Universität Stuttgart

Improved Algorithms for Map Rendering and Route Planning

Von der Fakultät für Informatik, Elektrotechnik und Informationstechnik der Universität Stuttgart zur Erlangung der Würde eines Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von

Thomas Mendel

aus Stuttgart

Hauptberichter: Prof. Dr. Stefan Funke

Mitberichter: Prof. Dr. Sabine Storandt

Tag der mündlichen Prüfung: 04.06.2020

Institut für Formale Methoden der Informatik

2020

Contents

Zusammenfassung

In dieser Arbeit werden verschiedene Teilbereiche der Kartendarstellung und der Wegefindung betrachtet.

Der erste Abschnitt befasst sich mit der Darstellung von Karten und besteht aus zwei Teilen. Im ersten Teil wird ein Ansatz vorgestellt, um detaillierte Karten vereinfacht darzustellen. Hierbei wird aus einer gegebenen polygonalen Flächenunterteilung eine Unterteilung mit weniger Knoten erzeugt. Es wird ein iteratives Verfahren auf Basis von Delaunay-Triangulierungen vorgestellt. Die neue Unterteilung ist überschneidungsfrei und berücksichtigt dazu Topologie- und Flächen-Einschränkungen. Außerdem wird eine optimale Lösung durch ein ILP berechnet um die Qualität der Heuristik mit dem Optimum zu vergleichen. Der zweite Teil beschäftigt sich mit Kartenbeschriftungen. Hierbei ist das Ziel einen Schriftzug gut leserlich in einem Polygon zu platzieren. Zuerst wird die Form des Polygons durch sein Skelett approximiert. Dann werden verschiedene Kreisbögen entlang des Skeletts ausgewählt und der Schriftzug auf den Kreisbögen platziert. Letztendlich wird die beste so gefundene Beschriftung ausgewählt.

Der zweite Abschnitt beschäftigt sich mit der Routenplanung und stellt eine Beschleunigungstechnik für Contraction-Hierarchies vor. Bei einer Contraction-Hierarchies-Anfrage werden unnötige Berechnungen durchgeführt. In einer Vorberechnung werden Informationen erzeugt, damit die unnötigen Berechnungen erkannt werden und nicht ausgeführt werden müssen. Dies beschleunigt die Anfragen entsprechend.

Summary

This work considers different aspects of map-rendering and routing.

The first part deals with the presentation of maps and consists of two chapters. The first chapter presents a technique to simplify detailed map data. Given a polygonal subdivision, we compute a subdivision with fewer vertices. We propose an incremental algorithm, which is based on Delaunay-triangulations. The new subdivision is intersection-free and respects topological as well as area constraints. Furthermore, optimal solutions are computed via an ILP to compare the heuristic's solutions to the optimum. The second chapter deals with map labeling. Here a given text should be placed inside a polygon legibly. First we approximate the shape of the polygon via its skeleton. Then different arcs along the skeleton are computed and the text is placed along those arcs. Finally, the best such placement is selected.

The second part presents an acceleration technique for contraction hierarchies. A contraction hierarchy query performs some unnecessary computations. In a preprocessing step, we compute auxiliary information to detect those superfluous computations and skip them. This accelerates the queries accordingly.

Introduction

Maps have evolved tremendously throughout history. The first maps were simple representations of landscape features carved into animal bones. The Babylonians already mapped their cities on clay tablets. Soon there also were simple world maps, most often depicting the earth as a flat surface.

Today's maps are much more refined and augmented with features. We now even have very detailed information about single buildings, which can be used to render 3d-models of entire cities. See Figure 1 for some example pictures.

We still encounter maps as a vital tool and they appear in a great variety of contexts and services.

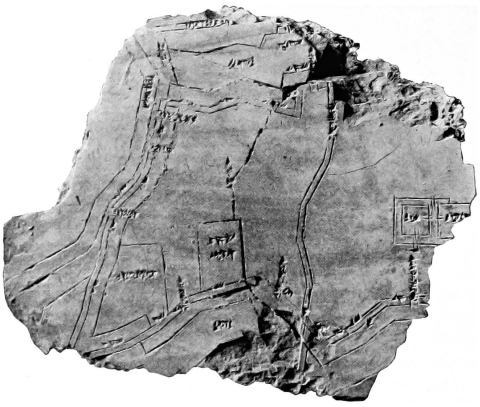
See for example the following use-cases:

- Find your way through an unknown city: When travelling through an unknown city, we can use map-services to find our current position. Furthermore these services can provide a nice overview of close sightseeing targets or other locations of interest (such as restaurants or shopping malls).

- Driving to a location: When driving to an unknown location, we can use online map services like Google Maps to find a way to our target. Even if we know a way to the location one might consider to have a look at Google Maps to get a better understanding of the current traffic situation along the route and maybe take a detour to get to the target faster.

Many services include maps to help users fulfill their tasks. Online Portals for booking hotels can show an overview of all hotels with available rooms on a map. Or delivery services can give you an overview of close pick-up an delivery stations. See Figure 2 for examples of those use-cases.

Of course the classic use-case of maps is navigation. Therefore most map-services also offer routing as part of their portfolio. In the case of large online services, fast and flexible algorithms are a necessity. A standard implementation of Dijkstra's algorithm can run up to several seconds per query. But an online service might need to handle tens of thousands of requests per second. So they use faster routing-schemes to answer all queries in reasonable time.

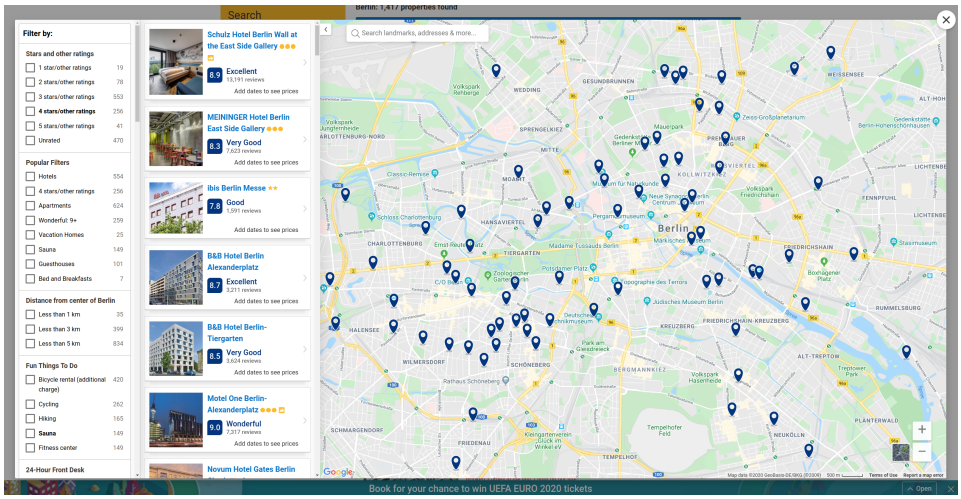


(a) City outlines on a clay tablet.

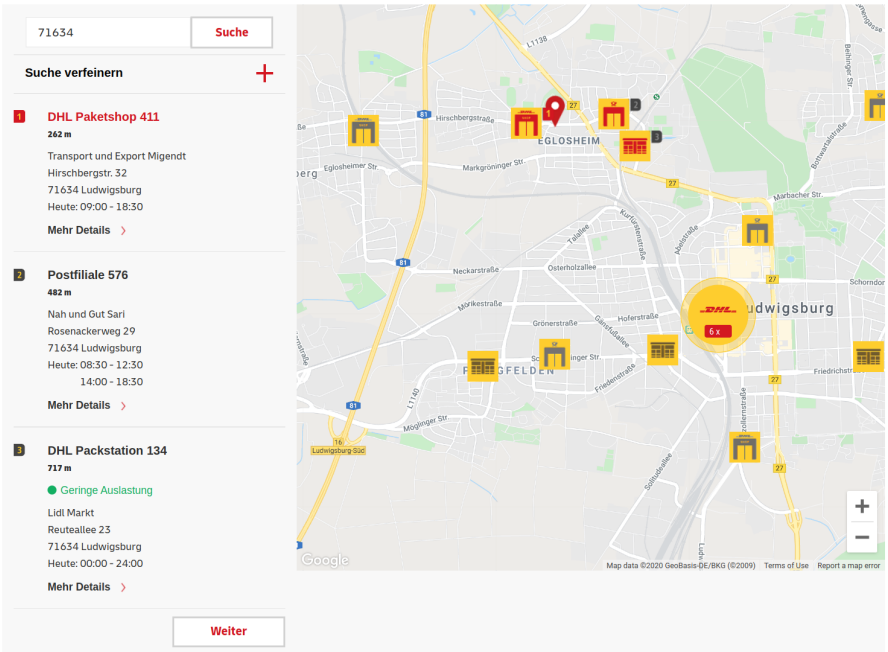
(b) A drawn map with detailed coastlines.

(c) A detailed map of Manhatten with 3d-models of the buildings.

Figure 1.: The evolution of maps.



(a) Overview of available hotel rooms on booking.com



(b) Overview of close posting and pick-up stations on dhl.de

Figure 2.: Use cases of maps in different services.

Google Maps offers a comprehensive suite of functionality in a single service. While the service is free of charge it's not really "free" in the sense that the dataset and algorithms used are private. Thanks to the open-source community there is a large amount of freely available map data. The most well known project is "Open Street Maps" (OSM - [52]). See Figure 3 for a comparison between Google Maps and OSM.

To be able to offer a free replacement for services like Google Maps we also need free implementations of the algorithms, favorably in the form of an open-source implementation of the complete service.

This work wants to contribute to this movement by proposing algorithms for some of the problems solved in map services, namely in map rendering and route-planning. All algorithms offer an implementation which is tested on freely available data-sets.

The first part of this work deals with the representation of maps.

First a scheme to reduce superfluous complexity is presented. It handles the simplification of polygonal subdivisions. Those occure in the context of maps as borders and outlines of entities such as countries, cities, and areas like forests. Preliminary versions of these works were presented and published on the *European Workshop on Computational Geometry* 2016 (Lugano, [24]), the *Meeting on Algorithm Engineering and Experiments* 2017 (Barcelona, [25]), *Meeting on Algorithm Engineering and Experiments* 2018 (New Orleans, [42]).

Then detailed explanation of an improved labeling scheme is given. Here the goal is to place text-labels in polygonal boundaries. Think about putting country names inside the country's borders on a map. A preliminary version of the work is published on Arxiv.org ([38]).

In the second part an acceleration technique for navigating through street-networks is presented. The scheme combines existing techniques to reduce redundancy in queries and therefore accellerates the computation. This work was presented and published in the *Special Event on Analysis of Experimental Algorithms* 2019 (Kalamata, [23]).

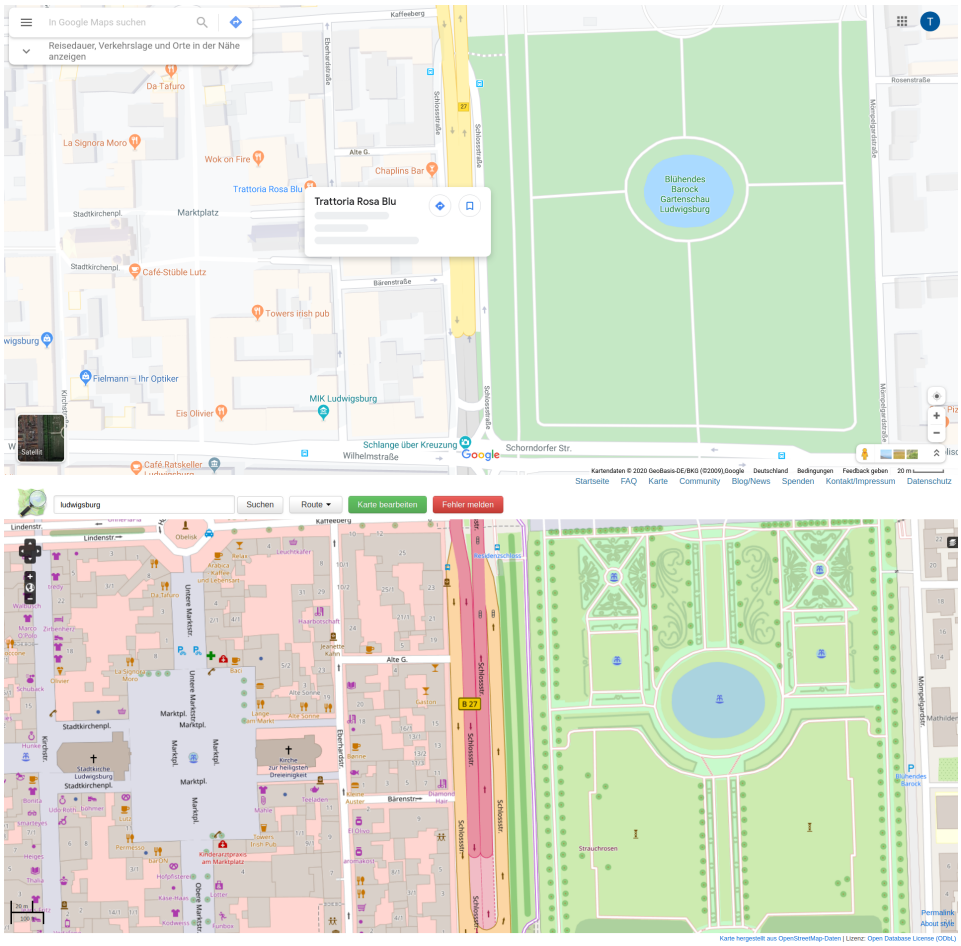


Figure 3.: GoogleMaps (top) and OSM (bottom) next to one another. Even tough the OSM data is available free of charge and collected by volunteers, it is highly detailed. The level of detail can even surpass Google Maps in this example.

Part I.

Map Rendering

General Problem

Maps can contain a very diverse set of features. Features can be represented as areas (e.g. countries, cities), as lines (e.g. streets, rivers) or as points (e.g. landmarks).

When working with maps, one of the main challenges is the presentation of the available information. The large amount of data available even from publicly accessible data must be tamed to give an understandable overview.

The presentation of a map needs to consider the context in which the data is used. We need to consider different aspects of the data for different tasks. When one is hiking, the focus of a map should be the surrounding paths and the geographic shape of the area. Think of contour lines. For longer hikes things like shelters and bbq-stations are also of interest. If one is cycling, only suitable paths should be shown. And even those paths should be differentiated, to show e.g. the difference between roads where cars are allowed and where they are not. During a road trip, the road-network is the most important piece of information. For longer drives the locations of gas stations and restaurants along the way are also relevant.

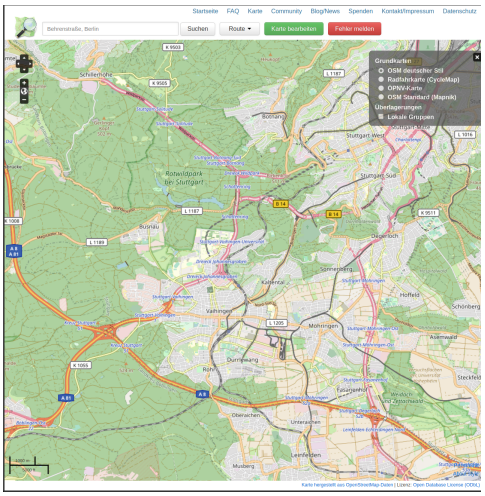
Then, there are additional levels of information which can augment a map. For example in the case of using a (digital) map while driving. Dynamic information, like the current or predicted traffic situation, can be of great help.

So one of the main tasks in map-rendering is filtering the data, such that only information fitting the current use-case is shown. And even the relevant information might be filtered to show only the most important parts.

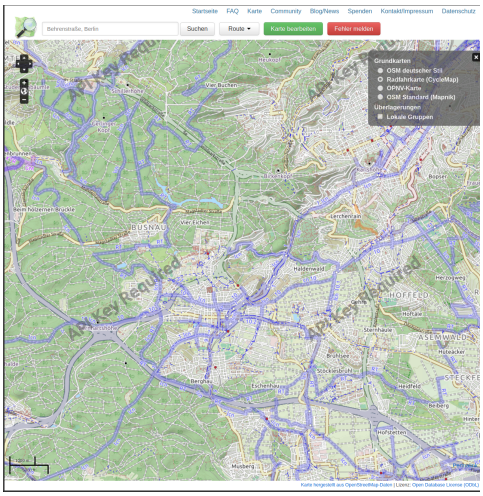
Aside from filtering out relevant features, also the representation of the very data must adapt to the situation. To find a specific building in a city a highly detailed representation helps to orient oneself exactly. If the map is to be used while driving (e.g. in a navigation system), there should only be very little information displayed.

Today even freely available data, like in the Open-Street-Map project, offers huge amounts of precise data. The data consists of roads, coastlines, rivers, country- and city-borders, local shops and much more. From a geometric perspective there is so much data available that it needs to be simplified. There is no point in using tens of thousands of points to define a polygon which is displayed on a low resolution display in a small area. The user can't even perceive all of the data. In a mobile context it can be favorable to receive a coarse representation of the data to save bandwidth.

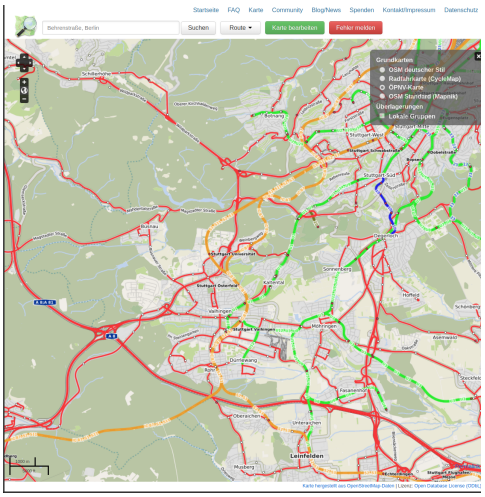
In addition to the colorful geometric representation of data on a map, there are also labels assigned to different objects. They differentiate and give mean-



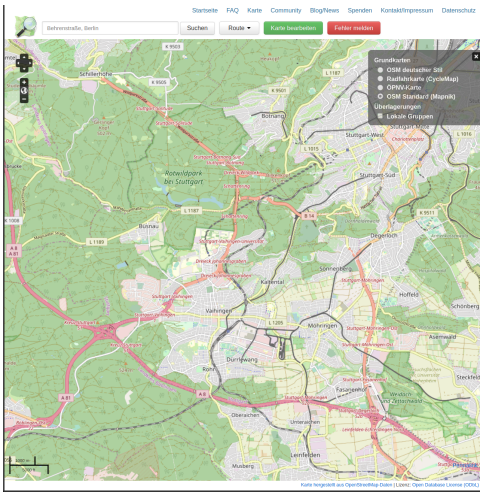
(a) Adapted for german streets.



(b) Adapted for cycling.



(c) Adapted for public transport.



(d) The default rendering style.

Figure 0.4.: Different rendering styles of the OSM dataset for different applications.

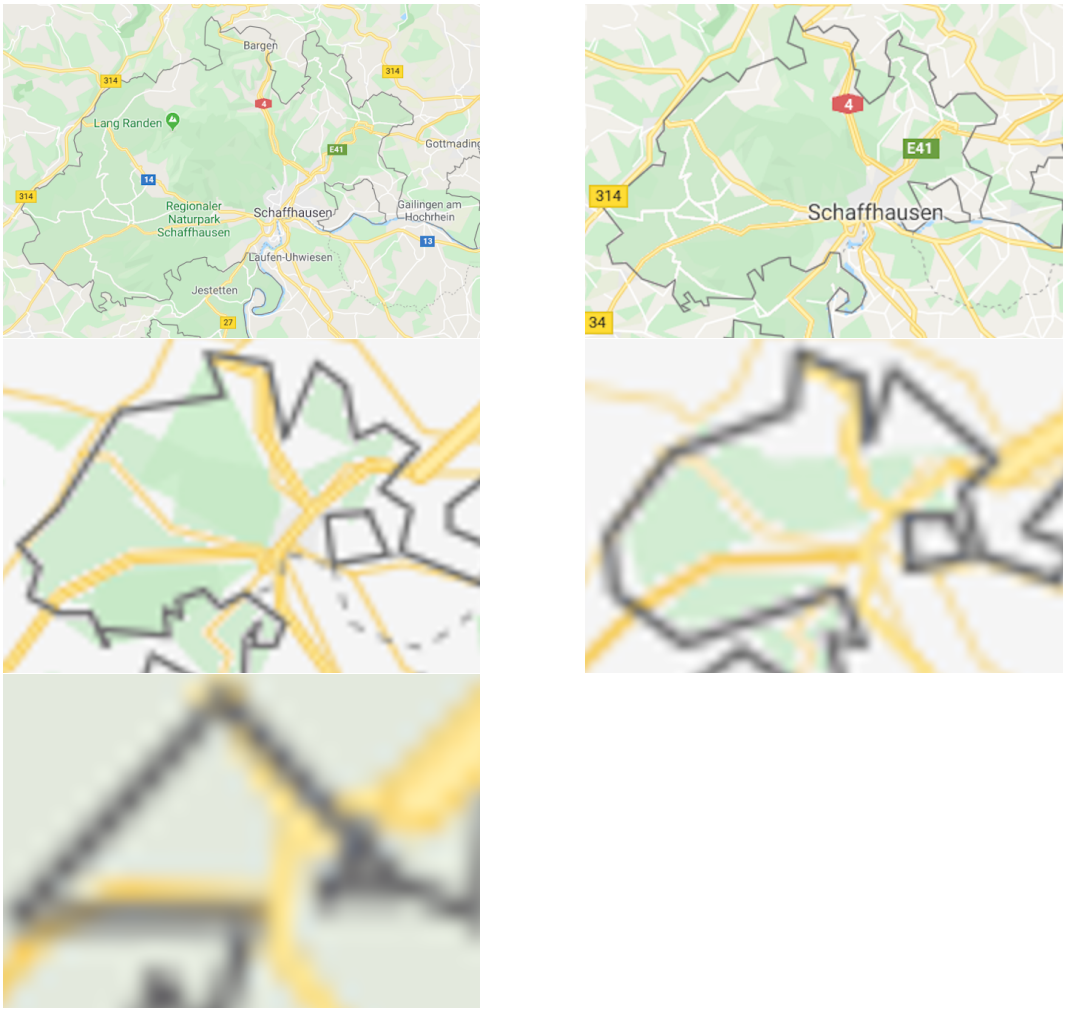


Figure 0.5.: Example of how Google Maps uses simplification. The pictures are scaled to show the same region on different zoom levels. The more one zooms out, the coarser the representation gets.

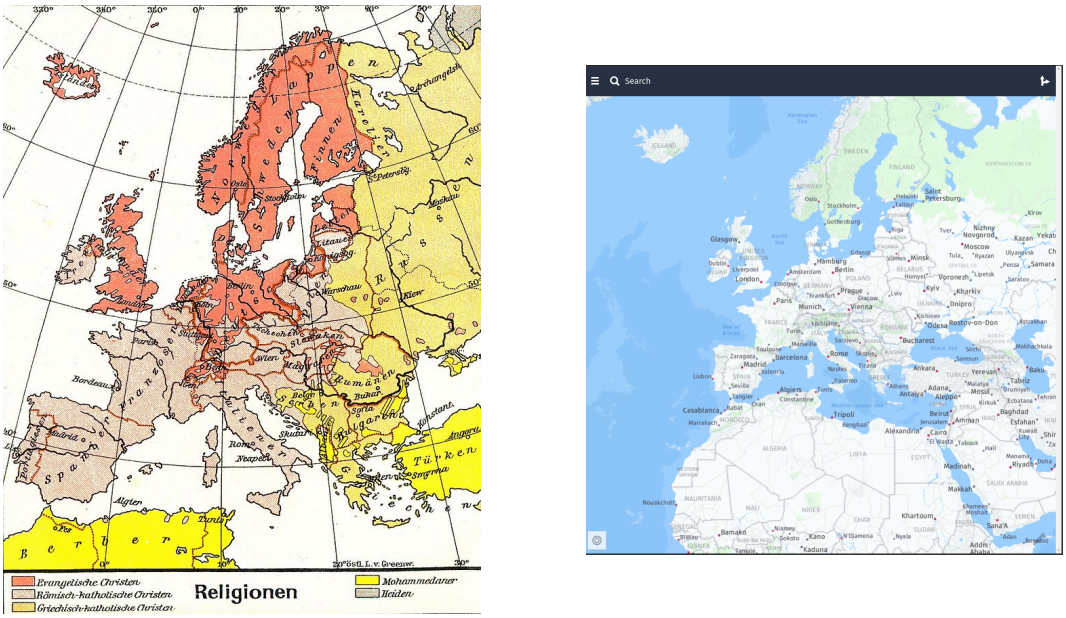


Figure 0.6.: Examples of classical labelings, which tended to be curved, and modern labelings which are mostly axis aligned.

ing to the distinct objects on the map and play an important role in localizing a position on a map. They also help in interpreting the map. Things like polygons and points of interest are relatively straight forward to display, because they are defined by their very position on the map. But there is a great deal of freedom on how to label them. Where should labels of point-like data (points-of-interest) or polygonal data (rivers, countries) be placed? While the labeling of point-like data mostly has to deal with the position of the text, polygonal data has even more freedom. The labels should visibly be assigned to the area. But it is unclear if they should be fully contained in the area. We can draw labels axis aligned or in a curved manner. Maybe the size of the label should be dependent on the polygon? Figure 0.6 show the labeling from a classical atlas, which uses labels that conform to the area's shapes, as compared to the labeling in a modern map-service, which uses axis aligned labels.

In the following we present our contributions. In the context of the geometric simplification of maps we provide a precise problem definition. We augment the classical definition with the useful notion of "local topological consistency". We present an ILP formulation which solves the problem exactly and is applicable for small instances. For larger instances we present a fast heuristic. Compared to the exact solution, we can see, that the heuristic computes good solutions

on small instances. On larger instances the heuristic is still able to compute a solution quickly.

For labeling areas of maps we improve an existing labeling scheme, so that it can be used in real-time contexts. This is achieved by removing unnecessary computations from the algorithm. This part was joint work with Filip Krumpe.

1. Map Simplification

One of the main challenges for rendering map data on the screen arises from the abundance of data. In its typical representation (e.g. in OpenStreetMap), a several hundred kilometers long highway consists of thousands of individual road segments. Rendering all of those nodes and even more so transmitting them over the internet is certainly a waste of time, in particular for a mobile device. So typically one *simplifies* the chain of segments by replacing subsequences of degree-2 nodes by single segments. Depending on the screen size and resolution, this can be done without really affecting the visual quality. The same holds for country borders and most of the available map features in general.

Especially in the context of map rendering, there are additional features to consider. For example we wouldn't want cities to end up in the wrong country or in the sea, after the simplification.

The following picture gives an example for these kinds of problems:

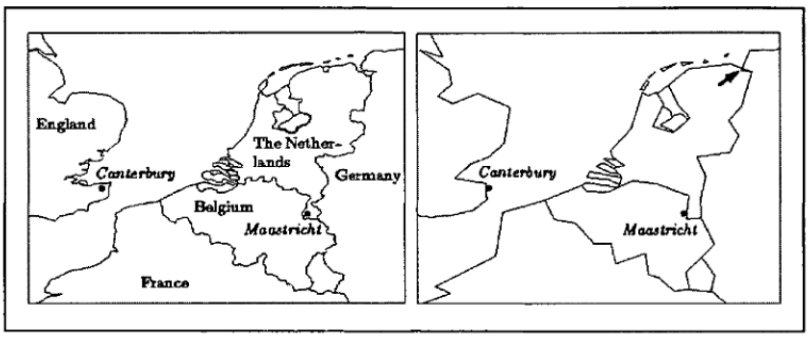


Figure 1.1.: A map of Western Europe with intersections and topology violations after line simplification of country boundaries (from [8], courtesy of de Berg et al.)

1.1. Problem Definition

A possible formalization for the problem is given by the following *Classical Line Simplification Problem*. Generalizations of the problem allow for more pleasant and reasonable output.

Line Simplification

In the classical line simplification problem (**CLSP**) we are given a polygonal chain $C = p_0 p_1 p_2 \dots p_n$ with $p_i \in \mathbb{R}^2$ and an error parameter $\varepsilon \geq 0$ and ask for a simplification of C. That is, indices $0 < i_1 < i_2 < \dots < i_k < n$ such that the polygonal chain $\widetilde{C} = p_0 p_{i_1} p_{i_2} \dots p_{i_k} p_n$ is a faithful approximation of C. Here 'faithful' means that for every 'shortcut' segment $s_j = \overline{p_{i_j} p_{i_{j+1}}}$ of the simplification, the furthest distance of a point in $\{p_{i_j}, \dots p_{i_{j+1}}\}$ to the shortcut segment s_j is at most ε. A natural optimization goal is to compute a faithful approximation with as few vertices as possible. That is, minimizing k. See Figure 1.2 for an example how the epsilon constraint can change the computed simplification.

Solving CLSP is of great interest, in particular in the map rendering context.

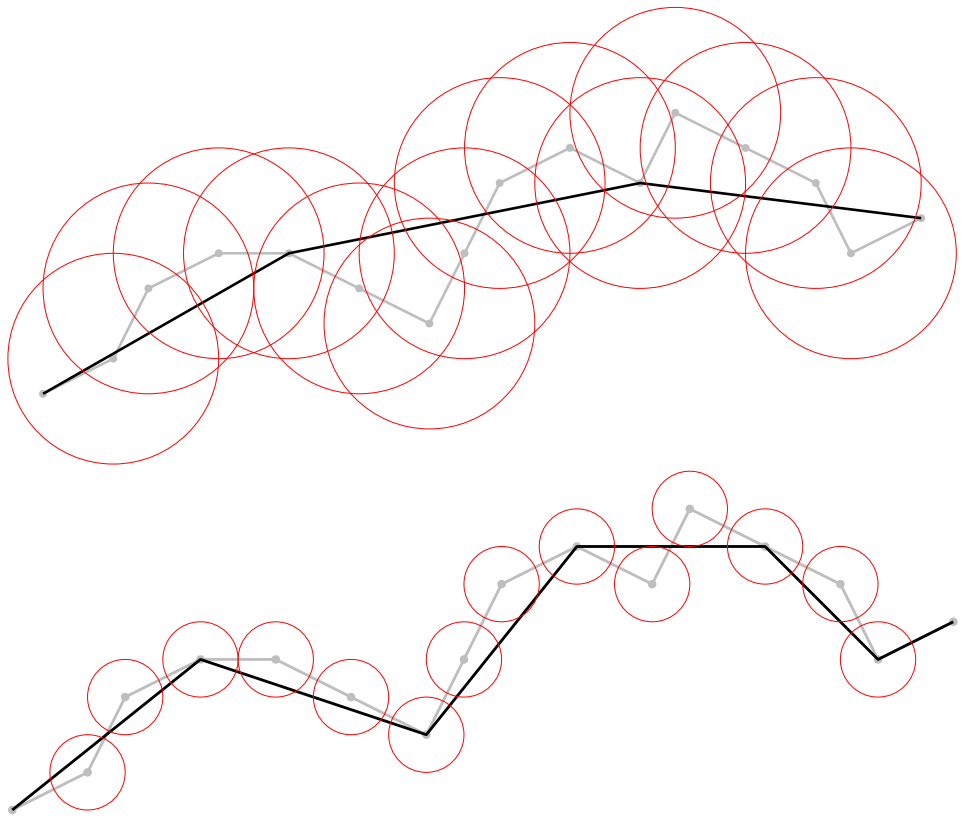


Figure 1.2.: Two simplifications for a single polyline with different ε-values. A circle of radius ε is drawn around every vertex. A larger value for ε allows for a simplification with fewer segments, but results in a coarser representation.

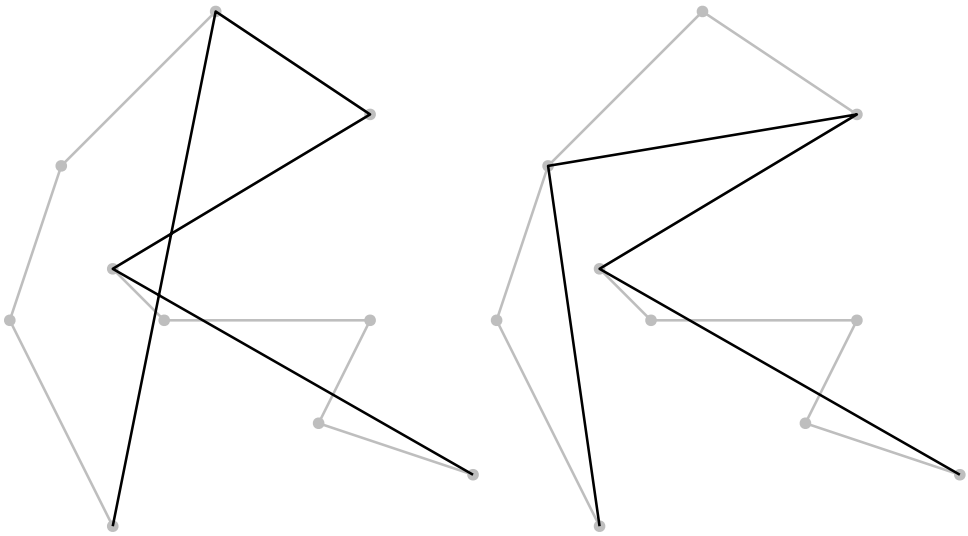


Figure 1.3.: The simplification on the left contains intersecting segments. The simplification on the right is intersection free.

Intersection Free

Naturally, simplification of degree-2-chains should not introduce intersections. So a sensible generalization of CLSP to the map rendering context is the map simplification problem (**MSP**). There, we are given a planar subdivision, in form of a planar straight-line embedding of a graph $G(V, E)$, and a parameter $\varepsilon \geq 0$. The goal is to solve CLSP for each maximal degree-2 chain of the graph without introducing intersections (within a single and between different degree-2 chains).

See Figure 1.3 for an example how simplifications can introduce intersections.

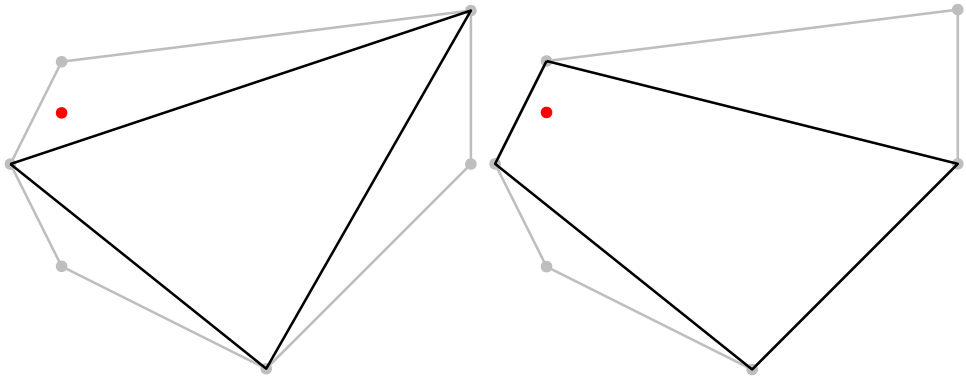


Figure 1.4.: With the left simplification a constraint point moves from the inside to the outside. The right simplification keeps the constraint point in the inside.

Topology constraints

Unfortunately, just solving MSP without additional care might lead to undesired effects (see Figure 1.1). In the simplification (right) of a map excerpt of Europe (left), some cities switched countries or ended up in the sea. This gives rise to a more general map simplification problem with topology constraints (**MSTOPOP**): Given a planar subdivision, a parameter ε, and a set of points $P \subset \mathbb{R}^2$, the goal is to solve CLSP in such a way that every point $p \in P$ remains in the same face as before. If some face is contained within another face, this relationship should also be preserved. See Figure 1.4 for an example, how simplifications must be aware of the constraint points.

In section 1.3 we will present a more local variant of this problem (**MSLOCTOPOP**), which better preserves spatial relationships in the map-rendering context.

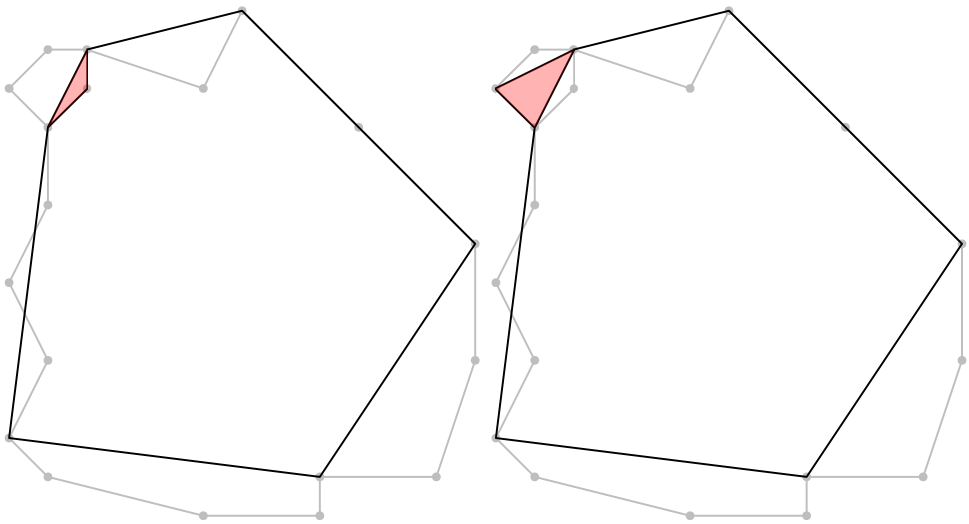


Figure 1.5.: In the left simplification the small area nearly vanishes because of the large ε value. Through area-preservation it maintains a larger area in the right simplification.

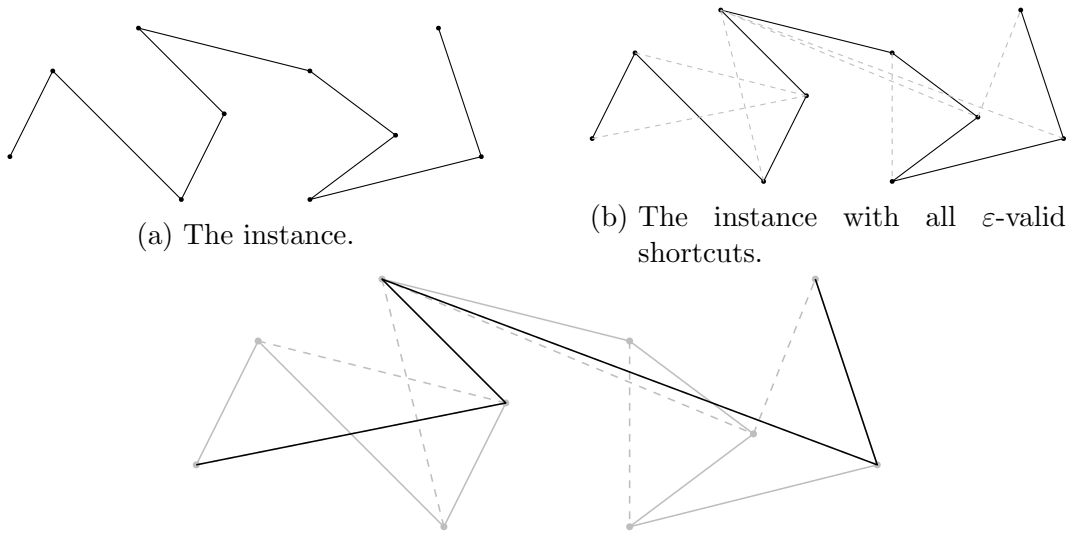
Area-Preservation

Even though the boundary movement is bounded by ε there might still be significant loss of area for (relatively) small features.

To counter this, an additional parameter $\delta \geq 0$ is introduced, which restricts the amount of area lost (or gained). So in addition to MSTOPOP we also enforce that for every face f with area A_f the area of it's simplified counterpart lies in the interval $[A_f \cdot (1/(1+\delta)), A_f \cdot (1+\delta)]$.

This is especially useful when rendering maps. By bounding the possible loss of area, we can make sure that every area roughly keeps its original significance. Even if large values of ε are applied to small features.

See Figure 1.5 for an example, how area preservation can help to maintain the visual impression for small details, even under large ε-values.



(a) The instance.

(b) The instance with all ε-valid shortcuts.

(c) The shortest path between the first and last node.

Figure 1.6.: Example run of the Imai/Iri-algorithm.

1.2. Related Work

For CLSP there are several known algorithms. The most popular being the algorithm by Douglas and Peucker [19]. Unfortunately, it neither guarantees absence of self-intersections nor optimality (i.e. minimum number of surviving points) of the result. Its worst-case running time is $\Theta(n^2)$, even though better running times are experienced in practice. Hershberger and Snoeyink in [34] showed how to speed-up the Douglas-Peucker algorithm to a worst-case running time of $O(n \log n)$. The algorithm by Imai/Iri [35] guarantees a result with minimum number of surviving points, but not the absence of self-intersections. Its running time is $O(n^3)$ in its original version, but an improved variant with a running time of $O(n^2)$ exists, see [14]. Figure 1.6 shows an example of this algorithm.

Estkowski and Mitchell [22] have shown, that for MSTOPOP (without topology points) it is NP-hard to obtain an approximate solution better than within a factor of $n^{1/5-\delta}$, $\delta > 0$. Their result carries over to MSP and its generalizations.

In [8] de Berg et al. consider a heuristic solution to MSTOPOP and MSP. Their algorithm has a running time of $O(n(n+m) \log n)$ where n is the number of vertices of the subdivision chains and m the number of topology constraints. Unfortunately, this algorithm has never been implemented and inherently bears a lower bound on the running time that is quadratic in the length of the

longest degree-2-chain, which to us made it not a very promising candidate for implementation of a fast heuristic.

For MSP an implementation is available in the CGAL-library [51] which follows [20].

There are also quite different approaches where the simplification does not have to use the original point set, e.g. [30]. Yet, also for this variant the authors could show, that computing a then so-called minimum-link simple polygon of a given homotopy type is NP-hard.

Another technique to simplify maps is given by Poorten and Jones [46]. Like our approach it is also based on Delaunay Triangulations. Their goal is different though: Rather than minimizing the number of vertices in the simplification they aim at preserving specific features of the input.

Instead of subsampling the vertices of a given subdivision, Goethem et al. [28] propose fitting curves to the input while preserving topology. They do not incorporate constraint points, though.

Abam et al. [1] consider a problem somewhat in between the ones stated above. For a given polyline, a set of constraint points, and an error bound the goal is to find an optimal simplification of the input line within that error bound, that is homotopic to the input line. They give a polynomial time algorithm to solve the problem. Further they define the notion of a strongly homotopic simplification, which allows exactly those simplifications our local definition does. They give an algorithm to compute all shortcuts that can be used in a strongly homotopic simplification, which runs in $O(n(n+m)log(n+m))$ time. After computing those shortcuts, they plug their results into the Imai/Iri framework to obtain a simplification. Unfortunately, their output might contain self-intersections.

As a side node, in the GISCup'14 – a competition held at the ACM SIGSPATIAL GIS conference 2014 – a variant of the problem (without a precision constraint – i.e., $\varepsilon = \infty$) was tackled by several teams.

Bose et al. describe different measures of area displacement in [11]. They only consider x-monotone polygonal chains. They propose an algorithm with quadratic running time for one of their measures. For the other two measures they prove $\mathcal{NP}$-hardness and propose an approximation algorithm for each them, again with super-quadratic running times.

In [13] Buchin et al. introduce an edge-move operation. This operation allows simplification of subdivisions that preserve topology as well as area. Additionally, no new edge orientations are introduced. This allows for nice schematizations but running times are rather high ("few hours" for maps with 1.8 million edges).

[43] describes an approach for area- and topology-preserving schematization,

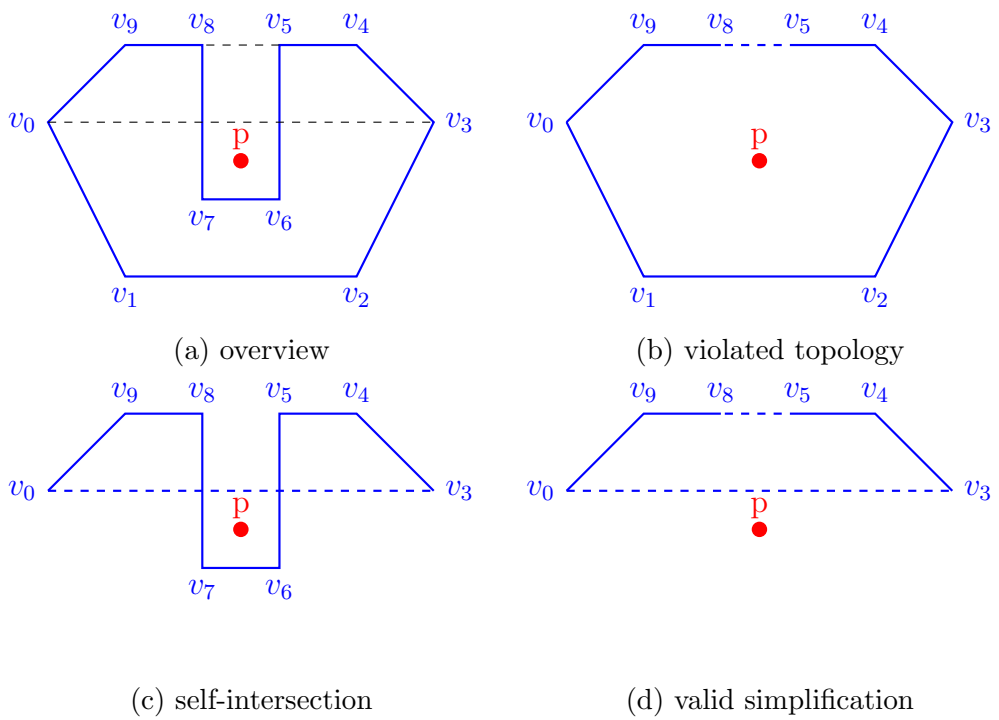
which outputs only rectilinear edges. This algorithm has again up to quadratic running times.

Our Contribution

This chapter presents an heuristic algorithm to solve the MSTOPOP-problem with additional area-constraints. We also present an ILP-formulation for the problem. The algorithm is evaluated on real-world data-sets. For the smaller data-sets we compute optimal solutions based on the ILP-formulation.

1.3. Concepts

Local Topology-Consistency



(a) overview (b) violated topology

(c) self-intersection (d) valid simplification

Figure 1.7.: Two invalid simplifications that become valid when applied simultaneously.

Consider the example in Figure 1.7, (a) where we have a planar subdivision with two faces – one U-shaped face bounded by $v_0v_1 \ldots v_9v_0$ and an outer face which also contains a topology constraint point p. For sufficiently large

values of ε, the simplification shown in Figure 1.7(d) is indeed a valid simplification according to MSTOPOP since p still lies in the outer face. This might be somewhat counterintuitive since p somehow 'switched sides' (even though topologically it is, of course, still in the right face). In particular, if we locally inspect the shortcut v_0v_3 replacing the chain $v_0v_1v_2v_3$ there is indeed a switch of sides; it is only valid by simultaneously shortcutting $v_5v_6v_7v_8$ by v_5v_8.

We believe that, in map rendering applications, it is more natural to demand that shortcuts *locally* do not make points switch sides. We therefore employ our own definition of topologically correct simplifications. This definition also matches what is implied by *strong homotopy* according to [1].

Definition 1. *For given $\varepsilon > 0$ and constraint point set P, a shortcut u_1u_k is considered a* valid shortcut *for the polygonal chain $C = u_1u_2 \dots u_k$ if*

- *the distance of u_i, $1 < i < k$ to the segment u_1u_k is at most ε.*
- *the polygon (possibly with self-intersections) defined by the polygonal chain*
  $C' = u_1u_2 \dots u_ku_1$ does not contain a constraint point (via the even-odd-rule [50]).

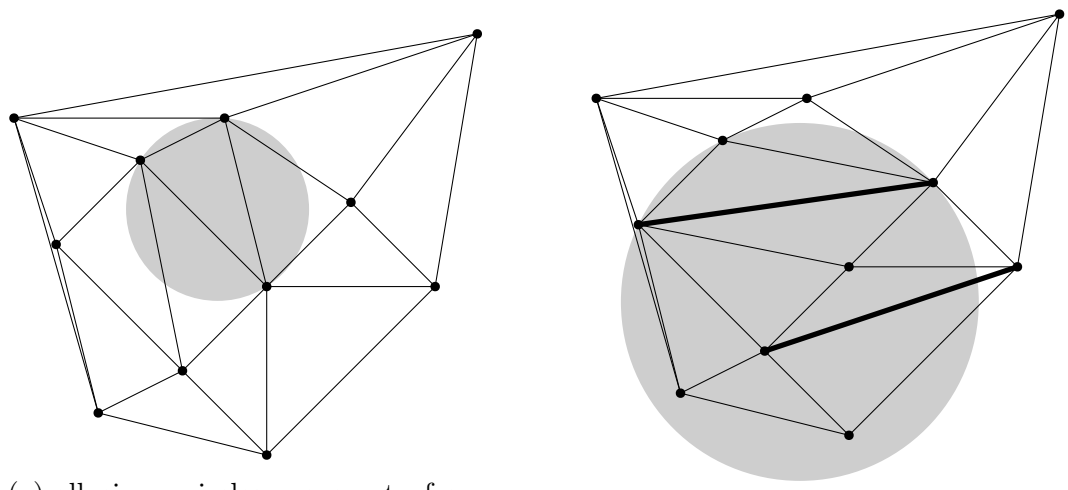
Definition 2. *For a planar subdivision, given as a straight-line embedding of a graph $G(V, E)$, a set of constraint points $P \subset \mathbb{R}^2$, and an $\varepsilon > 0$, the goal of the* Map Simplification with LOCal TOPOlogy constraints Problem (MSLOCTOPOP) *is to simplify degree-2 chains of G using non-intersecting valid shortcuts such that the total number of remaining vertices is minimized.*

As MSLOCTOPOP comprises MSP as a special case (with the set of topology constraints being empty), the hardness of approximation result in [22] carries over. Hence there is little hope to find a polynomial-time approximation algorithm which solves MSLOCTOPOP with an approximation ratio substantially better than $n^{1/5}$. In fact, avoidance of intersections during minimization seems to be the core of hardness, since with intersections not forbidden, [1] yields a polynomial-time solution.

1.4. Preliminaries

(Delaunay) Triangulations

Given a set of points $\mathcal{P} \subseteq \mathbb{R}^2$, a triangulation is defined as a partition of the Convex Hull of the point set into triangles, such that all triangles have their corners in $\mathcal{P}$. Such triangulations can be computed in time $\mathcal{O}(n \log n)$.



(a) all circumcircles are empty from point.

(b) Points in some circumcircle are "blocked" by a constrained edge.

Figure 1.8.: A point set and the corresponding Delaunay Triangulation (left) and Constrained Delaunay Triangulation (right). Bold edges indicate constraints. All circumcircles are empty from points, which are not hidden by a constraint edge.

A special case of a triangulation, is a Delaunay Triangulation. A Delaunay Triangulation is a triangulation with the special property, that any circumcircle of a triangle does not contain any point in its interior. Delaunay Triangulations are generally considered to be "nice" triangulations because they tend to have "fatter" triangles: A Delaunay Triangulation maximizes the minimum angle of its triangles.

Constrained Triangulations are a special case of triangulations. A constrained triangulation consists of a set of points, as well as a set of edges between some of the points. The edges should be non-intersecting. The goal is, to compute a triangulation of the point-set, which contains all of the input-edges.

We can also define, what a constrained Delaunay Triangulation is. Because some edges are required to be present, we can no longer guarantee, that all circumcircles are empty of points. But a weaker property can be guaranteed: Consider the constraint edges to be "blocking view". Then no circumcircle contains a point, which can be seen from the interior of the corresponding triangle.

See Figure 1.8 for examples of such triangulations.

There are a number of algorithms for computing (Delaunay) Triangulations:

flipping, incremental, divide-and-conquer, sweepline, via Voronoi-diagram-duality, and via 3d-convex-hull duality. See [33] for more details.

Computing the constrained Delaunay Triangulation of a simple polygon can even be done in linear time ([15]).

Linear Programming

Examine the following problem: As a company you can build different products from a given portfolio. You know the price for which each item can be sold and how much resources are in stock. The prominent question is: How much of each item should be produced, with a limited resource stock in mind, if the revenue should be maximized?

Consider the following notation: Let x_i be the amount of item i to be produced. Let p_i be the price per unit for item i. Let r_{ij} be the amount of resource j that is needed to produce one unit of item i. Finally s_j denotes the amount of resource j which is in stock.

Then the problem can be written as follows: We are looking for values of x_i which do not use more resources than we have in stock: $\forall j : \sum x_i \cdot r_{ij} \leq s_j$

From all possible values for x_i we want to select those, that maximize our revenue: $\max \sum x_i \cdot p_i$

This example shows the typical structure of linear programs, which is:

$$\max c'x$$

$$\text{s.t. } Ax \leq b$$

for real-valued vectors x, c, b and a real-valued matrix A.

Problems of this form can be solved using the simplex method ([16]). There even are polynomial time algorithms for these kind of problems, based on the interior point method ([37]).

A special case of Linear Programs are called Integer Linear Programs. For these programs, the values of the x_i should be discrete integers instead of real numbers.

While LPs can be solved in polynomial time, ILPs are NP-complete. This is rather easy to see, because we could encode 3-KNF into an ILP-instance. Even though there might be such really hard instances, there is a plethora of heuristics, which can handle even medium sized ILPs in reasonable time.

Because linear programs are used in a lot of industrial and business contexts there exists a lot of specialized software to handle such problems. One of those solvers is the Gurobi Optimizer offered by Gurobi ([31]). It offers

interfaces for different programming languages. The solver implements many highly engineered heuristics which are used to quickly solve the problems.

1.5. An ILP-based Exact Approach

This section describes an ILP-formulation for the MSLOCTOPOP problem with additional area constraints.

Let $\mathcal{C}$ be the set of chains, that is maximal sequences of degree-two points in the subdivision, where only the first and last point have degrees other than two. For any chain $c \in \mathcal{C}$, $|c|$ denotes its number of nodes and $c_0, c_1, \ldots c_{|c|-1}$ the sequence of nodes.

For every chain $c \in \mathcal{C}$ we introduce binary variables $x^c_{i,j}$. This variable indicates whether the nodes $c_{i+1}, c_{i+2}, \ldots c_{j-1}$ are replaced with the segment $\overline{c_i c_j}$ ($x^c_{i,j} = 1$) or not ($x^c_{i,j} = 0$).

The objective of the program is simply to minimize the sum of all variables:

$$\min \sum_{c \in \mathcal{C}} \sum_{0 \leq i < j < |c|} x^c_{i,j}$$

To make sure that chains are replaced with chains, we can use the following constraints. We start with exactly one segment from the first node and make sure every intermediate node has the same number of segments coming in and going out (that is: 1 if it is part of the simplified chain or 0 if it is omitted):

$$\forall c \in \mathcal{C} : \qquad \sum_{0 < i < |c|} x^c_{0,i} = 1$$

$$\forall c \in \mathcal{C} : \forall 0 < j < |c| : \qquad \sum_{0 \leq i < j} x^c_{i,j} = \sum_{j < k < |c|} x^c_{j,k}$$

The next set of constraints ensures we can only use variables whose corresponding segments are ε-valid:

$$\text{if } \exists i < j < k : \text{dist}(c_j, \overline{c_i c_k}) > \varepsilon \text{ then } x^c_{i,k} = 0$$

We only allow shortcuts that are locally topological consistent as described in section 1.3:

$$\text{if } \exists p \in \mathcal{P} : p \in \text{poly}(c_i, c_{i+1}, \ldots c_j, c_i) \text{ then } x^c_{i,j} = 0$$

Let orgArea denote the original area of a face and $\text{area}_f(c, i, j)$ denote the area of the complex polygon $c_i c_{i+1} \ldots c_j c_i$ and thus the area change of f, if

the corresponding shortcut gets realized (The sign of the area depends on the orientation of the chain, relative to the face). To keep the areas δ-close to the original areas, we employ the following constraint:

$$\forall f \in \mathcal{F} : \frac{1}{1+\delta} \cdot \text{orgArea}(f) \leq$$

$$\text{orgArea}(f) + \sum_{c \in \text{chains}(f)} \sum_{0 \leq i < j \leq |c|} x^c_{i,j} * \text{area}_f(c, i, j) \leq (1+\delta) \cdot \text{orgArea}(f)$$

Finally we need to prevent two segments from being picked simultaneously if they are intersecting:

$$\text{if } \overline{c_i c_j} \text{ intersects } \overline{c'_k c'_l} \text{ then } x^c_{i,j} + x^{c'}_{k,l} \leq 1$$

Complexity

The resulting ILP therefore has $V := \sum_{c \in \mathcal{C}} \binom{|c|}{2}$ variables.

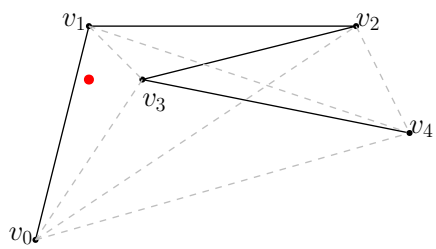
Additionally there are $|\mathcal{C}| + \sum_{c \in \mathcal{C}} |c|$ constraints to ensure a connected solution, up to V constraints for both ε-validity and topology-consistency, and finally up to $\binom{V}{2}$ constraints to avoid intersecting shortcuts.

Lazy constraints

Because the number of variables can be $\Theta(n^2)$ the number of *intersection-free* constraints can be $\Theta(n^4)$. Therefore these constraints should be implemented as *lazy constraints*. This means, we do not initially enter these constraints into the ILP-Solver, but check for the condition on every solution the solver presents us. If we detect an intersection, we add the constraints for the variables corresponding to the intersecting segments and have the solver continue with the new constraint-set. This process is repeated until an optimal solution without intersections is found.

Example

Let us go through the exact structure of the ILP for a small example. Consider the polyline on the right.



We introduce a variable for any pair of vertices, to represent all possible shortcuts. Therefore we have the following variables:

$$x_{01}, x_{02}, x_{03}, x_{04}, x_{12}, x_{13}, x_{14}, x_{23}, x_{24}, x_{34}$$

Because we want to end up with as few vertices as possible our objective is:

$$\min x_{01} + x_{02} + x_{03} + x_{04} + x_{12} + x_{13} + x_{14} + x_{23} + x_{24} + x_{34}$$

To make sure the simplified polyline starts at node 0 we pick exactly one of the outgoing shortcuts:

$$x_{01} + x_{02} + x_{03} + x_{04} = 1$$

To obtain a connected line we have to make sure to pick a shortcut from some note if and only if we also picked a shortcut ending in it:

$$\begin{aligned} x_{01} &= x_{12} + x_{13} + x_{14} && \text{(for node 1)} \\ x_{02} + x_{12} &= x_{23} + x_{24} && \text{(for node 2)} \\ x_{03} + x_{13} + x_{23} &= x_{34} && \text{(for node 3)} \end{aligned}$$

We can discard any shortcuts for which the ε-constraint is violated:

$$x_{02} = 0, \quad x_{03} = 0, \quad x_{04} = 0, \quad x_{13} = 0, \quad x_{24} = 0$$

To respect the local topology constraint the following shortcuts may not be chosen, because of the red constraint point:

$$x_{02} = 0, \quad x_{03} = 0, \quad x_{04} = 0$$

Finally, throughout the solving process we might need to add some of the following constraints to avoid intersecting shortcuts:

$$\begin{aligned} x_{02} + x_{14} &\leq 1 \\ x_{02} + x_{34} &\leq 1 \\ x_{14} + x_{23} &\leq 1 \end{aligned}$$

In the end the solution would be $x_{01} = x_{14} = 1$ and all other variables are 0.

1.6. A Heuristic Greedy Approach

While the ILP-formulation gives optimal results, it takes too much time to compute solutions even for medium sized instances. This is due to the quadratic number of variables. Therefore we propose an heuristic algorithm. To evaluate the quality of the algorithm, we compare its solutions to optimal solutions obtained via the ILP for small instances.

The heuristic is based on Constrained Delaunay Triangulations. We start with the original input and remove nodes one by one, maintaining a valid simplification at all times. The triangulation allows us to quickly decide whether a given node can be removed without violating any constraints.

1.6.1. Algorithm

Preparation

The algorithm starts by computing a constrained Delaunay triangulation. All nodes (endpoints of segments, as well as constraint points) become vertices of the triangulation. Additionally all segments of the input create a constrained edge in the triangulation. Afterwards we compute the area of each occurring face (of the input subdivision) as a baseline for our algorithm.

Main Loop

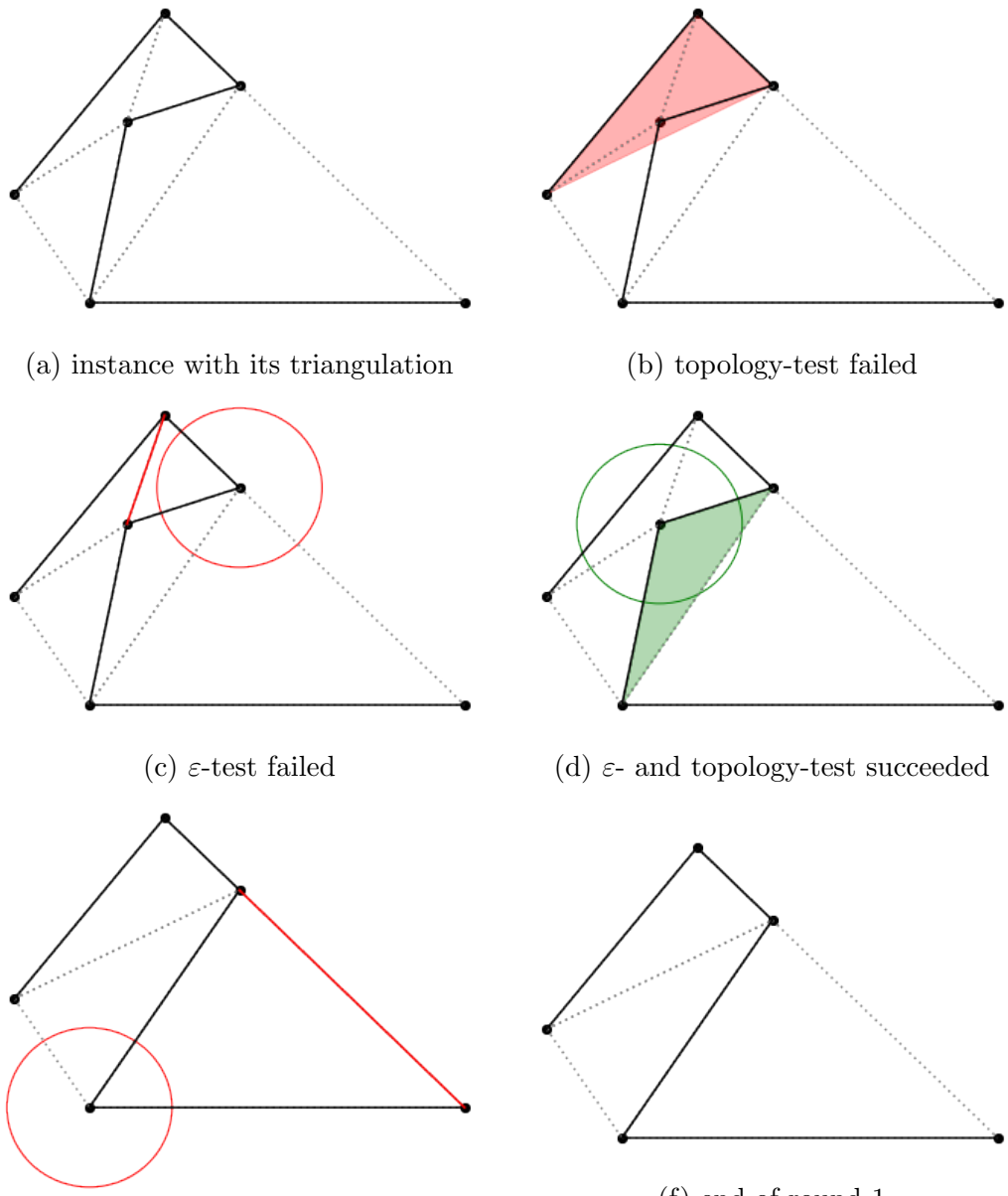
After initialization our algorithm follows a rather simple procedure. It considers all degree-2 nodes. For each node it checks the following conditions (let v be the considered node, and u and w its two neighbors):

- Is there any neighbor (in the triangulation) of v, that is not on the outside of the triangle $\Delta(uvw)$? (except u and w)
- If we subtract the area of the triangle $\Delta(uvw)$ from the face (in the subdivision) to the left of the segment $\overline{uv}$ and add it to the right face: Are the areas of both faces still δ-close to the initial areas?
- Are all nodes that were inbetween u and w in the original subdivision ε-close the shortcut $\overline{uw}$?

If all three conditions hold, we can remove v from the triangulation, unconstrain the edges uv and vw, and constrain the edge uw.

This loop is repeated until we removed less than a set fraction (e.g. 1%) of the remaining nodes.

Figure 1.9 shows a small example run of the heuristic.



(a) instance with its triangulation

(b) topology-test failed

(c) ε-test failed

(d) ε- and topology-test succeeded

(e) ε-test failed

(f) end of round 1

Figure 1.9.: Example Run of the algorithm. After triangulating the input, nodes are considered one after another and removed if possible. In the next round more nodes might be removed.

1.6.2. Bounding Distances

One of the main aspects of the algorithm is to always maintain a valid simplification. Making sure that the ε-constraint is fulfilled after some removal could be costly. Therefore we present techniques to make the check simpler.

exact

Whenever we need to know whether the segment $\overline{u_i u_j}$ is valid, we check all nodes inbetween. That is for each node u_k with $i < k < j$ we test whether the distance between the node u_k and the segment $\overline{u_i u_j}$ is less than ε.

upper bound

For each shortcut segment we keep an upper bound on how far it deviates from the replaced nodes. At first this upper bound is 0 for all segments.

When removing node v, with adjacent nodes u and w, the segments $\overline{uv}$ and $\overline{vw}$ would be replaced by the segments $\overline{uw}$.

Let's concsider a node p, which has been replaced by the segment $\overline{uv}$. The distance of p to the new segments $\overline{uw}$ is no more, than its distance to $\overline{uv}$ + the distance of v to the segment $\overline{uw}$.

That means that a valid upper bound for the new segment can be obtained by adding the distance between the removed node and the new segment to the maximum upper bound of the adjacent segments.

In some cases this upper bound might become very crude over time. Think of a sequence of points with a zig-zag pattern. The upper bound would grow with each removal, even though the maximum error is bounded by a constant.

This effect can be dampened by resetting the upper bound from time to time. The real value can always be obtained using the exact method.

ε-cones

For a sequence of points $p_0, p_1, \ldots p_n$, we sequentially check whether we can remove points $p_1, p_2, \ldots p_n$ and after each removal introduce the according shortcut. Imagine ε being large enough that we can remove all points. When removing point p_{i-1} we still have to make sure the points $p_1, p_2, \ldots p_{i-1}$ are all at most ε away from the segment $\overline{p_0 p_i}$. Naively we would compute the distance for all $i - 1$ points. This would lead to a quadratic running time.

We improve over this by generalizing the wedge-based approach of de Berg et al. [8]. Instead of only handling x-monotone chains we handle arbitrary chains. First we introduce the notion of an *ε-cone*. Figure 1.10 depicts such a

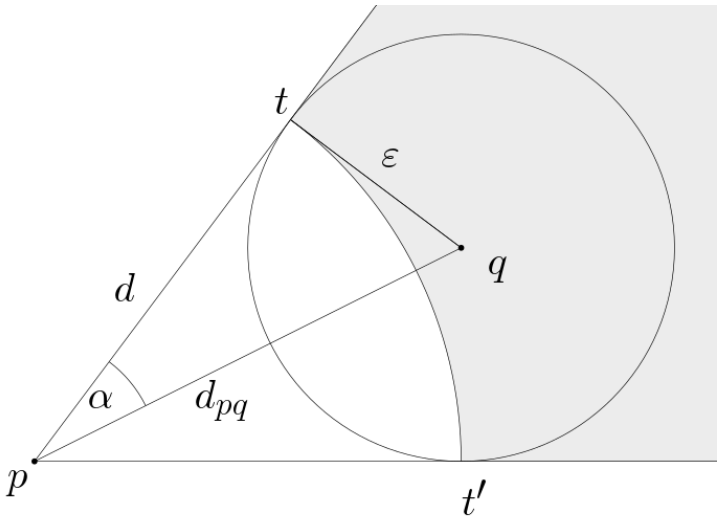


Figure 1.10.: ε-cone. For any Segment s originating from p to a point in the shaded area, q is at most ε away from s.

cone for two points p and q. Because ε is known all other values can easily be computed (notice that the angle $\angle ptq$ is 90 degrees). The ε-cone is the shaded region of the construction.

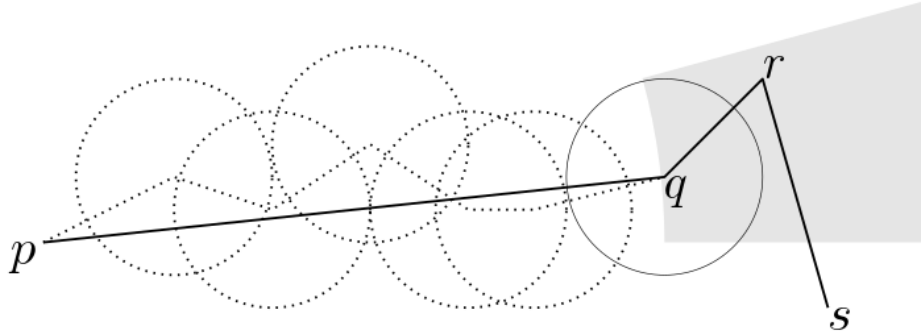
Given such an ε-cone of p and q, the following holds for any point r: If r lies in the ε-cone, q is at most ε away from the segment $\overline{pr}$. ε-cones can easily be intersected if they have a common source point.

In the mentioned example of removing nodes from a chain we can use these cones, by keeping the intersection of the ε-cones from all currently removed nodes, to reduce the complexity of the check from linear to constant time. Figure 1.11 visualizes the procedure.

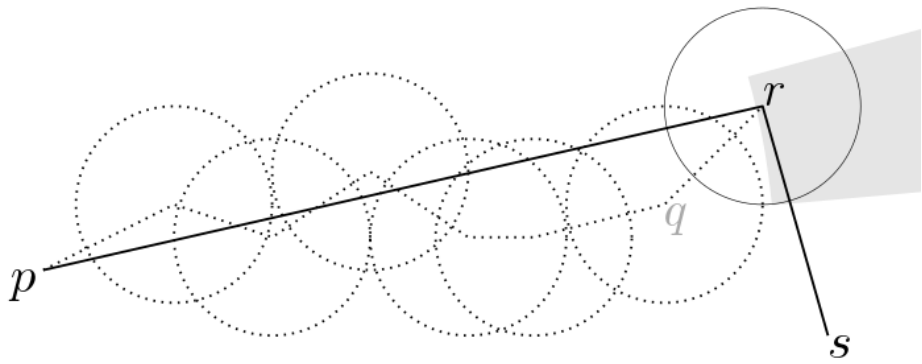
lazy combination

We can combine the methods of evaluating distances as follows. On the first run, when lots of nodes are removed, we use the ε-cone method. Later on we can employ a quick check via the upper-bound method and only resort to exact computation once this does not allow to prune any more nodes.

Furthermore, when we certified, that some node cannot be removed at a given time, this remains true until one of its neighbors has been removed. Only then could this node become removable at all.



(a) before removal of q



(b) after removal of q

Figure 1.11.: The first picture depicts the situation before removal of q. We can remove q because r lies in the intersection of all ε-cones seen so far. The second picture shows the situation after removal of q. We cannot remove r because s does not lie in the intersection of all ε-cones seen so far.

1.6.3. Managing Small Values Of δ

For very small values of δ the algorithm might get stuck, because every single removal would displace more area than δ permits. Even though the simultaneous combination of two or more removals would be feasible. To remedy this situation, we switch to the following strategy once the algorithm cannot remove any more vertices: For every chain we compute the set of vertices, which can be removed without considering the area-constraint. We now need to find a subset of those vertices, such that the sum of their respective area displacements is again feasible. We can only remove non-adjacent points safely at the same time, so this is an additional constraint to the selection problem.

We employ the following simple strategy to find a feasible subset of those points:

1. Greedily select a subset of non-adjacent points.

2. While the sum of the chosen subset is not feasible: Remove one element such that the sum of the remaining elements gets as close as possible to the feasible range.

This algorithm terminates when an appropriate subset has been found. It might return an empty set. In all cases we can then simultaneously remove all vertices in this subset.

1.6.4. Correctness

We start with a valid simplification (e.g. the trivial one, identical to the initial subdivision) and at all times keep it intact:

Lemma 1. *None of the performed removals makes the simplification invalid.*

Proof. The ε-constraints is taken care of via the ε-cones. To retain topology we only perform removals of nodes v (with neighbors u and w) if the triangle $\Delta(uvw)$ is empty. This condition holds if none of the neighbors of v (except u and w) lie within this triangle: Let $u_0, u_1, \ldots u_k$ be the sequence of neighbors of v starting with u and ending in w. Either any of them lies in the triangle or all of them lie outside. If any node is inside, we're done. If all of them lie outside, the union of their faces does not contain a vertex and forms a superset of the triangle. In this case we're also finished. (See Figure 1.12 for a visualization.) □

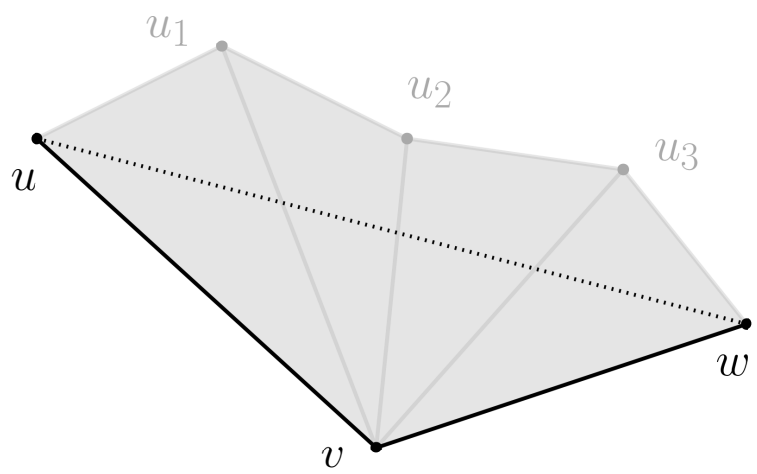


Figure 1.12.: If none of v's neighbors lies in the triangle $\Delta(uvw)$, the triangle is empty.

1.6.5. Running Time

The running time of the algorithm is bounded by the time to compute the initial constrained triangulation.

Lemma 2. *The algorithm takes time $\mathcal{O}((n+m)\log(n+m)+\sum D_i)$, with D_i being the degree (in the triangulation) of the i'th considered node.*

Proof. The initial constrained triangulation can be built in time $\mathcal{O}((n+m)\log(n+m))$. In each iteration of the loop we sweep linearly over the input and extend the ε-cones. Because we lose at least a constant fraction of the input nodes in every iteration (otherwise we stop) the total number of considered nodes is bounded by $\mathcal{O}(n)$. For every node we look at all of its neighbors. This takes time D, with D being the degree of the node. When we remove a node, we have to re-triangulate the resulting hole. This can be done in time $\mathcal{O}(D)$ ([15]).

The special procedure to handle small δ values can be executed in time $|c|\log|c|$ for any chain c. As the length of each chain is bounded by n the whole term is bounded by $n\log n$ and because of the exponential decrease of n the total work of all calls to this procedure is bounded by $n\log n$. □

In our experiments we never witnessed values of D greater than 50 and on average the observed values were less than 6. So in practice the algorithm behaves near-linear.

We employ a constrained Delaunay Triangulation instead of a regular constrained triangulation as it tends to keep the maximum node degree lower.

Because in practice nearly all nodes have small degree, we could modify the algorithm to fail the topology test for nodes with high degree (e.g. larger than a small constant). This would not make any noticeable difference for the algorithm, but it would allow to drop the $\sum D_i$ term from the theoretical bound of the running time.

1.6.6. Guarantees

Theorem 1. *The algorithm computes a valid simplification in time* $\mathcal{O}((n + m)\log(n + m) + \sum D_i)$.

1.7. Evaluation

In this section we provide some results about the running time and solution size of the implementations of our two approaches. Please note that the ILP solves the MSLOCTOPOP-problem, with the refined topological constraints. We use real world data sets of different sizes.

The experiments were executed on a single core of a standard laptop with an Intel Core i5-4300U CPU with 1.9GHz and 12GB of RAM. The triangulation based approach was implemented using the CGAL library [51]. For the ILPs we employed the Gurobi solver [31]. G++ 5.4 with the -O3 flag was used to compile the programs on a Linux installation.

We evaluate our algorithm on several real-world data-sets:

- GIS{1-5}: The data-sets from GISCup'14
- BW: County Borders of German federal state Baden-Württemberg with towns/cities/villages as constraint points
- GMY: Germany with all state-district borders and towns/cities
- EU: Europe with all federal-district-borders and cities

The data-sets BW, GMY and EU were extracted from OpenStreetMap.

1.7.1. Comparison Of Heuristic And Optimal Solutions

We compare our heuristic with optimal solutions obtained via the aforementioned ILP. Experiments are run on the GIS{1-5} data-sets as these are small enough to solve the ILP or at least compute usable lower-bounds. See table 1.1 for the results.

Consider the first column, which represents the instance "GIS1" The table shows, that it consists of 11 faces, 965 segments, and 26 constraint points. The first row of that column compares the results of the ILP, which computed an optimal solution, with the results of the heuristic for parameters $\varepsilon = 1$ and $\delta = 0.001$. The ILP computed a solution with 49 segments in 321.16 seconds, while the heuristic found a solution with 130 segments in 1.09 seconds. The first row of column "GIS4" states that the ILP found a solution of size 24136 and computed a lower bound of size 1777. The computation was terminated after 24 hours. The heuristic computed a solution of size 3427 in 32.85 seconds.

One can see, that both ε and δ have an effect on the running time.

A small value for ϵ means, that the algorithm terminates earlier, because it runs out of nodes to contract earlier. Let's consider the seconds column. The first 5 rows show the effect of decreasing ε-values for a fixed δ. As ε decreases the number of segments in both the optimal and the computed solution goes up. On the other hand the running times decrease. The ILP has to deal with fewer constraints, because all ε-invalid constraints can be discarded immediately. The heuristic also runs faster, because it runs out of valid shortcuts earlier.

Smaller values for δ make the problem harder, because the algorithm has to find multiple nodes to contract, so that the area-constraint stays fulfilled. The last 5 rows show what happens if we decrease δ. Naturally, the number of segments in the solutions goes up, because we have less freedom to choose the final set of segments. The running time increases also. This is because it gets harder for both the ILP, as well as the heuristic, to find a good combination of segments to fulfill the area-constraint.

With one exception (GIS3, $\varepsilon = .01$, $\delta = .0001$) our algorithm always manages to be within a factor of 3 to the lower bound obtained by the ILP. Also note, that the bounds for the seemingly bad cases are often far from tight and the algorithm might be even better, compared to the optimal solution.

1.7.2. Larger Instances

We evaluate our heuristic on several real-world data-sets. Even on continental sized instances the algorithm can compute reasonable small results in mere minutes. See table 1.2 for details.

Considering the dataset GMY, we can again observe the same trends as before. Reducing ε yields larger solutions, while running times decrease, because of earlier termination. Reducing δ also yields larger solutions, yet running times increase, because the heuristic searches for a valid combination of removals more often.

		GIS1	GIS2	GIS3	GIS4	GIS5
#faces		11	19	163	466	793
#segments		965	1518	8055	26661	25992
#constraint-points		26	124	151	356	1607
$\varepsilon = 1$	$\delta = .001$					
ILP solution (time)		49 (321.16s)	102 (1883.00s)	[1002, 692]	[24136, 1777]	[23001, 3529]
ALG solution (time)		130 (1.09s)	226 (2.93s)	1692 (17.00s)	3427 (32.85s)	6574 (56.25s)
$\varepsilon = .1$	$\delta = .001$					
ILP solution (time)		49 (668.47s)	102 (1598.37s)	[1002, 692]	[24136, 1777]	[23001, 3529]
ALG solution (time)		130 (1.27s)	226 (2.88s)	1692 (17.00s)	3427 (32.73s)	6574 (56.37s)
$\varepsilon = .01$	$\delta = .001$					
ILP solution (time)		87 (9.19s)	155 (36.29s)	[1187, 709]	[24475, 1785]	[22839, 3528]
ALG solution (time)		151 (0.81s)	262 (2.59s)	1692 (16.90s)	3429 (32.80s)	6577 (56.11s)
$\varepsilon = .001$	$\delta = .001$					
ILP solution (time)		490 (0.90s)	717 (1.99s)	2825 (5316.66s)	[2665, 2573]	[7288, 4771]
ALG solution (time)		538 (0.36s)	812 (1.10s)	3568 (9.61s)	3870 (32.48s)	7248 (55.14s)
$\varepsilon = .0001$	$\delta = .001$					
ILP solution (time)		853 (0.79s)	1264 (1.20s)	8004 (2.81s)	14488 (53.85s)	21086 (11.67s)
ALG solution (time)		856 (0.36s)	1267 (0.47s)	8004 (5.45s)	16388 (13.03s)	22339 (18.03s)
$\varepsilon = .01$	$\delta = 1$					
ILP solution (time)		84 (1.27s)	139 (2.48s)	579 (7.39s)	1476 (61.02s)	2600 (69.29s)
ALG solution (time)		98 (0.31s)	160 (0.44s)	588 (5.19s)	1491 (12.88s)	2622 (19.50s)
$\varepsilon = .01$	$\delta = .1$					
ILP solution (time)		84 (1.89s)	145 (5.73s)	597 (340.63s)	1564 (28128.73s)	2776 (19741.66s)
ALG solution (time)		98 (0.31s)	172 (0.46s)	624 (5.27s)	1637 (12.87s)	2900 (20.06s)
$\varepsilon = .01$	$\delta = .01$					
ILP solution (time)		86 (1.84s)	151 (10.57s)	[690, 678]	[22166, 1686]	[3476, 3123]
ALG solution (time)		107 (0.35s)	206 (1.84s)	963 (9.69s)	2250 (20.96s)	4259 (34.13s)
$\varepsilon = .01$	$\delta = .001$					
ILP solution (time)		87 (9.19s)	155 (36.29s)	[1187, 709]	[24475, 1785]	[22839, 3528]
ALG solution (time)		151 (0.81s)	262 (2.59s)	1692 (16.90s)	3429 (32.80s)	6577 (56.11s)
$\varepsilon = .01$	$\delta = .0001$					
ILP solution (time)		88 (441.45s)	[162, 158]	[7348, 727]	[26661, 1818]	[24344, 3785]
ALG solution (time)		201 (2.22s)	357 (3.48s)	2536 (20.94s)	4734 (37.98s)	9811 (67.87s)

Table 1.1.: Comparison on GISCup'14 data

		BW	GMY	EU
#faces		84	89	5182
#segments		137732	407538	3433890
#constraint-points		3996	2488	1092
$\varepsilon = 1$	$\delta = .001$	1328 (18.43s)	1693 (30.85s)	63544 (170.40s)
$\varepsilon = .1$	$\delta = .001$	1328 (19.27s)	1693 (30.95s)	63545 (169.00s)
$\varepsilon = .01$	$\delta = .001$	1363 (19.05s)	1702 (30.13s)	63550 (170.15s)
$\varepsilon = .001$	$\delta = .001$	4201 (16.57s)	3449 (28.20s)	64197 (166.14s)
$\varepsilon = .0001$	$\delta = .001$	22957 (10.14s)	18897 (27.11s)	120813 (250.79s)
$\varepsilon = .01$	$\delta = 1$	762 (9.69s)	809 (21.40s)	11342 (217.17s)
$\varepsilon = .01$	$\delta = .1$	813 (9.65s)	1116 (23.58s)	23218 (225.83s)
$\varepsilon = .01$	$\delta = .01$	967 (15.49s)	1247 (27.09s)	33331 (250.97s)
$\varepsilon = .01$	$\delta = .001$	1363 (19.05s)	1702 (30.13s)	63550 (170.15s)
$\varepsilon = .01$	$\delta = .0001$	1988 (28.53s)	3491 (33.86s)	124909 (310.63s)

Table 1.2.: Results on OSM data

1.8. Conclusions

Our proposed scheme differentiates itself from most other schemes in multiple ways. We can rule out algorithms, which do not guarantee intersection-free results, because this is one of the main reasons why this problem is hard at all. Furthermore most of the proposed algorithms have a super-quadratic running time, which prevents their use on large, real-world data-sets.

The algorithm by Imai/Iri ([35]) is a simple algorithm, which computes all possible shortcuts and then looks for a minimum link path in the resulting DAG. Because the number of shortcuts can be $\Theta(n^2)$ this algorithm has a worst-case running time of $\Omega(n^2)$. Depending on the cost of computing the actual shortcuts the running time could be even higher.

The algorithm of Abam et al. ([1]) uses the Imai/Iri algorithm as a framework to include constraint points. They compute all shortcuts that can be used without any topological inconsistencies and then also compute the minimum link path in the resulting graph. Their approach to compute all valid shortcuts has super-quadratic running time, though.

The CGAL implementation ([51]) is actually somewhat similar to our algorithm. Even though it does not incorporate constraint points, nor does it allow for area-preservation. It also uses a triangulation to quickly decide whether a node can be removed without introducing topological errors. The nodes are tested and removed according to their distance to the shortcut segment in ascending order. This means nodes which introduce a small error are removed first. When all remaining nodes would introduce an error greater than ε the algorithms stops.

Our algorithm distinguishes itself by allowing constraint points as well as area-preservation. Furthermore our ε-cones help to speed up the computation, as they allow for quick ε checks, when considering the nodes of a polygonal chain one after another.

One of the few schemes which allows area-preservation is proposed by Buchin [13]. But the main problem they solve is different from ours: Their goal is to compute a schematization of a map. Throughout their algorithm they maintain the orientation of edges while modifying them via an "edge-move" operation. Their scheme also allows for restricting edge orientation to a specific set of angles. When applied to similar sized instances their scheme takes "hours", when our algorithm computes a solution within few minutes.

Future Work

While removing points is the obvious way to reduce the number of points it would be interesting to also allow the use of additional vertices. Those vertices could lie on the poly-lines themselves or might be completely new points. With some cleverly chosen steiner points the simplification might yield even smaller results.

2. Area Labeling

The work presented in this part is joint work with Filip Krumpe.

Map labeling is a vast problem in which one aims at annotating text to maps. One can label point-like data (points of interests), one-dimensional data (streets, rivers) or two-dimensional data (lakes, countries). Each of these problems has its own set of challenges.

When labeling points, one has to decide which points can receive a label and where to place the label, so the labels don't intersect. When labeling streets and rivers, one has to decide if the labels should repeat multiple times (e.g. on a long street), or if it should bend around sharp corners for example. For areas one has to decide where to put the label. Does the label have to be completely inside the area? If there are multiple large spaces joined by narrow paths, should the label be written in all of the spaces separately? If not, in which should it be written? Should the label be straight and axis-aligned or can it be bent?

Recent literature recommends to label an area with a label that is completely contained within the area and fits the rough area shape.

Figure 2.1 shows how Google Maps labels countries and lakes. Countries are receiving axis-aligned labels, while the labels for lakes are bent along the shape of the polygon. Figure 2.2 depicts labelings occuring in printed maps. One can see, that printed maps also make use of bent labels.

Figure 2.3 shows a labeling of European countries, computed with our proposed approach.

Especially in interactive maps, one is faced with the problem that areas may only be partially visible when panning the map view. In a static setting, labels might not be visible for those areas. This leads to problems as displayed in Figure 2.4. The static labeling (left) is no longer appropriate because the label Russia is not completely visible. A better labeling is displayed in Figure 2.4 where the subpart of Russia currently contained in the viewport is labeled dynamically.

2.1. Related Work

Eduard Imhof in 1975 (see [36]) systematically described what a good labeling of a map is like. The main goal is to provide a good readability, i.e. that the

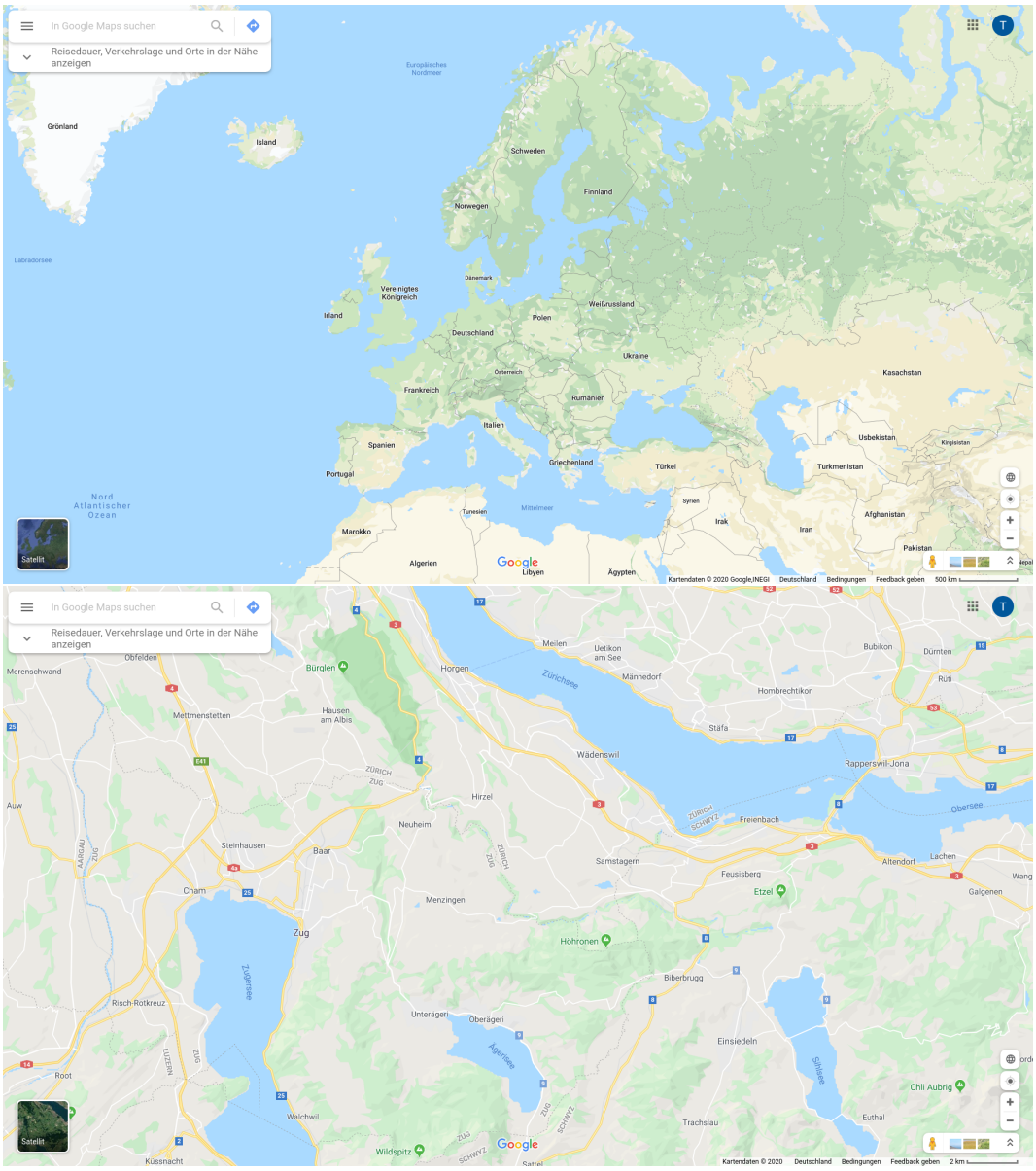


Figure 2.1.: In Google Maps, the countries are labelled axis-aligned. The labels for lakes are bent.

(a) Photograph from "Diercke Weltatlas".

(b) This picture is property of "landkartenindex.blogspot.com".

Figure 2.2.: Examples of curved labelings.

user can easily identify the feature corresponding to a label. Furthermore, the labels need to be non-overlapping, they should also reflect the importance and classification of its feature, and the map should be labeled with a good density. Concerning area features, he recommends to label it either with a horizontally aligned label or a curved label which is conform to the area shape. In both cases, the label should be completely contained within the area and leave a free space of one to one-and-a-half of a character to the area boundary. In case of a curved label, it should be based on a circular arc with a circular angle smaller then 60 degrees.

In the most recent publication in this field, M. Barrault in 2001 proposed a set of criteria to measure the quality of an area label [3]. When displaying an area label, there are several degrees of freedom, i.e. differing inter-character, -word and -line spacings. These can be adopted to better fit the label into its corresponding area. In the same publication, he is introducing an algorithm to compute area labels based on his proposed quality measures. This algorithms is based on the skeleton of the area, which is used to approximate the general shape of the polygon. The concrete label position is constructed by approximating paths in the skeleton by circular arcs. These arcs are used to place a label and evaluate the positioning. The best of these candidates is used to label the area. Barrault in his work also describes some shortcomings of his approach. The most important one is related to the enumeration of the paths which are considered to be the most promising candidates to be further evaluated.

In her bachelor thesis, N. Mendel reimplemented and evaluated his results on real world area data of the German state [41]. She could reproduce the shortcomings and showed that best candidates are often not considered at all. She also pointed out that the quality measure formula in Barrault's work contains an error leading to unwanted results. Details and a fix are provided in her work.

Quality Measures for Area Labels

To measure the quality of an area label Barrault in [3] is evaluating six criteria. In the following, the term *longitudinal* is used to describe the left-right dimension and respectively *latitudinal* for the top-bottom dimension.

Longitudinal extend: The extent along a circular arc should be maximized.

Longitudinal centre: The label should be centered in the polygon in the longitudinal dimension ...

Latitudinal centre: ... as well as in the latitudinal dimension.

Conformity: The base arc of the label should be conform to the shape of the labeled area.

Orientation: The more horizontal the label, the better.

Curvature: A label based on an arc with larger radius is preferred.

The center point and radius of a circular arc are defining the support line of the label - the line along which the label is bent. The possible label position is bounded by the points where the arc is intersecting the polygon. A label position is determined by two angles describing start and endpoint of the label along the support line. This concrete position is called the *baseline* of a label. A valid baseline length is obliviously determined by the label length and the different spacings. The latter are variables and in a general case can freely be chosen from given ranges.

The quality of a label candidate is evaluated by its perceived coverage of the polygon (PC). It is computed via an integral along the baseline, summing the minimum of the space above and below the label. An additional cost is induced if the endpoints of the label are too close to the border of the area. This is to prevent labelings from looking crammed.

Figure 2.3.: Labeling of the European countries with the automated labeling process we propose.

Figure 2.4.: Labeling of the European countries with Russia labeled statically (left), and with a dynamic labeling (right).

2.2. Preliminaries

2.2.1. Medial Axis

The **medial axis** of a planar shape was first defined by Blum in 1967 [9] and Lee in 1982 [39] amongst others. Given a simple polygon P representing the shape, its **medial axis** is defined as the locus of points p internal to P such that at least two points on the polygon's boundary are equidistant and closest to p. This definition can be applied to polygons containing holes in a straightforward manner. Each point on the medial axis can be assign a radius, describing the distance to the boundary. [18]

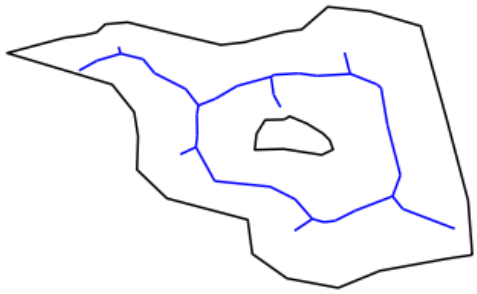
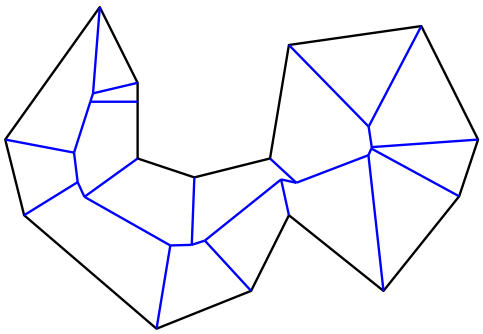


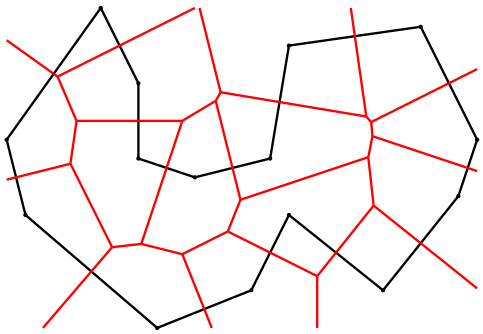
Figure 2.5.: Skeleton of a polygon with a hole.

As Schmitt in [49] and Brandt in [12] point out the medial axis, also called skeleton, of a polygon can be approximated using Voronoi Diagrams. The medial axis in this case is a special subset of Voronoi edges namely those who are completely contained in the polygon [40]. For each of these, we can approximate the minimum distance to the polygon boundary, which we will call its clearing (for details see Section 2.4.3). We are going to use this clearing to find paths through the skeleton-graph which offer a good amount of space to fit the label.

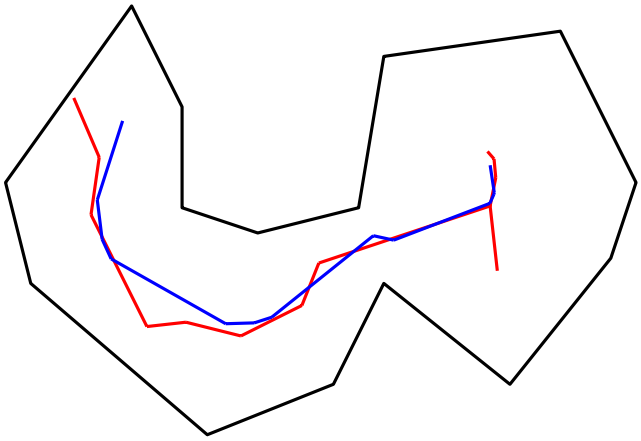
Figure 2.6 show the similarity of a polygon's skeleton to the internal edges of its Voronoi Diagram.



(a) the skeleton



(b) the Voronoi graph



(c) Overlay of parts of the Voronoi Diagram and the skeleton.

Figure 2.6.: Approximating the medial axis of a polygon via its Voronoi Diagram.

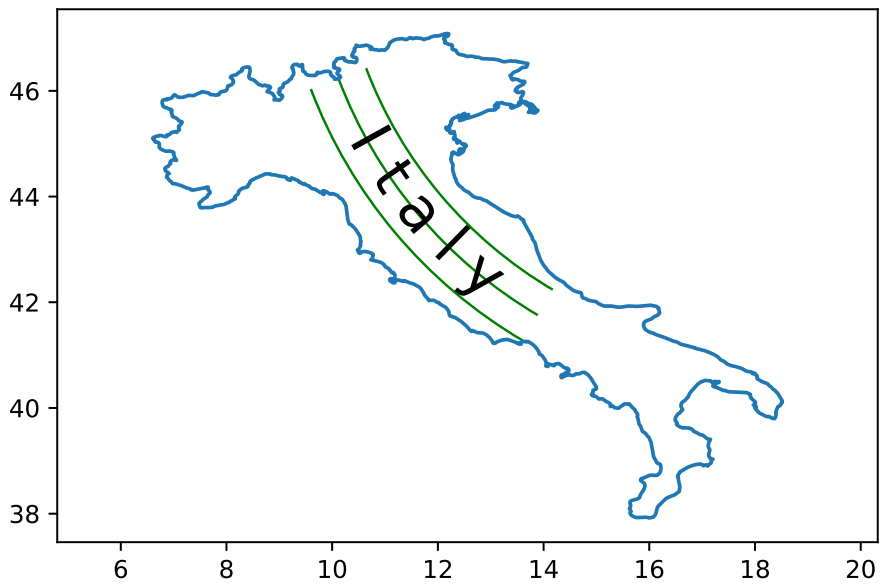


Figure 2.7.: A circular arc inside a polygon, with a text label along the arc. The upper and lower arcs indicate the maximal vertical extend of the letters. A space character is inserted to the left and to the right of the label for visual clarity.

2.3. Problem Definition

To get a formal definition of the problem we want to solve, we settle for the following setting. We want to place a text label inside a polygon. The text should be printed with a fixed inter-letter spacing. For a given string, this yields a ratio of its height and length, which we will call its *aspect*. The text label should be fully contained inside the polygon and the way it is placed should resemble the shape of the polygon if possible. To reduce the amount of freedom, we aim at placing the label along a circular arc. The space occupied by the label should be maximized.

That means, given a polygon and a text label, we want to compute a circular arc, starting and end angles such that the size of the label is maximized if placed along this part of the arc. Figure 2.7 shows an instance of this procedure.

2.4. Algorithm

2.4.1. High-level Idea

As we want to place our labels along a circular arc, we first have to find some arcs, which are a reasonable fit to the polygon. To find such an arc, we use an approximation of the medial-axis. A long path through this graph should be an appropriate representative of the area's shape. Because we want our labels to be placed along circular arcs, we fit a circle through the vertices of the path. Multiple candidates are enumerated and evaluated according to the optimal label-placement along the arc. Of the so obtained labelings, the largest is reported.

2.4.2. Barrault's Incarnation

Barrault's algorithm follows the steps described above. To decrease the complexity of the input polygon, morphologic erosion is applied. For the eroded polygon, a Delaunay Triangulation is computed. For each of the Delaunay triangles, a convex-combination of its corners defines a "center point" of the triangle. Those "center-points" of adjacent triangles are connected and thus form the edges of the medial-axis approximation.

After approximating the medial-axis in this manner, the 50 longest shortest paths are considered as candidates. A circle is fitted through each of them and investigated further as a possible label support line.

To find an optimal label placement for each support line, all possible placements (discretized) are considered. That is, every possible combination of starting and ending angles. For each placement, the label is evaluated. The placement with the highest score is then returned as the optimal label.

A major drawback of Barrault's approach is his choice of the 50 paths he is evaluating. These paths are mostly very similar and so are the fitted circular arcs, as N. Mendel shows in [41]. As a result, many promising alternatives are not considered at all. Additionally, the evaluation of the possible label placements contain several integral computations, all of them being computationally intensive. Overall, the computation takes a long time and in many cases does not even leads to good results.

2.4.3. RALF – Real-time Area Label Fitting

We go beyond Barrault's algorithm in several points. Firstly, we use a medial axis approximation based on the Voronoi graph. This allows for each edge

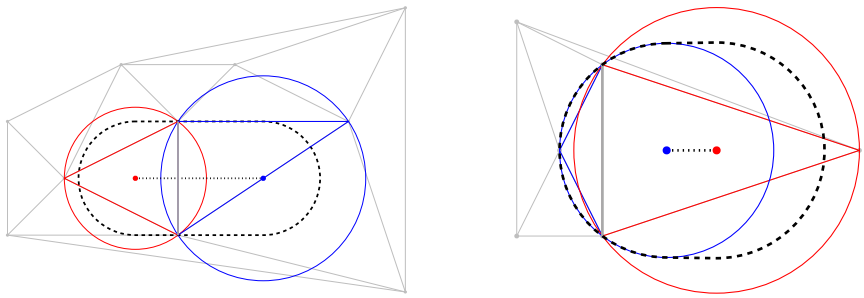


Figure 2.8.: The clearance of a Voronoi edge if the centers are on the same side (left) or on different sides (right) of the Delaunay edge.

in the medial axis to approximate the minimum distance to the boundary polygon. We call this distance the clearance of an edge.

This clearance value is then used to find paths in the skeleton which are promising to fit a large label through. This discards paths that are to close to the border of the area, which would restrict the label size.

We also improve path selection by computing a more diverse set of paths.

A third improvement is an approach to finding an optimal position for the label along a candidate arc. Here, we are proposing a new scheme to compute an optimal placement along the arc.

Space Around The Medial Axis

To get an approximation of the medial-axis where we are able to bound the distance to the boundary polygon, we proceed as follows: We compute the Delaunay Triangulation of the boundary polygon. For each Delaunay triangle, the Voronoi center is defined as the center of the circumcircle of the Delaunay triangle. We connect the Voronoi centers of adjacent Delaunay triangles if this so called Voronoi edge is completely contained within the polygon. For these edges, we can approximate the clearance, i.e. the minimum distance to the boundary polygon. We need to distinguish two cases: If the Voronoi centers are on different sides of the Delaunay edge, the minimum clearance is half the length of the radical line of the two circumcircles (i.e. the Delaunay-edge itself). In the second case, both Voronoi centers are located on the same side of the Delaunay edge. Then the clearance is the minimum of the radii of the two corresponding circumcircles. See Figure 2.8 for an example.

The rational behind this is as follows: All points closer to the segment than the computed distance are also contained in at least one of the Voronoi-balls. Those balls are empty of other points by definition of a valid Delaunay Tri-

angulation. Therefore, there are no points within the cleared segment. The clearance of the segment might still intersect the boundary of the polygon. But this can be remedied if the boundary is sampled sufficiently dense. Furthermore, we only use these clearance-values as guidance but do not rely on them for correctness (i.e. keeping the labeling within the polygon).

The so constructed skeleton graph has an associated clearance value for each of its edges.

Finding Candidate Paths

Having constructed the skeleton graph and the clearance values, one wants to find promising paths in the skeleton. These paths should allow to place a label of maximum size.

We aim to find a set of k diverse candidate paths which we further investigate to place a good label. Our strategy is based on the following observation: If we place a label along a given path, the minimum clearance of the path-edges hints at the maximum possible height of the label along this path. We therefore are looking for paths with a high minimal clearance, whose length allows to fully utilize the vertical space promised by this clearance. That is the length of the path should be no less than $l_{min} = \frac{2*clearance}{aspect}$.

The idea is to start with a large clearance value (e.g. the maximum clearance value) and remove all edges of the skeleton which have smaller clearance value. In this subgraph, we search for shortest paths such that their length is larger than the appropriate minimum length. If we can't find enough paths, we reduce the clearance and search for the remaining paths in the subgraph filtered with the new clearance. In our case, we start the first round with the maximum clearance in the graph and reduce it by $\sqrt{2}$ each round. I.e. we half the area of the label box we search for.

In detail, we proceed as follows: In each component of the pruned skeleton, we start with an arbitrary node and search for the node which is furthest away. This is done with one dijkstra call by tracking the root node of every shortest-path-tree. The so found nodes form our set of start nodes. We now search for the node which is furthest from our set of start nodes - also requiring only one dijkstra call with all the nodes as sources. The so found pair of nodes approximates the longest shortest path in the pruned skeleton (this method is exact for trees but not for arbitrary graphs). If the path length is larger than l_{min}, we report the path and add its vertices to the set of start-nodes. If we did not yet find k paths, we repeat the search with the new set of start-nodes. If the found path is of shorter length, we decrease our clearance by $\sqrt{2}$, refilter the graph and proceed as described. We repeat this until we have found k

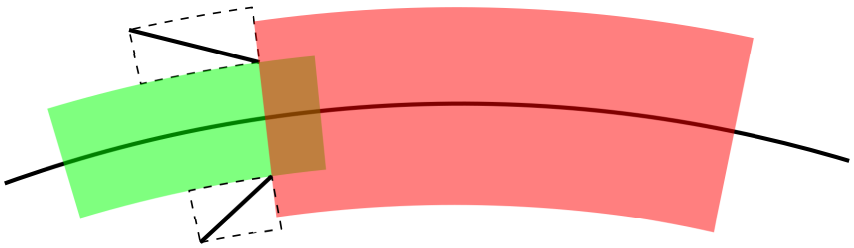


Figure 2.9.: The segments of the polygon restrict the label size. If the label is placed below or above a segment, the segment constrains the possible size (green label). We have to move the label considerably to the side so it can grow (red label).

paths.

A circle is fitted through each of the candidate paths. Let $p_1, \dots p_n$ be the points of the path. We compute a center c and a radius r such that the term $\sum_{i=1}^{n} (|p_i - c| - r)^2$ is minimized.

Label Placement

Given a circle, a polygon and a text label, we aim to find the position along the arc such that the size of the label can be maximized. We can compute this optimal placement in time $n \log n$, where n denotes the size of the polygon.

Let us first consider how a single polygon-segment constrains the label placement. We employ the following simplification: A circular bounding-box is constructed around each polygon-segment. For a given circle, this object has the same angular extent as the segment. Its inner and outer radius are defined by the minimum and maximum distance between the segment and the center of the circle.

There are two cases: First if the label size is restricted by the segment in its height, then we can move it along the arc without getting any benefit. In the second case, the size of the label is restricted by the segment in its length. In this case, the size of the label increases if we move the center of the label away from the segment. The more we shift it away from the segment, the more we can increase the size of the label. If we consider the possible size of the label as a function of the angle where the label is placed on the circle, we get a piecewise function with three parts: When the label gets closer to the segment the possible height decreases until it can fit below/above the segment. It then stays constant, while passing above/below the segment and finally increases. Let's call those functions "wedge". For a given angle, they tell us how large a label can be if it is placed at this angle on the circle. This is illustrated in

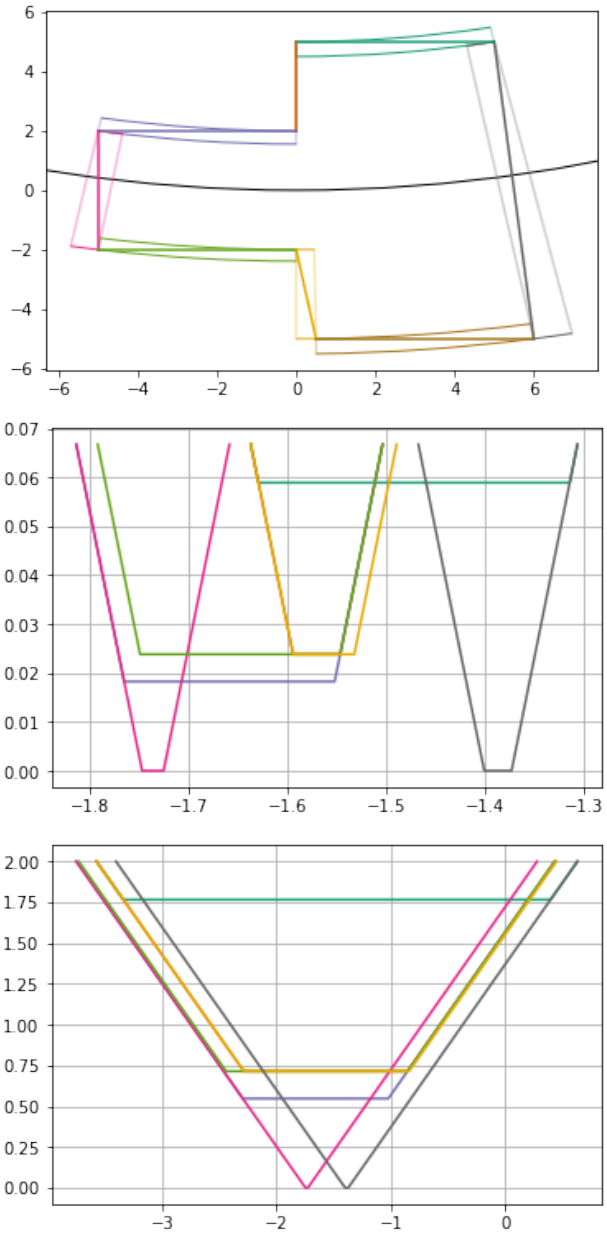


Figure 2.10.: Polygon segments with their circular bounding boxes (top) and the corresponding bounds in the circular diagram for a very tall label (middle) and a very long label (bottom). The tall label is constrained by the cyan segment. We can actually move it a little to left or to the right. The long label is constrained by the pink and the gray segments.

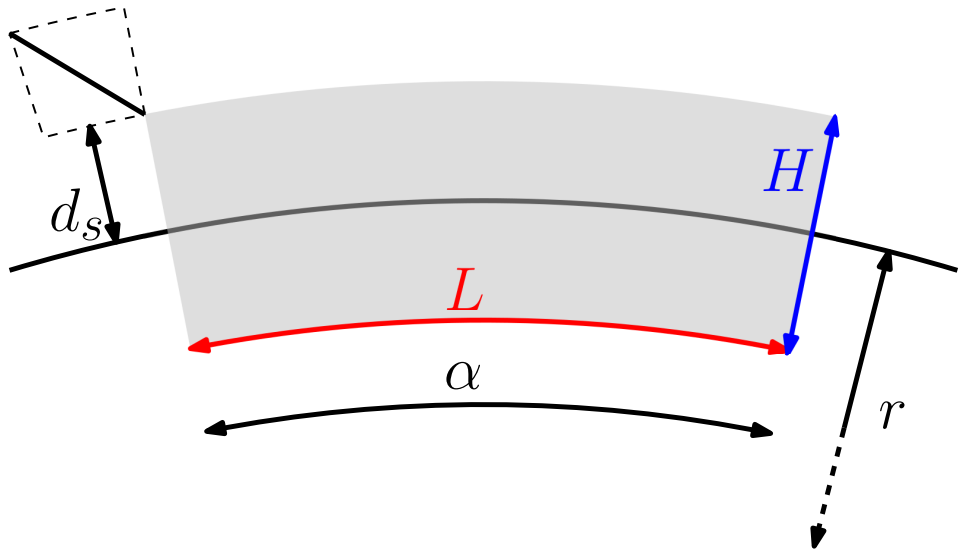


Figure 2.11.: The label can start to grow, when its corner touches the corner of the segment's bounding box.

Figure 2.9.

We now construct all those "wedges" from the segments and find the highest point, which is below each of the wedges. This point describes the angle where the label width is maximal. Because the label-aspect is fixed, this means that the label size is maximized. So this yields the optimal label placement.

To find this point, we first consider the complete circle from 0 up to $2 \cdot \pi$. We then consider each wedge, from lowest to highest, and restrict the possible placements. When there are no more valid placements left, we return the highest point seen. A simple example instance is depicted in Figure 2.10.

The active wedges can be organized in a segment tree. For any height, the set of wedges looks like a set of segments. When going up, these segments grow. When two wedges intersect, we can merge the associated segments.

We can enumerate the wedges with a heap, to access them from lowest to highest. Because we can stop the computation, when there are no more valid placements left, we only consider a small amount of segments. Considering the example in Figure 2.10 with the long label, we would only inspect the pink and gray wedges before returning the optimal placement.

Wedge Computation

In this chapter, we will go through the math needed to actually compute the wedges. For any mathematical symbols, please consider Figure 2.11 as a

reference. Furthermore A denotes the aspect of the label.

First let's derive the relationship between the height of the label (H), its width (L) and the spanned angle (α) for a given circle with radius r.

By definition the following holds:

$$H = L \cdot A \tag{2.1}$$

Furthermore we can easily derive:

$$L = (r - H/2) \cdot \alpha \tag{2.2}$$

For a given segment s, let d_s denote the minimal distance of the segment to the circle. If the height H of our label is less than $2 \cdot d_s$ the segment does not interfere with the label placement. If it is greater, we can compute the spanned angular range α by plugging (2.2) into (2.1) and solving for α. The center of the label needs to be at least $\alpha/2$ from the segment.

With the special case of $H = 2 \cdot d_s$ we can compute exactly the placement of the label for which the wedge transitions from one linear function to the next. Coming from the left the label shrinks, until it can fit below the segment. It then slides along without changing size. Finally, its size can increase once again if its far enough to the right.

The following statements are equivalent:

- The height of the label is maximized.
- The length of the label is maximized.
- The area of the label is maximized.
- The angular extent of the label is maximized.

It's easiest to describe the wedges in terms of maximum angular extent.

Let α_{d_s} be the alpha value, such that H equals $2 \cdot d_s$. Also, let β_1 and β_2 be the angles between the circle center and the segment's endpoints. Finally let α_l denote the angle on which the label center is placed.

If $\alpha_l > \beta_2 + \alpha_{d_s}/2$ the maximum possible label extent is $\alpha_{d_s} + 2 \cdot (\alpha_l - (\beta_2 + \alpha_{d_s}/2))$

If $\alpha_l < \beta_1 - \alpha_{d_s}/2$ the maximum possible label extent is $\alpha_{d_s} + 2 \cdot ((\beta_1 - \alpha_{d_s}/2) - \alpha_l)$

If α_l falls within those bound the value is exactly α_{d_s}.

This yields 3 piecewise linear functions for the wedges.

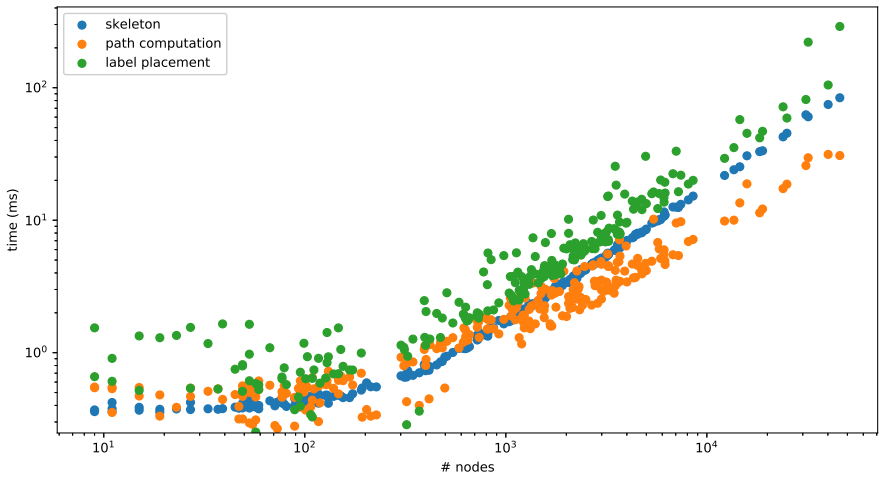


Figure 2.12.: Running time of the main operations on different sized data-sets.

2.5. Evaluation

2.5.1. Implementation

We implemented our algorithm in `C++`. For the geometric operations we relied on the CGAL Library [51]. Graph searches were done with the help of the Boost Graph Library [10]. The code was compiled with gcc 8.3. The experiments were run on a standard desktop computer with a Intel Xeon E3-1225v3 CPU with 3.20GHz.

2.5.2. Benchmarks

We evaluated our code on a data set of countries. The data-set was obtained from [45]. For some of the countries multiple polygons were provided to accommodate small islands belonging to those countries. The complete polygon set consisted of $687,820$ nodes. We computed all labels in about 4 seconds. The running times for the main operations are depicted in Figure 2.12.

Path selection

One of our main contributions is the refined path selection. In this experiment, we compare our path selection against what Barrault proposed (picking the 50 longest paths).

instance	# nodes	skeleton	paths	label	total
Antarctica	31919	60.348	29.610	220.830	310.788
Argentina	8543	15.210	7.163	19.988	42.361
Australia	18925	33.434	12.156	46.900	92.490
Austria	2195	3.746	3.373	5.019	12.138
Belgium	1307	2.236	2.120	3.282	7.638
Brazil	18315	32.933	11.374	41.867	86.174
Canada	40125	74.711	31.376	104.849	210.936
China	23943	42.578	17.325	71.771	131.674
Denmark	1943	3.423	4.115	6.154	13.692
CzechRepublic	1847	3.032	2.898	4.270	10.200
Egypt	3593	6.525	3.358	10.928	20.811
Finland	4411	7.824	3.696	13.121	24.641
France	6199	10.983	4.631	16.044	31.658
Germany	4755	8.145	3.935	12.016	24.096
Greece	5431	9.618	10.184	16.340	36.142
Greenland	31101	62.488	25.852	81.491	169.831
Iceland	6127	11.545	5.974	13.684	31.203
Hungary	1701	2.766	2.941	3.823	9.530
India	13629	24.040	10.008	35.318	69.366
Ireland	4527	7.985	5.150	13.734	26.869
Italy	3903	6.753	4.129	15.740	26.622
Japan	4959	8.446	4.728	30.376	43.550
Kyrgyzstan	2257	3.638	4.139	5.537	13.314
Liechtenstein	59	0.400	0.527	0.570	1.497
Luxembourg	395	0.762	1.124	1.305	3.191
Mexico	12249	21.770	9.854	29.309	60.933
Norway	15827	30.643	18.811	45.282	94.736
Poland	2673	4.666	2.287	6.910	13.863
Portugal	1761	2.995	2.868	4.592	10.455
Russia	45847	84.000	30.770	290.066	404.836
Spain	4317	7.440	3.671	12.110	23.221
SriLanka	1551	2.755	2.098	3.749	8.602
Sweden	6781	12.581	5.489	22.430	40.500
Switzerland	1501	2.578	3.163	3.737	9.478
Turkey	6113	10.692	5.528	14.736	30.956
UnitedKingdom	7423	13.186	9.751	21.869	44.806
UnitedStatesofAmerica	25017	45.336	18.710	59.059	123.105
Vatican	15	0.368	0.546	1.338	2.252

Table 2.1.: Running times for some of the instances. Times are in ms.

instance	#vertices	time_clear	time_bf	size_rel	time_rel
Deutschland	141338	1.010	107.749	0.353	106.6
Mecklenburg-Vorpommern	10154	0.054	0.335	0.476	6.1
Bayern	117003	0.826	71.474	0.139	86.5
Brandenburg	19730	0.114	1.091	1.837	9.5
Schleswig-Holstein	5632	0.030	0.135	0.662	4.4
Baden-Württemberg	34679	0.202	3.314	0.425	16.4
Sachsen	37967	0.235	4.526	0.228	19.2
Hessen	21797	0.138	1.473	0.482	10.6
Berlin	5890	0.032	0.179	0.646	5.4
Sachsen-Anhalt	13749	0.080	0.517	0.263	6.4
Bremen	2459	0.014	0.053	0.373	3.7
Thüringen	29332	0.166	2.539	1.029	15.2
Nordrhein-Westfalen	39475	0.229	4.838	0.489	21.0
Hamburg	2987	0.016	0.054	0.663	3.3
Saarland	6246	0.034	0.166	1.040	4.7
Rheinland-Pfalz	24393	0.133	1.433	0.942	10.7
Niedersachsen	28789	0.173	2.312	0.863	13.3

Table 2.2.: Comparing two path selection strategies. Choosing fewer paths according to the clearing values of the medial-axis results in smaller running times. Selecting 50 longest paths in a brute-force way takes longer and gives worse quality for most instances.

For this comparison, we picked up to 7 paths with our scheme. Because of the lower number of paths the algorithm terminates earlier, because less paths are considered, but also the quality of the results improved. See Table 2.2 for a comparison of the two strategies.

Consider the instance "Sachsen". This instance consists of 37967 vertices. When selecting paths with respect to the clearing around the skeleton, the whole algorithm takes 235ms to terminate. When selecting 50 paths in a brute-force-manner, the algorithm takes 4.5s in total. Even though the brute-force variant inspects more paths, the computed label is only roughly one fifth times the size of the label computed with our scheme. Even though it took 19.2 times longer to compute the result.

The brute-force approach beats our stategy three times on the 16 datasets. Two of those labels are very close (less than 5%) in size. Only in the case of Brandenburg, the brute-force method computes a label nearly twice the size of our label.

Figure 2.13.: Examples of computed labels.

2.6. Conclusions

There is not a lot of other work on area labeling.

Early works for labeling areas include the work of Roessel ([47]). In this work, he proposes a scheme to compute a maximal, axis-aligned rectangular box inside a given polygon. This is achieved by partitioning the polygon into horizontal stripes and then scanning through the segments of the polygon. This algorithm is more primitive than our algorithm as it only considers horizontal labels in a rectangular box.

Another approach for labeling general geometries of maps simultaneously was proposed by Edmondson et al ([21]). They first propose valid labelings for the different objects (point-, line- and area-features). For every object, a set of candidate labels is computed. Given this set of labels, they perform simulated annealing to find a solution which minimizes some penalty value. The penalty considers overlapping between labels, between labels and map objects, as well as the quality of the individual object labelings. Computing possible positions for area-labels is rather primitive. They compute the so called "inset polygon", which consists of all points inside the polygon, where the label could be placed. They then sample a set of those positions as candidates. This algorithm differs from our approach because it computes positions for multiple labels at once. If we are only considering the placement of area-labels, we see the same shortcomings as in Roessel's algorithm.

In [27] Goethem et al propose an algorithm to label groups of islands. They consider straight line labels as well as circular labels. The label types are further differentiated whether they overlap the island group or not. The goal is to find a labeling, which minimizes the maximum minimal distance between the label and any island. To compute the actual labeling, they use point-line duality and computation of the minimum-width annulus. The algorithm is rather involved and has (in the case of circular labels) quadratic running time.

When labeling polygonal areas, the work done by Barrault ([3]) is still one of most recent contributions.

With our approach, we have improved the most obvious shortcomings of Barrault's algorithm. Our improved path selection has been shown to improve the size of the computed labels, even though we inspect less paths. Furthermore, we improved the placement of the label along the candidate arcs. Instead of trying any valid position in a brute-force manner (and testing the whole polygon against all considered placements), we scan the segments of the polygon once and derive the optimal placement.

Future Work

As discussed in the beginning of the section, there are a lot of degrees of freedom to formulate the labeling problem. Instead of putting the labels along circular arcs, we could consider other shapes. For some instances, a circular arc is not a good fit. So other shapes can help to provide a better integration of the label inside the polygon. But one has to make sure that readability of the resulting labeling does not suffer.

Furthermore, we have seen, that the running times depend heavily on the size of the input polygon. We could use some sort of simplification to reduce the complexity of the input polygon. This could greatly reduce the size of the input, without really affecting the results. In the context of our proposed scheme, we would have to make sure to keep the maximum length of the polygon segments small enough to still get a good approximation of the medial axis.

Part II.

Route Planning

3. Stalling Traces

There are lots of services which compute the quickest paths to drive a car from some position to a destination. Such services can be hosted online and compute the paths on a server or the services can be downloaded to a mobile device and do the computation offline. Figure 3.1 shows some examples for such services.

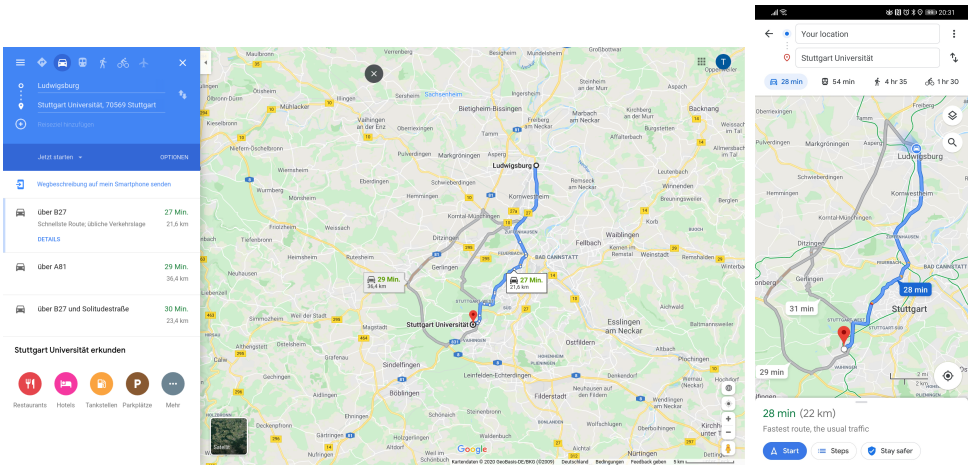
In both cases fast and efficient algorithms are needed. The online services have to serve many requests simultaneously without too much of a delay. Offline services on the other hand have to make use of the limited resources of mobile devices efficiently.

Using plain dijkstra's algorithm can take up to multiple seconds to find a route in a country-sized network.

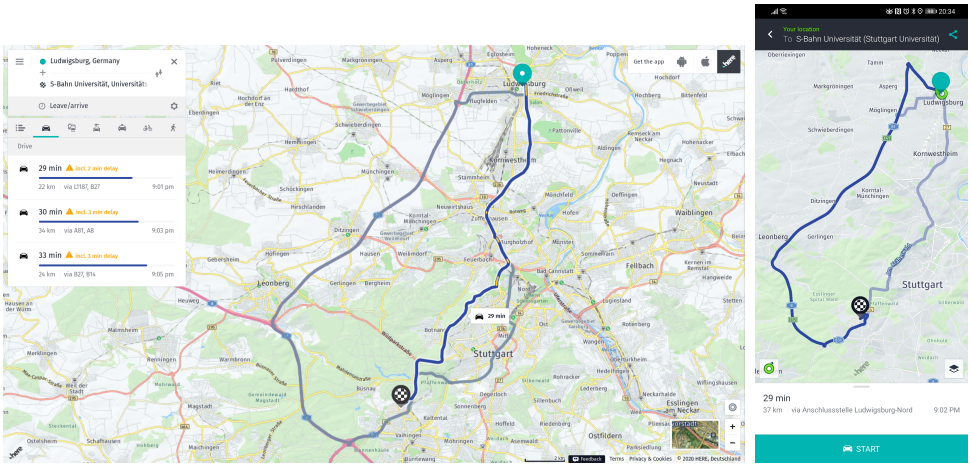
In this section we propose a preprocessing-step which accelerates a well-known routing-scheme by around 30%. Even though the preprocessing step takes a good amount of time to compute, the space overhead is reasonably small.

3.1. General Problem

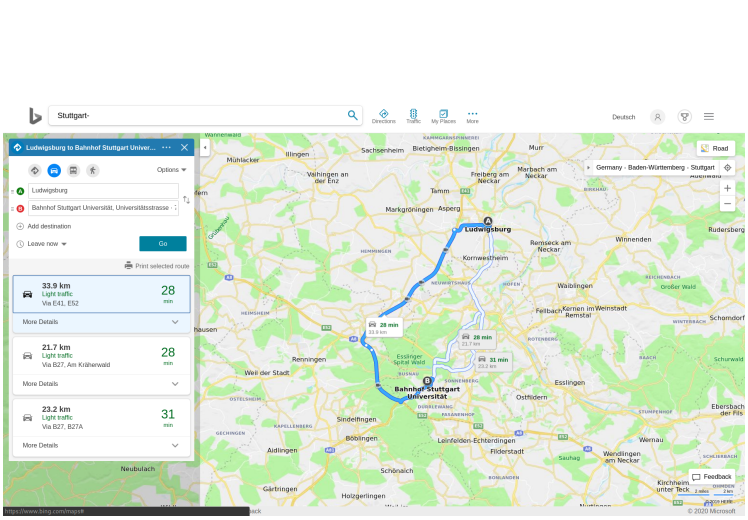
While the problem of computing shortest paths in general graphs with non-negative edge weights seems to have been well understood already decades ago, the last 10–15 years have seen tremendous progress when it comes to the specific problem of efficiently computing shortest paths in *real-world road networks*. Here the main idea is to spend some time in a preprocessing step where auxiliary information about the network is computed and stored, such that subsequent queries can be answered much faster than via standard Dijkstra's algorithm. One might classify most of the employed techniques into two classes: ones that are based on *pruned graph search* and such that are based on *distance lookups*. Most approaches fall into the former class, e.g., reach-based methods [32], [29], highway hierarchies [48], arc-flags-based methods [7], or contraction hierarchies (CH) [26]. Here, basically Dijkstra's algorithm is given a hand to ignore some vertices or edges during the graph search. The achievable speed-up compared to plain Dijkstra's algorithm ranges from one magnitude ([32]) up to three orders of magnitudes ([26]). In practice, this means that a query on a country-sized network like that of Germany (around



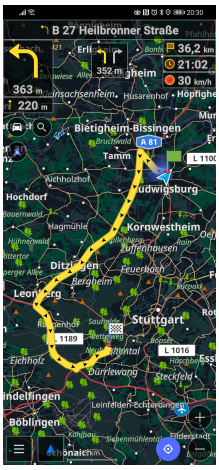
(a) The Google Maps Web- and Mobile- Routing-Service



(b) The Here Maps Web- and Mobile- Routing-Service



(c) The Bing Maps Web-Routing-Service



(d) The OSMand Mobile-Routing-App

Figure 3.1.: Examples of Routing-Services

20 million nodes) can be answered in less than a *millisecond* compared to few seconds of Dijkstra's algorithm. While these methods directly yield the actual shortest path, the latter class is primarily concerned with the computation of the (exact) distance between given source and target queries – recovering the actual path often requires some additional effort. Examples for such distance-lookup-based methods are transit nodes [6], [4] and hub labels [2]. They allow for the answering of *distance queries* another one or two orders of magnitudes faster.

In spite of their inferior query times, the methods based on pruned graph search are more popular in practice, because most of the time the actual shortest paths are in fact needed, and the methods based on distance lookups typically incur quite a considerable space overhead. For example, for a network of around 20 million nodes, the hub labeling scheme [2] requires to store for each node in the order of hundreds distance labels to allow for quick query answering. Hence the space consumption of the precomputed auxiliary information by far exceeds the space consumption of the original graph itself. For most methods based on pruned graph search, the space consumption of the precomputed auxiliary information is very moderate compared to the original graph itself. See [5] for a comprehensive survey on the topic.

3.2. Related Work

3.2.1. Contraction Hierarchies

The contraction hierarchies approach [26] computes an overlay graph in which so-called shortcut edges span large sections of the shortest path. This reduces the hop length of optimal paths and therefore allows a variant of Dijkstra's algorithm to answer queries more efficiently.

The preprocessing is based on the so-called *node contraction* operation. Here, a node v as well as its adjacent edges are removed from the graph. In order not to affect shortest path distances between the remaining nodes, shortcut edges are inserted between all neighbors u, w of v, if and only if uvw was the only shortest path (which can easily be checked via a Dijkstra run). The cost of the new shortcut edge (u, w) is set to the summed costs of (u, v) and (v, w). In the preprocessing phase all nodes are contracted one-by-one in some order. The rank of the node in this contraction order is also called the *level* of the node.

After having contracted all nodes, a new graph $G^+(V, E^+)$ is constructed, containing all original edges of G as well as all shortcuts that were inserted in the contraction process. An edge $e = (v, w)$ – original or shortcut – is

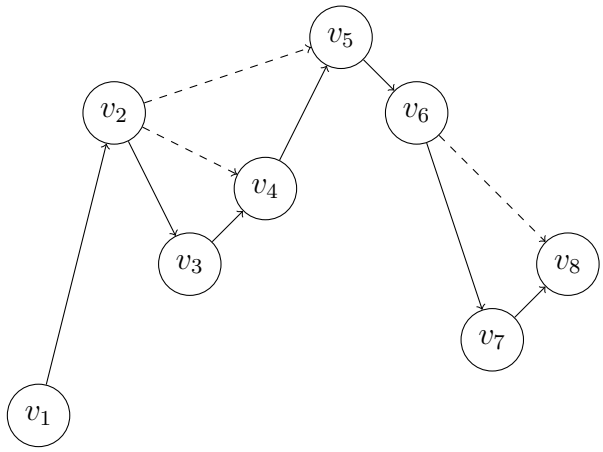


Figure 3.2.: Augmentation of shortcuts along a shortest path.

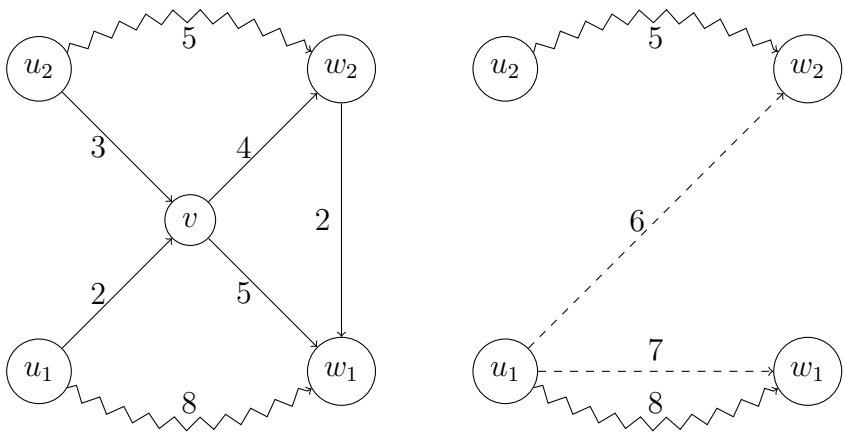


Figure 3.3.: Shortcut creation on node removal.

called upwards, if the level of v is smaller than the level of w, and downwards otherwise. By construction, the following property holds: For every pair of nodes $s, t \in V$, there exists a shortest path in G^+, which first only consist of upwards edges, and then exclusively of downwards edges. This property allows to search for the optimal path with a bidirectional Dijkstra only considering upwards edges in the search starting at s, and only downwards edges in the reverse search starting in t. This reduces the search space significantly and allows for answering of shortest path queries within the *milliseconds* range compared to *seconds* on a country-sized road network.

3.2.2. Hub Labels

Hub Labeling is a scheme to answer shortest path distance queries which differs fundamentally from graph search based methods. Here the idea is to compute for every $v \in V$ a *label* $L(v)$ such that for given $s, t \in V$ the distance between s and t can be determined by just inspecting the labels $L(s)$ and $L(t)$. All the labels are determined in a preprocessing step (based on the graph G), later on, the graph G can even be thrown away.

There have been different approaches to compute such labels (even in theory); we will be concerned with labels that work well for road networks and are based on CH again, following the ideas in [2]. To be more concrete, the labels we are constructing have the following form:

$$L(v) = \{(w, d(v, w)) : w \in H(v)\}$$

Here we call $H(v)$ a set of *hubs* – important nodes – for v. The hubs should be chosen such that for any s and t, the shortest path from s to t intersects $L(s) \cap L(t)$.

If such label sets could be computed, the computation of the shortest path distance between s and t boils down to determining the node $w \in L(s) \cap L(t)$ minimizing the summed distance. If the labels $L(.)$ are stored lexicographically sorted, this can be done in a very cache-efficient manner in time $O(|L(s)| + |L(t)|)$.

Knowing about CH, there is a natural way of computing such labels: simply run an upward Dijkstra from each node v and let the label $L(v)$ be the settled nodes with their respective distances. Clearly, this yields valid labels since CH answers queries exactly. The drawback is that the space requirement is quite large; depending on the metric and the CH construction, one can expect labels consisting of several hundreds to thousands node-distance pairs. It turns out, though, that many of the labels created in such a manner are useless as they do not represent shortest-path distance (as we restricted ourselves to a search in the upgraph only); pruning out those reduces the number of labels by a factor of 4. A source target distance query can then be answered in the *microseconds* range.

The computation of hub labels can be made more efficient than simply scanning the upwards-search-spaces. To reduce the amount of graph-searches we have to do, we observe the following: The upward-search space of a node is the union of the upward search-spaces of its upward neighbors. This means we can compute hub labels from highest to lowest level. For every node we simply merge the computed labels of its up neighbors. In this merging step we

need to remove duplicates as nodes can occur in more than just one neighbor's search-space.

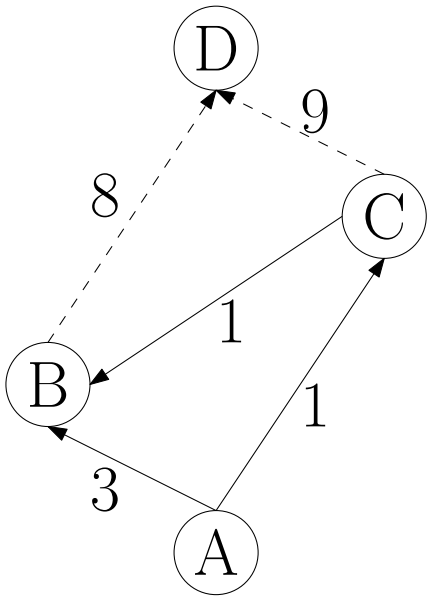
Furthermore we can filter out incorrect distances as follows: After computing the hub labels for a given node, we know that we also have correct hub labels for all the nodes occurring in this node's label, as they are higher level. We can then use the hub labels themselves to check, whether the recorded distance is correct. Incorrect entries are removed from the label.

Example

Consider the graph on the right.

When we compute the hub labels for node A we have already computed the labels for the nodes B and C.

The relevant labels are the forward labels of nodes B and C (B_f and C_f), and the reverse label of node B (B_r):
$B_f = [(B, 0), (D, 8)]$
$B_r = [(B, 0), (C, 1)]$
$C_f = [(C, 0), (D, 9)]$

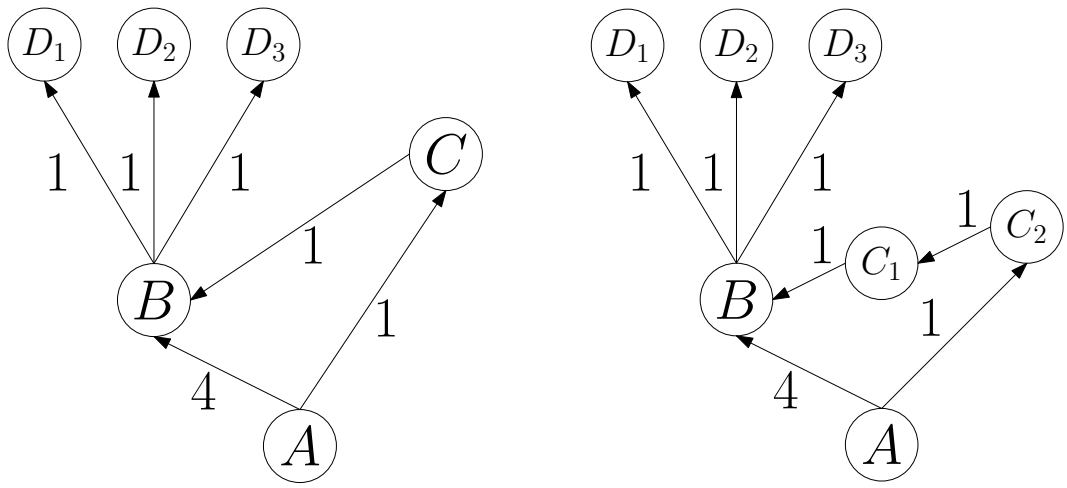


We compute the labels of A separately for paths going over B and C:
$AB_f = [(A, 0), (B, 0 + 3), (D, 8 + 3)]$
$AC_f = [(A, 0), (C, 0 + 1), (D, 9 + 1)]$

Now we can merge both labels:
$\tilde{A}_f = [(A, 0), (B, 3), (D, 10), (D, 11)]$

We only keep the smallest entry for each node:
$\hat{A}_f = [(A, 0), (B, 3), (C, 1), (D, 10)]$

We check all distances in the temporary label, with the label itself (bootstrapping). When looking at label B_r and $\hat{A}_f$, we see, that the distance from A to B is 2, not 3. Therefore we remove the entry from the label and get:
$A_f = [(A, 0), (C, 1), (D, 10)]$



(a) Stall-on-demand can identify the incorrect label of node B (4 instead of 2) and does not need to explore nodes D_i.

(b) Stall-on-demand cannot identify the incorrect label of node B (4 instead of 3) and therefore explores nodes D_i.

Figure 3.4.: The limits of 1-hop stall-on-demand.

3.3. Stalling Traces

3.3.1. Idea

As we already observed in the construction of hub labels, the exploration of the upgraph during the CH query phase might visit nodes with non-shortest path distances. Obviously, none of the nodes settled with non-shortest path distance are relevant for answering a shortest path query, yet they contribute to the query time. In the original CH paper [26] a technique, which they call *stall-on-demand*, was suggested which identifies some of the nodes with non-shortest path distances in the exploration of the upgraphs. Note, that there is a trade-off between the decrease in query time due to the reduced number of nodes to consider and the effort to identify nodes with non-shortest path distance.

In its simplest form, the stall-on-demand strategy from [26] works as follows: Consider the upgraph search from the source s (the reverse upgraph search from the target t works analogously). When a node v is pulled from the priority queue with distance label $d(v)$, one inspects all *incoming* neighbors w with (v, w) and $level(w) > level(v)$. Clearly, if $d(w) + c(w, v) < d(v)$, $d(v)$ cannot be the shortest path distance from s to v and hence the exploration (in particular relaxation of outgoing edges) of v can be 'stalled'. Of course,

this procedure does not necessarily identify all nodes with non-shortest path distances, yet it is easy to implement and still prunes the search considerably. More involved stall-on-demand strategies explore a larger neighborhood to conclude for even more nodes to bear non-shortest-path distances. Yet, the additional effort at query time is not rewarded by a respective more reduced search space. See Figure 3.4 for an example of how stall-on-demand can reduce the number of nodes explored in the upward-search.

The contribution of this work is the idea of precomputing perfect stalling decisions. In that way, we can benefit from a maximally reduced search space during the CH search without incurring a runtime penalty for performing a stall-on-demand computation at query time. It turns out that this can be done with moderate space overhead.

The first idea that comes to mind is to store for each node of the upgraph of s (and analogously for t) a bit whether it is reachable within the upgraph with shortest path distance. There are some disadvantages of this idea: First, we might store information for nodes in the upgraph that would never be encountered during the search because all immediate predecessors have already been stalled. Second, since the desired information varies for different sources s, we have to store for each s and each v in the upgraph of s whether v is reachable from s on a shortest path within the upgraph. While the actual information is only a single bit, storing the identity of each v is quite costly, e.g., a node ID is typically 64 bits. Storing several hundreds or thousands of such items results in several kilobytes additional memory for each node in the graph.

3.3.2. Computation

Note that if CH-based hub labels are available, the decision whether a node v just pulled from the priority queue with distance label $d(v)$ in the upgraph search from s can be made by using hub labels to look up the correct shortest path distance and comparing with d. Clearly, these decisions are perfect in a sense that we stall exactly those nodes that are not reachable on a shortest path within the upgraph. Yet, the requirement of having precomputed hub labels in the background just to speed up CH queries is prohibitive in practice due to the considerable space consumption.

Now the main idea to enjoy the benefits of stalling at query time without the runtime penalty of stall-on-demand or the space overhead of hub labels is to simulate the upgraph searches with perfect stalling during a preprocessing step (with the help of precomputed CH-based hub labels) and only record the respective decisions as a bit stream – which we call *stalling trace* – in the

order they are taken. For each node v we store two stalling traces – one for the upgraph search where v acts as source, one for the reverse upgraph search where v acts as target. Apart from representing perfect stalling decisions, this approach not necessarily requires a bit for every node of the upgraph of v; if for a node w in the upgraph of v, all immediate predecessors are not reachable on shortest paths from v, w will never be pulled from the priority queue (due to the perfect stalling decisions) and hence does not require a bit in the stalling trace.

In summary, the preprocessing phase of our method looks as follows:

```
PREPROCESSING(G)
1. Construct CH
2. Construct CH-based hub labels HL
3. for each node:
   a. simulate upgraph searches
   b. store stalling traces
4. discard HL and only keep CH and stalling traces
```

At query time we simply run an ordinary CH-query but use the stalling traces to have perfect stalling decisions during the exploration of the upgraphs. Algorithms 1 and 2 illustrate the precomputation of a trace as well as the query for an unidirectional dijkstra run. In the bi-directional case we simply use the traces of the source as well as the target node.

We will see in the next section that this strategy pays off. Queries are considerably accelerated without incurring a major space overhead compared to pure CH representation.

Lightweight Construction

Most of the memory consumption comes from computing and storing the hub labels for the whole graph. Even though the hub labels are only an intermediate step for computing the stalling traces they consume most of the memory throughout the process. So the question arises, whether we can compute the traces without computing all hub labels at once.

Let us look into the possibility of computing stalling traces for a set of nodes. To obtain the complete bit-trace, we have to consider all of the upgraphs originating from any of the nodes. Furthermore we need to have hub labels available for all of those nodes in any of the upgraphs, so we can make the decision whether a distance is correct or not. As we need to have the forward as well as the backward hub labels available, we can compute the

```
Data: Node s, UpGraph G
Result: Boolean Vector "trace"
trace = Vector[Bool];
distance = Vector[Integer](default=INF);
pq = MinHeap;
pq.push(0, s);
distance[s] = 0;
while pq not empty do
    settle next node from pq;
    if distance is correct then
        trace.push(TRUE);
        relax outgoing edges;
    else
        trace.push(FALSE);
    end
end
```

Algorithm 1: Computing the trace for a single node. The decision whether a distance is correct is made via precomputed hub labels.

```
Data: Node s, UpGraph G, Trace t
Result: Distance Vector "distance"
distance = Vector[Integer](default=INF);
pq = MinHeap;
pq.push(0, s);
distance[s] = 0;
while pq not empty do
    settle next node from pq;
    if t.next() == TRUE then
        relax outgoing edges;
    end
end
```

Algorithm 2: Using the trace during upgraph search.

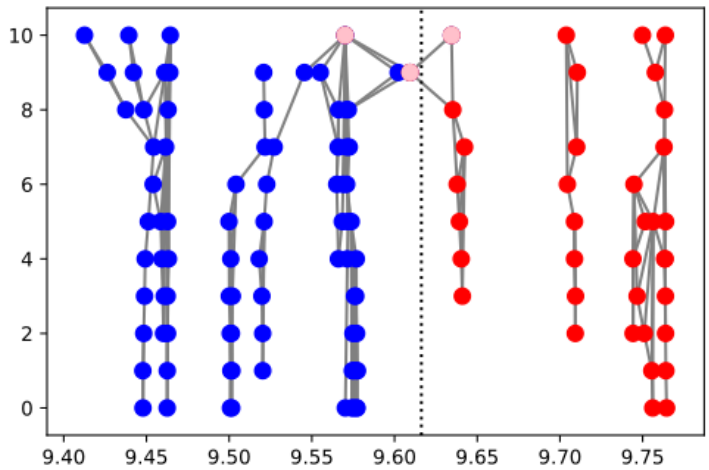


Figure 3.5.: The upward search-spaces of a graph-partitioning may overlap (pink nodes)

"upwards-closure" for a set of nodes: Namely all the nodes in the up-graph, *disregarding* the direction of the edges.

If we chose the subset of nodes in such a way that most of the nodes in the respective up-graphs are within the subset, we can compute stalling-traces for a partition of the graph with minimal overhead. Figure 3.5 illustrates how the upward-search spaces of two cells might overlap.

Simply using a grid to partition the nodes suffices. Using a 10-by-10-grid the overhead turns out to be around 1%. But memory consumption for the hublabel computation reduces to around 1%, because we can discard the labels of a finished grid cell.

Even though some labels are computed multiple times in the lightweight scheme, running times are virtually the same. This stems from the much smaller memory footprint.

Table 3.1.: Data sets for benchmarking

	STGT	BW	GER
nodes	1.16M	3.67M	24.61M
edges (original + shortcuts)	4.32M	13.76M	91.61M
CH shortcuts	1.97M	6.37M	41.82M
HL avg. label size	81.2	109.5	236.28

Table 3.2.: Running times for precomputation (in hh:mm:ss)

	STGT	BW	GER
hub labels	00:02:04	00:10:44	04:52:21
traces	00:01:16	00:07:13	02:40:43
total	00:03:21	00:17:57	07:33:05

3.4. Evaluation

In the following we report on our experiments with the improved query scheme for contraction hierarchies. All implementations were compiled using `g++ 7.3.0` and executed on a single core of a Ubuntu Linux 18.04 system with an Intel Xeon E3-1225v3, 3.2GHz and 32GB of RAM.

Data Sets, CH and HL precomputation

We consider three data sets that were extracted from the OpenStreetMap project [52]. The datasets we considered are:

- STGT: Administrative area of the city Stuttgart
- BW: The State Baden-Württemberg
- GER: The Country Germany

We used travel time as a metric. For all three data sets contraction hierarchies as well as CH-based hub labels were computed using the standard approaches in [26] and [2]. See Table 3.1 for the resulting characteristics of our data sets. Note that both graphs and metrics differ from the ones used in [26] or [5], hence in particular the number of shortcuts as well as the average label sizes differ.

Table 3.2 shows the running times for computing the stalling-traces. Table 3.3 summarizes the result of our preprocessing step. Note that our implementation is single-threaded; we expect considerable speedup by parallelization

since any number of vertices can be processed in parallel. Observe that we require less than the graph representation itself of additional memory to store the stalling traces.

Queries

Now let us analyze the effect on query times. We report on average query times for random source target queries for plain Dijkstra (**Dijk**), plain CH without stall-on-demand (**CH**), CH with standard stall-on-demand (**CHso**), and CH with perfect stalling (**CHps**), stating both number of settled nodes as well as the actual query times. All numbers are averaged over 100,000 queries except for plain Dijkstra, where we only made 100 queries due to time constraints. See Table 3.4 for the results.

As to be expected all CH variants are at least a factor of 1,000 more efficient than plain Dijkstra, both in terms of number of settled nodes as well as actual query time (both max and average). Comparing the CH variants for the largest graph, the CHso variant on average settles only around 35% of the nodes, the query times are around 39% of the standard CH query. Somewhat to our surprise, CHps only settles slightly less nodes than CHso, yet the average query times improve to just 25% of the standard CH query. So one-hop stall-on-demand is almost perfect in detecting nodes with non-shortest-path distances; the additional improvement of CHps in query time is due to not having to perform stall-on-demand and simply use the precomputed stalling trace. We also observe that both stalling variants become more effective the larger the graphs get. The respective maximum values behave similar to the averages.

Table 3.3.: Stalling Trace Construction

	STGT	BW	GER
avg. trace length (Bits/node)	130.2	202.6	489.4
max. trace length (Bits/node)	233	332	851
total trace space (MBytes)	36.0	177.3	2871.5
Graph size (incl. CH) (MBytes)	182.5	580.9	3870.2

Table 3.4.: Query benchmarks: measuring average number of settled nodes as well as query times.

	STGT				BW				GER			
	Dijk	CH	CHso	CHps	Dijk	CH	CHso	CHps	Dijk	CH	CHso	CHps
# settled												
avg	627k	313	206	201	1.65M	586	321	311	12.9M	2212	782	761
max	1.14M	634	390	377	3.59M	1183	585	577	24.4M	4161	1388	1297
query-time												
avg in μs	155ms	85	66	47	439ms	200	124	83	4.4s	1159	457	301
max in μs	298ms	133	95	70	951ms	313	175	119	8.6s	1874	644	427

3.5. Conclusions

Precomputation has an important place in shortest-path computation. If lots of shortest path queries are to be answered the time to compute some additional information, to speed up subsequent queries, is well spend. The additional information could be hints to guide a dijkstra search, augmentations to the graph-structure to allow for a quicker traversal, or some other information extracted from the graph, which allows for distance computation without looking at the graph at all anymore.

Reach ([32]) aims at identifying which nodes are important for long-distance queries and which are not. The preprocessing computes a value for each node, which states whether the node is part of some long shortest path or not. When the dijkstra search has expanded enough from the source and target node, nodes with small reach-values can be discarded.

A different scheme for goal direction is Arc-Flags ([44]). The underlying graph is partitioned into parts of roughly equal sizes and with few edges interconnecting the components. For each edge a bit vector is computed. The i'th bit indicates whether this edge lies on a shortest path to the i'th component. During the query only edges which lie on a shortest path to the component of the target node are considered.

The performance of Arc-Flags is dependent on the size of the graph-partitioning. Small partitions introduce larger bit-vectors on each edge as well as increased pre-computation time. Larger partitions diminish its effect, because routing inside the target partition is still slow.

Both, Reach and Arc-Flags prune large parts of the graph from the dijkstra search-space. Our approach always benefits from the already much more reduced CH search-space, which is further pruned by the stalling-traces.

Hub labels ([2]) represent another class of speed-up technique. The preprocessing step computes a list of labels for each node. Computing distances between nodes can then be achieved without considering the underlying graph at all: The labels of both nodes are intersected and the minimum distance of any common nodes is returned. Computation of the hub labels can be done via contraction hierarchies. The upwards search space of a node can be used as a label.

While talking about hub labels there are also compressed hub labels ([17]), which also reduce the hefty memory footprint of normal hub labels. The compression makes use of the observation that nearby vertices share large parts of their labels. Labels are transformed into trees and common subtrees of different labels are only stored once. This space reduction is bought with worse running times than standard hub labels (about a factor of 5). They

are also a little more involved in terms of complexity for the pre-computation. The resulting data-structure is bigger than the graph itself.

Our scheme shares the goal of representing the minimal necessary search-space without storing too much data. The stalling-trace can be used to identify exactly those nodes with minimal distances in the upward CH-searches. Hub labels (with pruning) also store only the necessary information. The compression scheme merely aims at reducing the amount of data stored, which is still more than our approach. In total the compressed hub labels offer faster query times than our approach but use a more involved computation and use more storage.

Future Work

Providing good service is important for many map and routing providers. Accelerating queries plays an important role in two ways. If the service cannot answer queries in reasonable time users are repelled. On the other hand faster queries allow the service to answer more complex queries in reasonable time (e.g. evaluating multiple alternatives).

Contraction Hierarchies are a popular speed-up scheme for computing shortest paths in road network. And while providing incredible speed-ups already we always want to find ways to improve query times further.

Bibliography

[1] M.A. Abam et al. "Computing homotopic line simplification". In: *Computational Geometry* (2014).

[2] Ittai Abraham et al. "Hierarchical Hub Labelings for Shortest Paths". In: *Proc. of the 20th Annual European Symposia on Algorithms (ESA)*. 2012.

[3] M. Barrault. "A methodology for placement and evaluation of area map labels". In: *Computers, Environment and Urban Systems* (2001).

[4] H. Bast et al. "In Transit to Constant Time Shortest-Path Queries in Road Networks". In: *Proc. of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX)*. 2007.

[5] Hannah Bast et al. *Route Planning in Transportation Networks*. 2015. arXiv: `1504.05140 [cs.DS]`.

[6] Holger Bast et al. "Fast Routing in Road Networks with Transit Nodes". In: *Science* (2007).

[7] Reinhard Bauer and Daniel Delling. "SHARC: Fast and Robust Unidirectional Routing". In: *Proc. of the 10th Workshop on Algorithm Engineering and Experiments (ALENEX)*. 2008.

[8] Marc de Berg, Marc van Kreveld, and Stefan Schirra. "A new approach to subdivision simplification". In: *ACSM/ASPRS Annual Convention & Exposition Technical Papers*. 1995.

[9] Harry Blum. "A transformation for extracting new descriptions of shape". In: *Models for the perception of speech and visual form*. 1967.

[10] Boost. *Boost C++ Libraries*. 2019. URL: `http://www.boost.org`.

[11] Prosenjit Bose et al. "Area-preserving approximations of polygonal paths". In: *Journal of Discrete Algorithms* (2006).

[12] J. W. Brandt. "Convergence and Continuity Criteria for Discrete Approximations of the Continuous Planar Skeleton". In: *CVGIP: Image Understanding* (1994).

[13] Kevin Buchin, Wouter Meulemans, and Bettina Speckmann. "A New Method for Subdivision Simplification with Applications to Urban-area Generalization". In: *Proc. of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. 2011.

[14] W.S. Chan and F. Chin. "Approximation of polygonal curves with minimum number of line segments". In: *Proc. of the 3rd International Symposium on Algorithms and Computation*. 1992.

[15] Francis Chin and Cao An Wang. "Finding the Constrained Delaunay Triangulation and Constrained Voronoi Diagram of a Simple Polygon in Linear Time". In: *SIAM Journal on Computing* (1998).

[16] Thomas H. Cormen et al. *Introduction to Algorithms, Third Edition*. 2009.

[17] Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. "Hub Label Compression". In: *Proc. of the 12th International Symposium on Experimental Algorithms (SEA)*. 2013.

[18] Tamal K. Dey and Wulue Zhao. "Approximating the Medial Axis from the Voronoi Diagram with a ConvergenceGuarantee". In: *Algorithmica* (2004).

[19] 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". In: *Cartographica* (1973).

[20] Christopher Dyken, Morten Dæhlen, and Thomas Sevaldrud. "Simultaneous curve simplification". In: *Journal of geographical systems* (2009).

[21] Shawn Edmondson et al. "A General Cartographic Labeling Algorithm". In: *Cartographica* (1996).

[22] Regina Estkowski and Joseph S. B. Mitchell. "Simplifying a Polygonal Subdivision While Keeping It Simple". In: Proc. of the Seventeenth Annual Symposium on Computational Geometry (SoCG). 2001.

[23] Stefan Funke and Thomas Mendel. "Improved CH-Queries via Perfect Stall-On-Demand". In: *Proc. of the Special Event on Analysis of Experimental Algorithms (SEA^2)*. 2019.

[24] Stefan Funke et al. "Map Simplification with Topology Constraints: Exactly and in Practice". In: *Abstracts of 32nd European Workshop on Computational Geometry (EuroCG)*. 2016.

[25] Stefan Funke et al. "Map Simplification with Topology Constraints: Exactly and in Practice". In: *Proc. of the 19th Workshop on Algorithm Engineering and Experiments (ALENEX)*. 2017.

[26] Robert Geisberger et al. "Exact Routing in Large Road Networks Using Contraction Hierarchies". In: *Transportation Science* (2012).

[27] Arthur van Goethem, Marc J. van Kreveld, and Bettina Speckmann. "Circles in the Water: Towards Island Group Labeling". In: *Proc. of the 9th International Conference On Geographic Information Science (GI-Science)*. 2016.

[28] Arthur van Goethem et al. "Topologically Safe Curved Schematisation". In: *The Cartographic Journal* (2013).

[29] Andrew V. Goldberg, Haim Kaplan, and Renato F. Werneck. "Reach for A*: Efficient Point-to-Point Shortest Path Algorithms". In: *Proc. of the 8th Workshop on Algorithm Engineering and Experiments (ALENEX)*. 2006.

[30] Leonidas J. Guibas et al. "Approximating Polygons and Subdivisions with Minimum Link Paths". In: *Proc. of the 2nd International Symposium on Algorithms (ISA)*. 1991.

[31] LLC Gurobi Optimization. *Gurobi Optimizer Reference Manual*. URL: `http://www.gurobi.com`.

[32] Ronald J. Gutman. "Reach-Based Routing: A New Approach to Shortest Path Algorithms Optimized for Road Networks". In: *Proc. of the 6th Workshop on Algorithm Engineering and Experiments and the First Workshop on Analytic Algorithmics and Combinatorics (ALENEX/ANALCO)*. 2004.

[33] *Handbook of Computational Geometry*. 2000.

[34] John Hershberger and Jack Snoeyink. "Speeding Up the Douglas-Peucker Line-Simplification Algorithm". In: *Proc. of the 5th International Symposium on Spatial Data Handling (SDH)*. 1992.

[35] Hiroshi Imai and Masao Iri. "Polygonal Approximations of a Curve – Formulations and Algorithms". In: *Computational Morphology*. 1988.

[36] Eduard Imhof. "Positioning Names on Maps". In: *The American Cartographer* (1975).

[37] L.G. Khachiyan. "Polynomial algorithms in linear programming". In: *USSR Computational Mathematics and Mathematical Physics* (1980).

[38] Filip Krumpe and Thomas Mendel. *Computing Curved Area Labels in Near-Real Time*. 2020. arXiv: `2001.02938 [cs.HC]`.

[39] Der-Tsai Lee. "Medial Axis Transformation of a Planar Shape". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)* (1982).

[40] Michael McAllister and Jack Snoeyink. "Medial Axis Generalization of River Networks". In: *Cartography and Geographic Information Science* (2000).

[41] Natalia Mendel. "Dynamische Beschriftung von Gebietshierarchien - Entwicklung und Implementierung der Beschriftung von hierarchischen Gebietsunterteilungen". Bachelorthesis. 2018.

[42] Thomas Mendel. "Area-Preserving Subdivision Simplification with Topology Constraints: Exactly and in Practic". In: *Proc. of the 20th Workshop on Algorithm Engineering and Experiments (ALENEX)*. 2018.

[43] Wouter Meulemans, André van Renssen, and Bettina Speckmann. "Area-Preserving Subdivision Schematization". In: *Proc. of the 6th International Conference On Geographic Information Science (GIScience)*. 2010.

[44] Rolf H. Möhring et al. "Partitioning graphs to speedup Dijkstra's algorithm". In: *ACM Journal of Experimental Algorithmics* (2006).

[45] *Natural Earth*. URL: `https://www.naturalearthdata.com`.

[46] P. M. Van Der Poorten and Christopher B. Jones. "Characterisation and generalisation of cartographic lines using Delaunay triangulation". In: *International Journal of Geographical Information Science* (2002).

[47] Jan W. van Roessel. "An Algorithm for Locating Candidate Labeling Boxes Within a Polygon". In: *The American Cartographer* (1989).

[48] Peter Sanders and Dominik Schultes. "Engineering highway hierarchies". In: *ACM Journal of Experimental Algorithmics* (2012).

[49] Michel Schmitt. "Some examples of algorithms analysis in computational geometry by means of mathematical morphological techniques". In: *Proc. of the French Workshop on Geometry and Robotics (GeoRob)*. 1988.

[50] M. Shimrat. "Algorithm 112: Position of Point Relative to Polygon". In: *Communications of the ACM* (1962).

[51] The CGAL Project. *CGAL User and Reference Manual*. 2019. URL: `https://doc.cgal.org/4.14/Manual/packages.html`.

[52] *The OpenStreetMap Project*. 2014. URL: http://www.openstreetmap.org.

Erklärung:

Hiermit versichere ich, diese Arbeit selbständig verfasst und nur die angegebenen Quellen benutzt zu haben.

(author)