Systems Journal*, 4(1):25–30, 1965, 10.1147/sj.41.0025.

[BvDFA16] Benedikt Budig, Thomas C. van Dijk, Fabian Feitsch, and Mauricio Giraldo Arteaga: Polygon consensus: Smart crowdsourcing for extracting building footprints from historical maps. In *24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, SIGSPATIAL '16. ACM, October 2016, 10.1145/2996913.2996951.

[car00] To Naomi Mitchison 25 April 1954. In Humphrey Carpenter and Christopher Tolkien (editors): *The Letters of J. R. R. Tolkien*, chapter 144, pages 193–197. Houghton Mifflin Co, 2000.

[CCR+07] Mo Chen, Rezaul Chowdhury, Vijaya Ramachandran, David Roche, and Lingling Tong: Priority queues and Dijkstra's algorithm. UTCS technical report TR-07-54, University of Texas at Austin, October 2007. `https://www.cs.utexas.edu/ftp/techreports/tr07-54.pdf`.

[Chi92] Nancy Chinchor: Muc-4 evaluation metrics. In *4th Conference on Message Understanding*, MUC4 '92, pages 22–29. Association for Computational Linguistics, June 1992, 10.3115/1072064.1072067.

[CSV14] Bin Chen, Weihua Sun, and Anthony Vodacek: Improving image-based characterization of road junctions, widths, and connectivity by leveraging OpenStreetMap vector map. In *2014 IEEE Geoscience and Remote Sensing Symposium*, pages 4958–4961, July 2014, 10.1109/IGARSS.2014.6947608.

[DC18] Weiwei Duan and Yao Yi Chiang: SRC: A fully automatic geographic feature recognition system. *SIGSPATIAL Special*, 9(3):6–7, January 2018, 10.1145/3178392.3178396.

[DCL+19] Weiwei Duan, Yao Yi Chiang, Stefan Leyk, Johannes H. Uhl, and Craig A. Knoblock: Automatic alignment of contemporary vector data and georeferenced historical maps using reinforcement learning. *International Journal of Geographical Information Science*, 34(4):824–849, December 2019, 10.1080/13658816.2019.1698742.

[Dij59] Edsger W Dijkstra: A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, December 1959, 10.1007/BF01386390.

[DP73] David H Douglas and Thomas K Peucker: Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographica: The International Journal for Geographic Information and Geovisualization*, 10(2):112–122, October 1973, 10.3138/FM57-6770-U75U-7727.

[EM94] Thomas Eiter and Heikki Mannila: Computing discrete Fréchet distance. Technical Report CD-TR 94/64, TU Vienna, April 1994. `http://www.kr.tuwien.ac.at/staff/eiter/et-archive/cdtr9464.pdf`.

[Fel11] Ariel Felner: Position paper: Dijkstra's algorithm versus uniform cost search or a case against Dijkstra's algorithm. In *Fourth Annual Symposium on Combinatorial Search*. AAAI Press, July 2011.

[Fré06] Maurice Fréchet: Sur quelques points du calcul fonctionnel. *Rendiconti del Circolo Matematico di Palermo (1884–1940)*, 22(1):1–72, 1906.

[GT96] Andrew V. Goldberg and Robert E. Tarjan: Expected performance of Dijkstra's shortest path algorithm. Technical report, NEC Research Institute, June 1996. `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.54.4349`.

[HB12] Jan Henrik Haunert and Benedikt Budig: An algorithm for map matching given incomplete road data. In *20th International Conference on Advances in Geographic Information Systems*, SIGSPATIAL '12, pages 510–513. ACM, November 2012, 10.1145/2424321.2424402.

[HC19] Mohammad D. Hossain and Dongmei Chen: Segmentation for object-based image analysis (OBIA): A review of algorithms and challenges from remote sensing perspective. *ISPRS Journal of Photogrammetry and Remote Sensing*, 150:115–134, April 2019, 10.1016/j.isprsjprs.2019.02.009.

[HMWJ97] Christian Heipke, Helmut Mayer, Christian. Wiedemann, and Olivier Jamet: Evaluation of automatic road extraction. In *3D Reconstruction and Modelling of Topographic Objects Joint ISPRS Commission III/IV Workshop*, volume 32, part 3–4W2 of *ISPRS Archives*, pages 151–160, September 1997, 10.14463/GBV:1067824685.

[HRF+07] Jiuxiang Hu, Anshuman Razdan, John C. Femiani, Ming Cui, and Peter Wonka: Road network extraction and intersection detection from aerial images by tracking road footprints. *IEEE Transactions on Geoscience and Remote Sensing*, 45(12):4144–4157, November 2007, 10.1109/TGRS.2007.906107.

[Hum06] Britta Hummel: *Map matching for vehicle guidance*, chapter 10, pages 211–222. CRC Press, first edition, November 2006, 10.1201/9781420008609-20.

[KHL07] John Krumm, Eric Horvitz, and Julie Letchner: Map matching with travel time constraints. Technical paper, SAE International, April 2007, 10.4271/2007-01-1102.

[KWDG15] Hannes Koller, Peter Widhalm, Melitta Dragaschnig, and Anita Graser: Fast hidden markov model map-matching for sparse and noisy trajectories. In *18th IEEE International Conference on Intelligent Transportation Systems*, pages 2557–2561, September 2015, 10.1109/ITSC.2015.411.

[NK09] Paul Newson and John Krumm: Hidden markov map matching through noise and sparseness. In *17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, GIS '09, pages 336–343. ACM, November 2009, 10.1145/1653771.1653818.

[NW70] Saul B. Needleman and Christian D. Wunsch: A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, March 1970, 10.1016/0022-2836(70)90057-4.

[Ram72] Urs Ramer: An iterative procedure for the polygonal approximation of plane curves. *Computer Graphics and Image Processing*, 1(3):244–256, November 1972, 10.1016/S0146-664X(72)80017-0.

[RJ86] Lawrence R. Rabiner and Biing Hwang Juang: An introduction to hidden markov models. *IEEE ASSP Magazine*, 3(1):4–16, January 1986, 10.1109/MASSP.1986.1165342.

[RN09] Stuart Russell and Peter Norvig: *Artificial Intelligence A Modern Approach*, chapter 3.4 Uninformed Search Strategies, pages 81–91. Prentice Hall Series in Artificial Intelligence. Third edition, 2009.

[Sla13] G. William Slade: The Viterbi algorithm demystified. Available online, March 2013. https://www.researchgate.net/publication/235958269.

[SLS+12] Renchu Song, Wei Lu, Weiwei Sun, Yan Huang, and Chunan Chen: Quick map matching using multi-core CPUs. In *20th International Conference on Advances in Geographic Information Systems*, SIGSPATIAL '12, pages 605–608. ACM, November 2012, 10.1145/2424321.2424428.

[SS19] Mahmoud Saeedimoghaddam and Tomasz F. Stepinski: Automatic extraction of road intersection points from USGS historical map series using deep convolutional neural networks. *International Journal of Geographical Information Science*, 34(5):947–968, November 2019, 10.1080/13658816.2019.1696968.

[usg65] Louisville quadringle, Colorado. Map, Scale 1:24 000, U.S. Geological Survey, 1965. https://store.usgs.gov/product/273896.

[usg01] Bray quadringle, California-Siskiyou county. Map, Scale 1:24 000, U.S. Geological Survey, 2001, ISBN 978-0-607-89126-3. https://store.usgs.gov/product/46688.

[vDCD20] Thomas C. van Dijk, Yao Yi Chiang, and Weiwei Duan: A tool for aligning geojson linestrings to bitmap images. Available online, August 2020. https://github.com/tcvdijk/lineman.

[vDFH20] Thomas C. van Dijk, Norbert Fischer, and Bernhard Häussner: Algorithmic improvement of crowdsourced data: Intrinsic quality measures, local optima, and consensus. In *28th International Conference on Advances in Geographic Information Systems*, SIGSPATIAL '20, pages 433–436. ACM, November 2020, 10.1145/3397536.3422260.

[Vit67] Andrew J. Viterbi: Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, 13(2):260–269, April 1967, 10.1109/TIT.1967.1054010.

[Wen85] Ludwig Wenng: Karte des Landgerichtes Würzburg. Map, Scale 1:200 000, 1885. https://www.digitale-sammlungen.de/en/view/bsb00013499.

[WF74] Robert A. Wagner and Michael J. Fischer: The string-to-string correction problem. *J. ACM*, 21(1):168–173, January 1974, 10.1145/321796.321811.

[Zin16] Alois Zingl: A rasterizing algorithm for drawing curves. Technical report, Technikum-Wien, 2016. https://zingl.github.io/Bresenham.pdf.

A. Supplemental Figures

A.1 Alignment quality of lineman with the classic heuristic on the Louisville map 74
A.2 Alignment quality of lineman with the bisector heuristic on the Louisville map 74
A.3 Alignment quality of lineman with the classic heuristic on the Bray map 75
A.4 Alignment quality of lineman with the bisector heuristic on the Bray map 75
A.5 Alignment quality of the imagewalk algorithm on the Louisville map . . 76
A.6 Alignment quality of the imagewalk algorithm on the Bray map 76
A.7 Alignment quality of the linewalk algorithm on the Louisville map . . . 77
A.8 Alignment quality of the linewalk algorithm on the Bray map 77

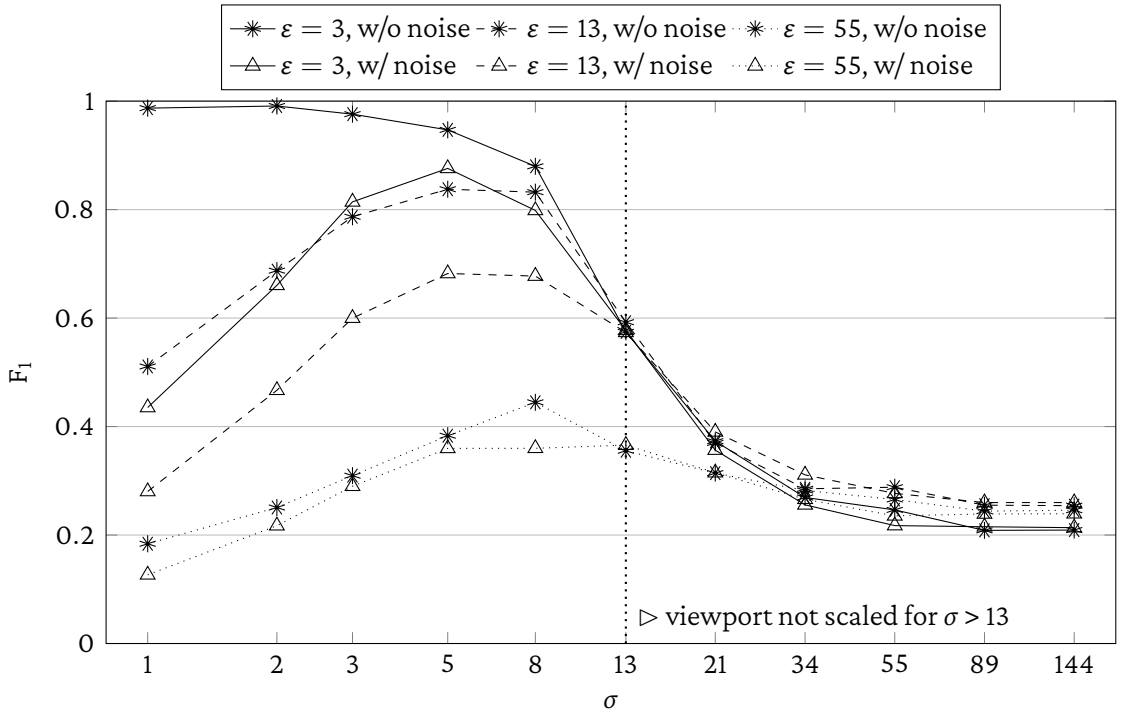


Fig. A.1.: Alignment quality of lineman with the classic heuristic on the Louisville map. See Section 7.5 for details.

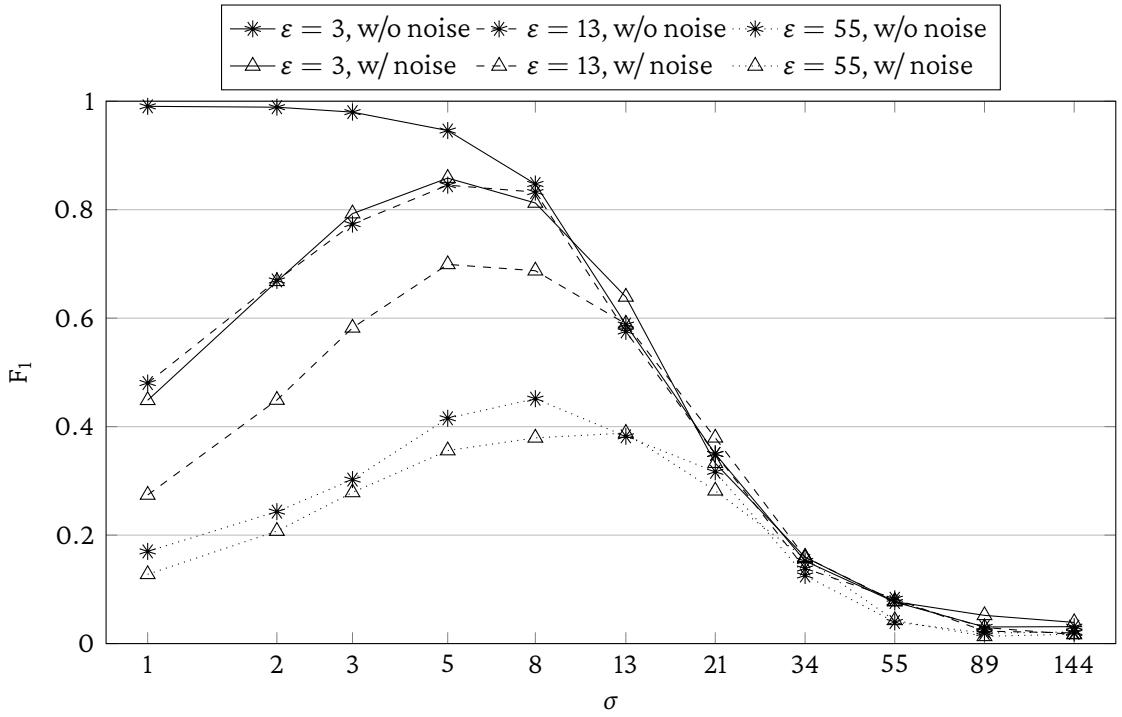


Fig. A.2.: Alignment quality of lineman with the bisector heuristic on the Louisville map. See Section 7.5 for details.

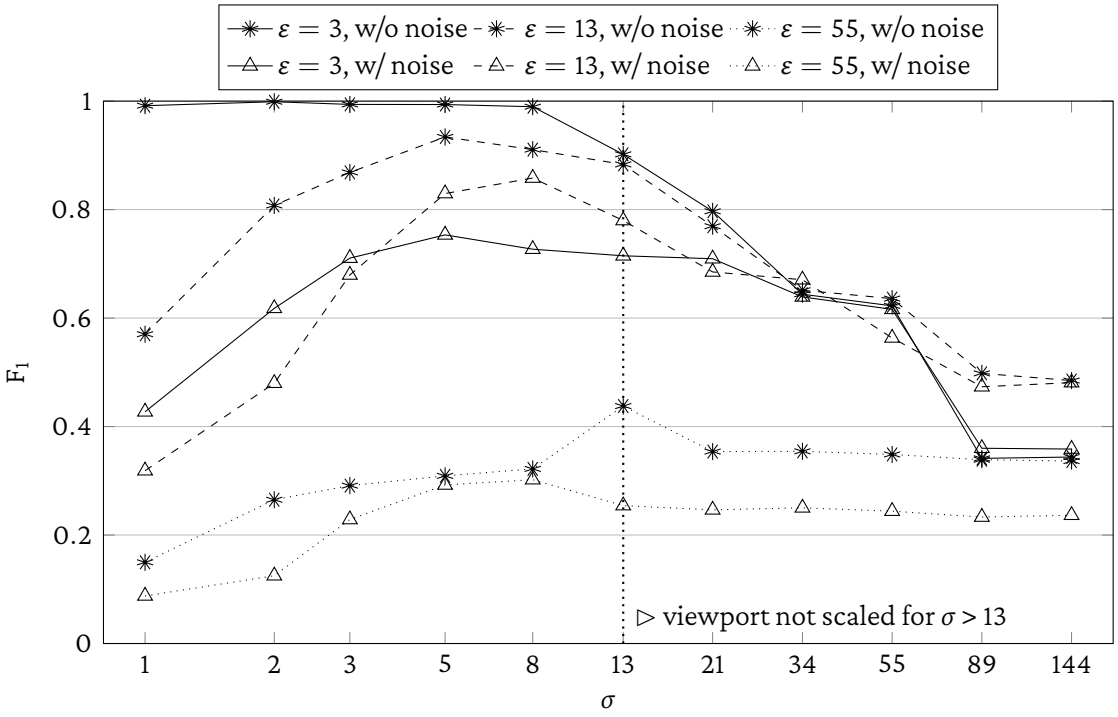


Fig. A.3.: Alignment quality of lineman with the classic heuristic on the Bray map. See Section 7.5 for details.

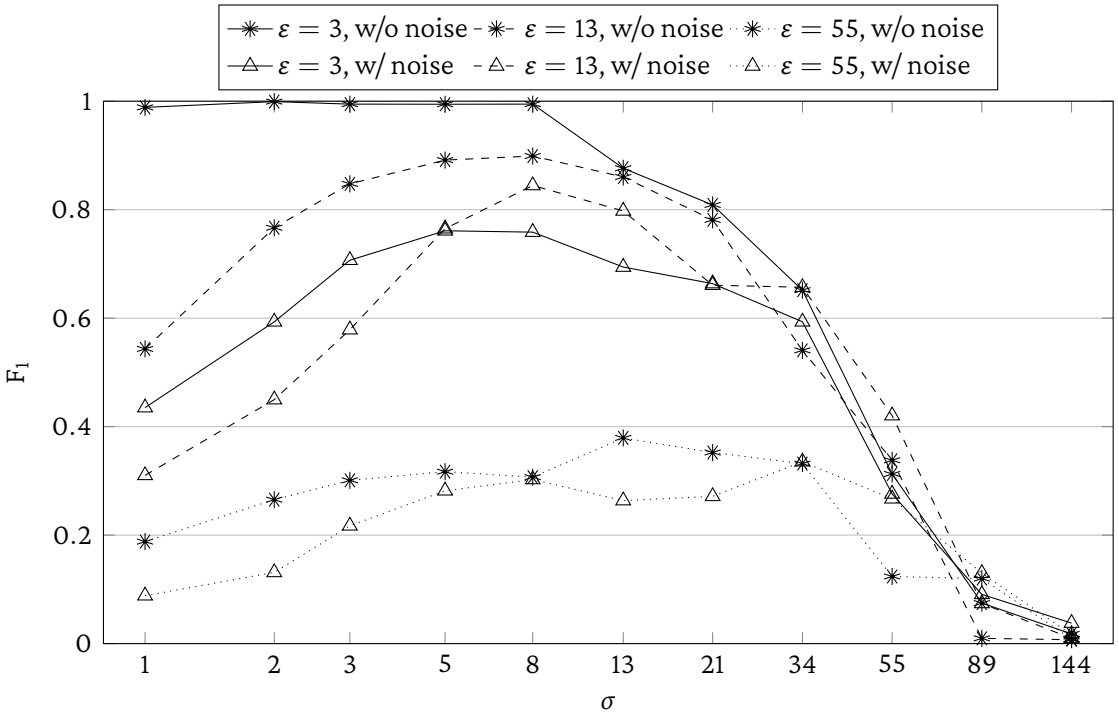


Fig. A.4.: Alignment quality of lineman with the bisector heuristic on the Bray map. See Section 7.5 for details.

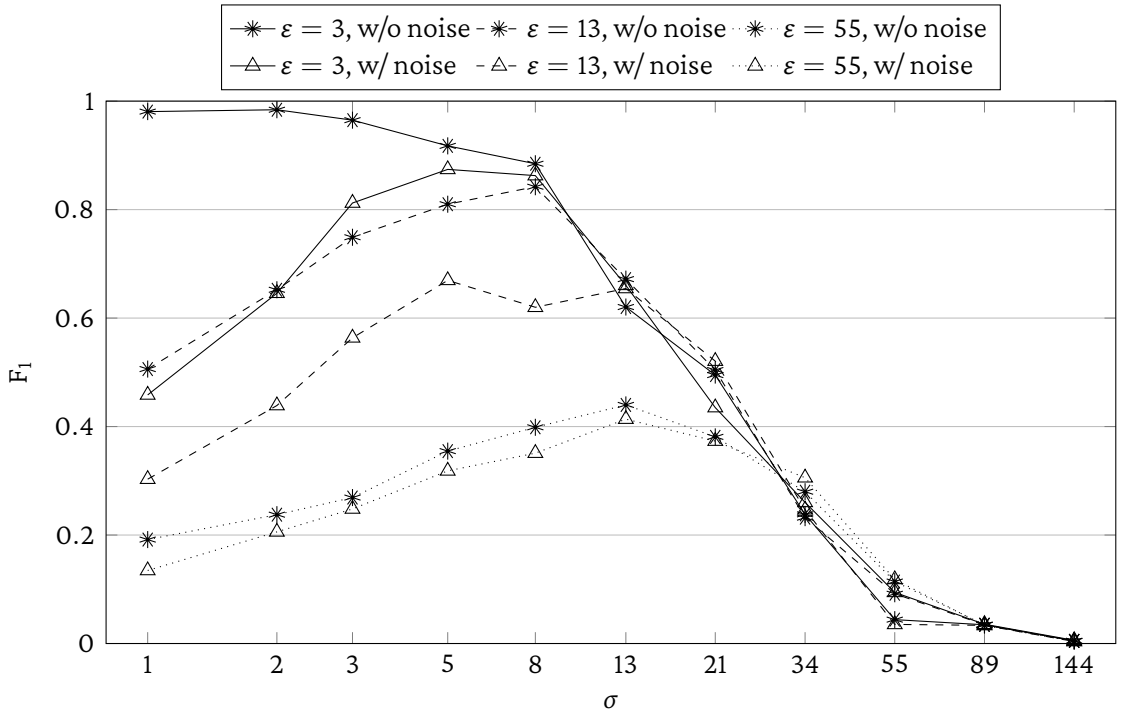


Fig. A.5.: Alignment quality of the imagewalk algorithm on the Louisville map. See Section 7.6 for details.

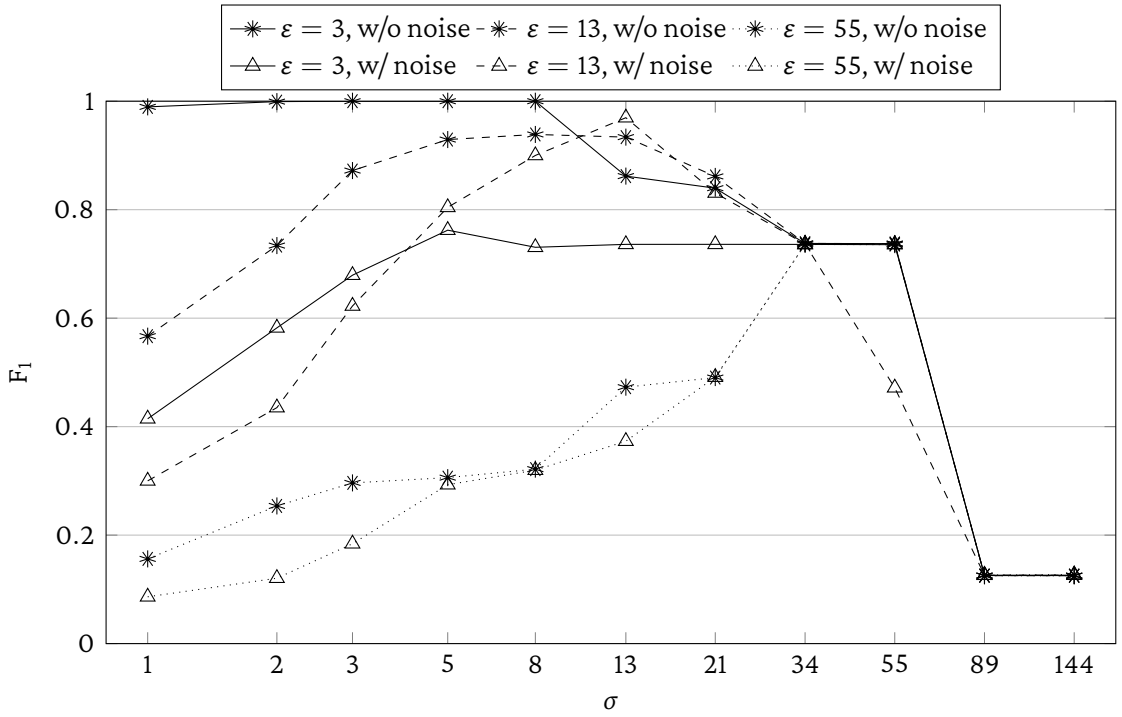


Fig. A.6.: Alignment quality of the imagewalk algorithm on the Bray map. See Section 7.6 for details.

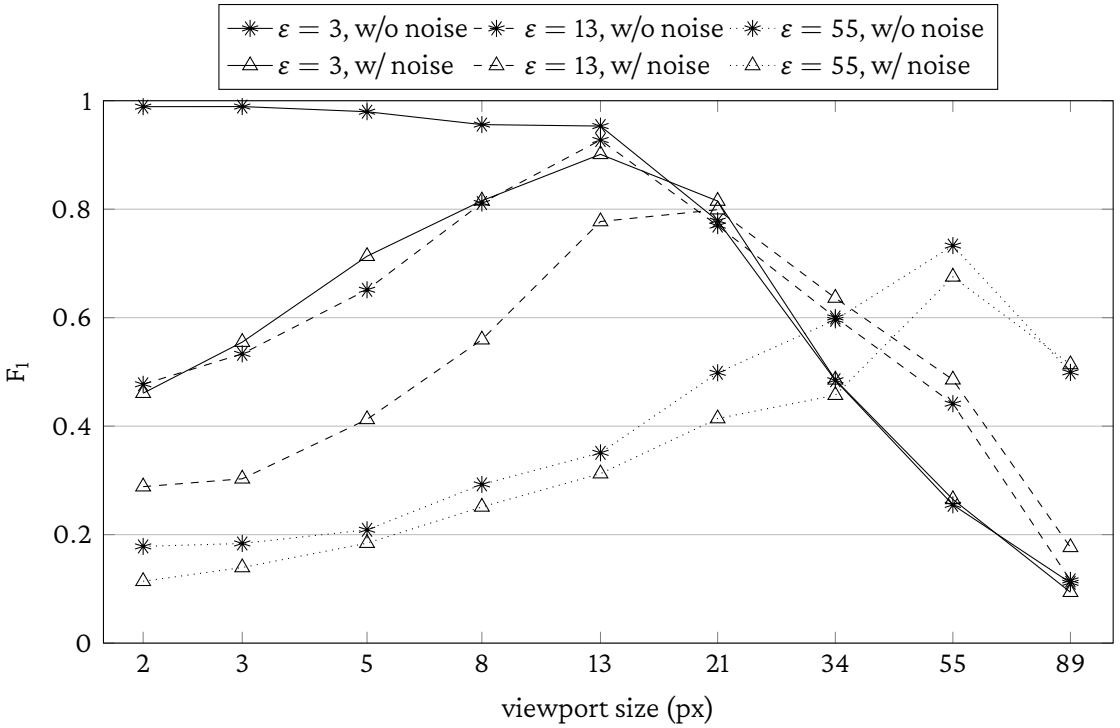


Fig. A.7.: Alignment quality of the linewalk algorithm on the Louisville map. See Section 7.7 for details.

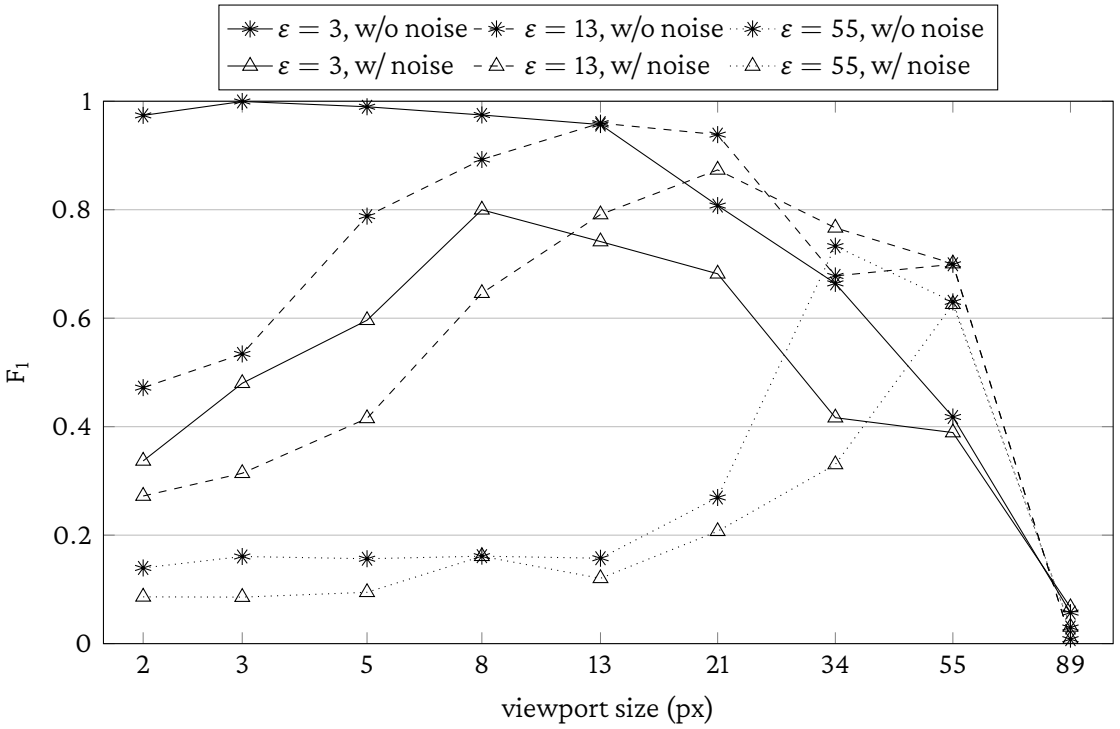


Fig. A.8.: Alignment quality of the linewalk algorithm on the Bray map. See Section 7.7 for details.

Erklärung

Hiermit versichere ich die vorliegende Abschlussarbeit selbstständig verfasst zu haben, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben, und die Arbeit bisher oder gleichzeitig keiner anderen Prüfungsbehörde unter Erlangung eines akademischen Grades vorgelegt zu haben.

Würzburg, den 16. November 2021

. . . . . .
Tim Hegemann