Institute of Formal Methods in Computer Science

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Over-the-Web Retrieval and Visualization of Massive Trajectory Sets

Lukas Baur

Course of Study:	Informatik
Examiner:	Prof. Dr. Stefan Funke
Supervisor:	M. Sc. Tobias Rupp
Commenced:	June 1, 2021
Completed:	December 1, 2021

Abstract

As the number of cost-effective GPS-supporting devices continues to increase tremendously, the number of recorded trajectories, i.e., measured sequences in time and space, explodes. The enormous potential of this data in terms of information retrieval data mining analysis of various kinds requires advanced storage and retrieval solutions. In [25], Funke et al. presented a data structure PATHFINDER (PF) based on a state-of-the-art speed-up technique for shortest path planning allowing for efficient trajectory compression and rather complex query answering. Since PF results are returned in compressed form by default and their complete decompression is only possible with the help of the internal data structure, a separation of a retrieval server and lightweight clients is nontrivial. Even more problematic is the amount of data produced by naively decompressing large queries: transferring the fully unpacked paths is neither meaningful nor feasible taking usual response waiting times into consideration.

This work closes the gap by presenting partially decompression and postprocessing methods based on aggregation, pruning, filtering, simplification, and batching in order to accomplish predefined use case goals. The presented methods are theoretically examined, tested on different real-world and synthetic datasets consisting of up to 10 000 000 trajectories, and critically reviewed with regard to applicability. In addition, for the special use case that relies exclusively on spatio-temporally independent queries, a PATHFINDER-based static tiling server was included which could be conceptually extended to online tiling for query answering as a prototype showed.

Kurzfassung

Aufgrund der ständig stark wachsenden Menge an kostengünsitgen GPS-fähigen Geräten steigt die Anzahl der aufgezeichneten zeitlich-örtlich verfolgten Pfadsequenzen, auch Trajektorien genannt, explosionsartig an. Um das enorme Potential dieser Daten in Hinblick auf Data-Mining Anwendungen nutzbar zu machen, werden ausgereifte Speicher-, Indizierungs- und Suchstrukturen benötigt. Zur effizienten Kompression und Suchanfragenbeantwortung stellten Funke et al. die Datenstruktur PATHFINDER (PF) vor, die eine moderne Kürzeste-Wege-Suche Technik einbindet, die selbst auf komplexen zeitlich-räumlichen Anfragen schnell Ergebnisse liefert. Da der PF die gefundenen Pfade in komprimierter Form zurückliefert, deren aber Dekomprimierung nur mithilfe der internen Datenstruktur wieder entpackt werden kann, ist eine Client-Server-Trennung nicht ohne Weiteres umsetzbar. Hinzu kommt, dass ein naives Entpacken aller komprimierten Pfade eine beträchtliche Ausgabegröße erreicht, die weder in vertretbarer Zeit auf Client-Seite empfangen und angezeigt werden kann noch interpretierbare Aussagen erlaubt.

Diese Arbeit schließt diese Lücke, indem durch partielle Dekomprimierung und erweiterten Export-Methoden, die auf Aggregation, Filterung, Vereinfachung und batchweises Übertragen beruhen, zuvor definierte Use Cases abgedeckt werden können. Die vorgestellten Verfahren werden theoretisch untersucht, auf realen und künstlich erzeugten Pfaddatensätzen mit Kardinalität bis zu zehn Millionen getestet und kritisch in Bezug auf Anwendbarkeit hinterfragt. Für den speziellen Anwendungsfall von zeitlich und örtlich uneingeschränkten Anfragen wurde ein PF-basierender Tiling-Server miteingebunden, der neben statischen Kartenausschnitten zu Online-Anfragen erweitert werden kann, wie es bereits prototypisch umgesetzt wurde.

Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

CH Contraction Hierarchy. 18

EBPF Edge-Based Pathfinder. 80

FMI Institute for Formal Methods of Computer Science. 28

GLP Graph Layer-Pruning. 65

HTTP Hypertext Transfer Protocol. 52

ID Identifier. 32

MOE maximum offset error. 49

OSM OpenStreetMap. 28

PF Pathfinder. 18

PLOU Plain Level Order Update. 53

PU-SOU Partially Unpacked Sequential Order Update. 56

R-SOU Rootedge Sequential Order Update. 56

REST representational state transfer. 49

RMOE relative maximum offset error. 50

ROU Randomized Order Update. 55

SGA Segment Graph Algorithm. 34

SOU Sequential Order Update. 51

SUA Sensitive CH-Unpacking Algorithm. 87

ULOU Unpacking Level Order Update. 57

WSUA Weighted Sensitive Unpacking Algorithm. 91

1 Introduction

With the advent of ubiquitous computing, a huge amount of positional data of high-quality is exchanged and captured on a daily basis. According to actual studies, almost 15 billion mobile devices exist and 1.38 billion smartphones were sold only in 2020 [54, 55]. The gigantic amount of GPS-ready devices makes it possible to record, centrally collect and analyze motion data of all kinds to draw conclusions about user behavior, traffic patterns, habits or anomalies, or to automate map creation and optimization. Thus, animal [10], vehicle [32] and ship [3] movements can be evaluated to name just a few well-known analysis examples. An important subfield represents the matching of trajectories with map data, which allows to express trajectories using paths on attributed uni- or bidirectional connection graphs. Although information is forfeited by matching continuous point sequences to nearest nodes in the graph, similar path sequences can be merged and edges can be reused. Trajectory analysis then narrows down to graph theoretical tasks.

To visualize, analyze and filter for trajectories matching certain criteria, search structures for quick retrievals were developed. In the context of shortest path computation, a layered graph structure called Contraction Hierarchies modifies the original graph by artificially adding edges to the graph, which at the same time drastically reduces the number of edges needed to describe a path on average. Storing a path using a Contraction Hierarchy comes down to iteratively *compressing* the input path by replacing edges with shortcuts. In the publication of [25], both concepts were combined resulting in an efficient retrieval data structure capable of returning matches within a few microseconds per reported trajectory. To visualize the found trajectory set, however, a reversion of the path-compression must be applied. Since a complete unpacking of the paths is quite time-consuming and the resulting paths have a very considerable size, such a basic export function is not sufficient for time-critical applications. To emphasize this issue by an example, when requesting the whole Saarland10M dataset naively, a json response having several gigabytes in size can be expected, which clearly falls aside the acceptable scope.

Objective

The goal of this work is to develop a suitable web application in JavaScript that supports interaction with the PATHFINDER data structure as presented in [25] using a suitable user interface. A client-server environment is created to separate end-users and retrieval runtime. In order to avoid long waiting times for the user, suitable aggregation and compression procedures are to be developed that allows interactive exploration.

Structure

This work is structured into two introductory and six research-driven chapters. A brief overview of each chapter is listed in the following.

To start with, a concise introduction is presented:

Chapter 1 – Introduction This work's implementation scope is sketched, definitions of use cases needed for evaluation are given and major related work is included.

Chapter 2 – Foundations Central terminology is determined, the key concepts reused in the following chapters, and predecessor projects are explained. To mention the most prominent concepts, Contraction Hierarchies and the PATHFINDER algorithms will become central, while the PATHFINDER^{web} serves as a groundwork this work builds on.

The following chapters resulted from and summarizes the author's research and are laid out chronologically according to the time they were addressed and implemented. This explains the transitions from one topic to another, since detected drawbacks of a specific method were tried to be remedied by subsequent approaches. Each chapter presenting a concrete approach closes with a concise discussion including exemplary scenarios and a use-case assessment based on measured values if available.

Chapter 3 – Segment Graph After introducing an exemplary indexing-method evaluated on real world data illustrating the concept of edge aggregation, the more advanced Segment Graph Algorithm is presented, theoretically analyzed and tested on real trajectory data.

Chapter 4 – Batched Transmission The theoretical concept of batched transmission and its evaluation scheme is pointed out. Various batching-methods are presented and competed against each other.

Chapter 5 – Graph Layer-Pruning This chapter's method filters result edges based on an importance criteria and substitutes edges of little importance by a heatmap.

Chapter 6 – Edgebased-PATHFINDER Slightly modifying the original PATHFINDER algorithm by early termination yields a slightly faster and locally bounded edge-based version which easily translates to the weighted case, but requires further postprocessing.

Chapter 7 – Tiling Since online-request answering exceeds acceptable waiting times for very large bounding boxes, the inclusion of tiling returns pre-caclulated responses at minimum time lag when requests are time- and space independent.

Chapter 8 – Future Work This work closes by listing the method's limitations and showing potential future extensions addressing those.

1.1 Contributions

The core of this work is represented by the PATHFINDER^{vis} implementation consisting of a comprehensive server infrastructure for trajectory retrieval combined with an extendable frontend counterpart to cover various use cases defined on differing zoom scales. The core implementation which originated from the PATHFINDER^{web} project, has been extended with an advanced setting management, path

importing and exporting functionalities, heatmap support and transparency filters for overlaying trajectories returned as a whole. To visualize shared path usages on a microscopic scale, the Segment Graph Algorithm provides a solid conversion from compressed paths into a graph consisting of edge-strokes only which supports basic offline trajectory exploration. Since a reduction of response time contributes to enhanced usability, both a class-based batching decoder-encoder framework was integrated into the client-server infrastructure and two basic transmission schemes have been implemented serving as proof of concept. A rather abstract analysis of batching mechanisms compares various transmission modes by measuring their performances on a real world dataset. Pruned filtering extracts graph-specific characteristic path strokes and replaces less significant but expensive lower level edges by an aggregated heatmap layer. Slightly modifying the initial PATHFINDER retrieval function yields an edge-based exporting scheme typically returning compact responses due to truncating. The overall client-server environment is supplemented by an additional PATHFINDER-powered tiling server which is both capable of answering static tile requests as well as interactive on-the-fly plotting.

1.2 Use Cases

As discussed earlier in the introduction, plotting all requested paths is not feasible in reasonable time. It is obvious that there is a correlation between the degree of visualization details and the network usage. Apart from that, not all use cases necessarily expect a full detailed result. If one requests to see all trajectories from a continental view, for instance, many details can be pruned without changing the visual representation too much. In the following chapters, this trade-offs between information quantity, visual quality and processing time will play a major role in the algorithmic design process.

Not all presented exporting strategies satisfy the users equally since the individual expectations differ considerably, hence to evaluate the upcoming methods, the following use cases are defined.

Definition 1 (uses cases)

1. **Microscopic Analysis**: The user's intention is to focus on a very limited spacial area, e.g. only a single city. Therefore, the depicted result should be of high resolution and low level of abstraction, potentially restricted by temporal constraints. Single trajectories should be distinguishable to allow for a comprehensible start-target research. A potential user might be a city planner wondering which places are frequently visited and or show noticeable drive-by potential for different week days or varied seasonal choices.
2. **Inter-City Analysis**: The main goals are the understanding of inter-city relationships and coarse movement structures. Temporal variability also plays a significant role in this case. Traffic researchers for example intend to detect route-anomalies or search for potential overloaded sections needed to be refined, e.g., by constructing bypasses. A list of potential applications has been published in [4].
3. **Macroscopic Analysis**: The user manly focuses on high-level representations using continental zoom levels. Fast interaction is preferred over a large degree of details. □

Each of the aforementioned scenarios can further be restricted to the case where no temporal constrains are made.

1.3 Related Work

This section gives an overview of relevant previous work and important contributions. As shown further below, some ideas have been modified and reused in this work.

To start with the research of trajectory plotting,in 2017, Owens researched on finding a suitable data structure for storing movement trajectories and suggests to use a contraction hierarchy based approach [44]. The most important reference is the PATHFINDER (PF) paper [25] and its code repository [24]. The authors tackle the problem of storing and retrieval of a massive amount of road network trajectory data using a contraction hierarchy based index structure. On a continent-sized graph with over 400 million nodes and nearly a billion edges, all trajectories within a certain time-space restricted request can be reported in less than a few microseconds per trajectory. Besides the fast retrieval, the space for storing all trajectories reduced by a factor of nearly ten using path compression on the contraction hierarchy edges [25]. Since the data structure presented plays a crucial role for this work, a more in-depth description will be presented in Chapter 2.

As a first extension for [25], a group of students embedded the PF code framework into a server-client environment to allow for requests over the network using a *Pistache* [45] server [52]. The frontend is powered by the *Bootstrap* framework [12] and the JavaScript library *Leaflet* [1] for map support. An experimental plotting approach was shown, as well as an extensive user interface for adjusting request parameters.

Bekas developed a system for plotting street graphs using the graphics engine *Vulkan* [31] in 2019, which in principal can be integrated into the PF environment, but required a GPU [8].

1.3.1 Contraction Hierarchies

The concept of Contraction Hierarchys (CHs) originates from route planing and goes back to Geisberger in 2008 [27]. Since then, a lot of useful applications have been found and CHs are still under research (see latest publications from Proissl, Rupp, Funke, Wagner and Buchhold [9, 48, 49], all published in 2021). Besides the PF, which utilizes the CH structure for compression and retrieval, CHs are also used in the context of road and game maps where edge weights change frequently [15].

To our best knowledge, however, only little research on visualizing or plotting of contraction hierarchy graphs in particular has been done prior to this work. To mention related a work, however, Funke et al. used an augmented CH for efficient map rendering by extracting simplified subgraphs while allowing the actual route computations to be shifted to the client [26].

1.3.2 Trajectory Visualization and Analysis

Visualizing trajectories is a well researched topic across several domains [6]. Besides visual traffic-, movement analysis (see [34, 58]) and urban planning (e.g. [5, 18]), trajectory pattern recognition and learning ([28, 59, 60]) are of special interest. While the latter category focuses more on analytic, the former two intensively deal with plotting trajectories to explore and learn from the dataset.

As a tool for supporting exploration, the concept of *lenses* plays an important role: Krüger et al. [33] use different user-defined filters for interactively adjusting the user's view preferences. Besides origin- and destination-filters which restrict the result sets to trajectories starting and ending in the respective areas, waypoint lenses screen out paths not intersecting them at least once. Using boolean operations, the lenses can be nicely interlinked which each other while hierarchical time views help to filter for special day types.

In the paper of Al-Dohuki et al., a massive taxi trajectory dataset was indexed, serving as a base for their exploration application, which also allows for semantic searches using street names and points of interest [18]. A framework for indexing, storing, retrieving and visualizing of trajectories semi-automatically has been presented by [17]. The authors also match data trajectories with street graphs, and enhance the datapoints with region, zip code and cell structure information to allow for more complex retrieval queries.

1.3.3 Graph Complexity Reduction

Reducing a given graph can be grouped into the concepts of *pruning* and *simplifying*. In our case, graph pruning describes the process of extracting relevant edges from a graph resulting in a simpler subgraph only containing significant (defined analogous to [14]). Simplification as a more abstract concept allows for introducing new or editing existing edges and nodes. They both have in common that the resulting graph can be described using fewer information while keeping most of its characteristics and shape.

Chimani et al. presented various pruning strategies for continuous graph simplification: They formalized the edge-selection procedure as an optimizations problem called *EdgeScheduling* and solved it greedily and using an approximation, since finding the optimal solution in general is $\mathcal{NP}$-hard [11]. The problem of road segment selection with stroke and stability constraints was researched by van Dijk et al. [16]. A stroke is a set of adjacent edges and returning edges in strokes is preferred over single independent edges. Additionally, the graph structure should change as little as possible while varying the simplification over time. This change from one instance to its following is penalized using stability conditions. The authors have shown that each problem individually can be solved optimally via dynamic programming or MinCostFlow respectively whereas the combination is $\mathcal{NP}$-hard [16].

Graph simplification is closely related to line simplification by replacing unconnected adjacent edge strokes by their simplified versions. Two methods we relate to further below were developed by Douglas and Peucker [19]. They first showed three different eliminating methods: Reducing points one by one, optimizing w.r.t. a mathematical optimization criterion and deletion based on specific carthographic features like crests and troughs. Then an iterative vertex selection process is presented, where a decision is made based on the distance to the line-baseline.

A more recent method developed by van der Poorten et al. uses triangulation combined with a triangle importance metric as enhancement of earlier methods [47].

For most of the scenarios the retrieved edge will be too large to apply common reduction techniques in an online-processing fashion, since already unpacking can be very expensive in terms of time needed, which will be showed in the following chapters Prior to introducing the implemented exporting algorithms, the motivation for designing them and their properties, the most basis concepts are explained in the following chapter.

2 Foundations

This chapter introduces the main terminology and defines basic concepts which are used later in this work. Most of the definitions are based on [25], which is also the base for this work.

The input is defined with respect to an input graph G, where each note originates from a coordinate in $\mathbb{R}^2$. A central concept in this work is the *trajectory*, which will be defined at the very beginning:

Definition 2 (Trajectory) A trajectory is a path $\pi = v_0 v_1 \dots v_n$ is an ordered sequence of nodes v_i taken from a graph G linked with a same-length time stamp sequence $\tau_0 \tau_1 \dots \tau_n$ with $\tau_i < \tau_{i+1}$ where two consecutive nodes are connected by an edge $e = (v_i, v_{i+1})$ from G. The latter restriction ensures a valid rewriting of the trajectory as $\pi = e_1 e_2 \dots e_n$ with $|\pi| = n$. A set of trajectories is bundled in a trajectory collection $\mathcal{T} = [\pi_1, \dots, \pi_N]$. □

Depending on the concrete context, one representation is preferred over the other, but both terminologies can be used interchangeably.

Definition 3 (Requests) A spacial request is a bounding box $Q_s = [x_l, x_u] \times [y_l, y_u] \subseteq \mathbb{R}^2$ defined by a lower and a upper coordinate, which searches for all paths π with non-empty intersection. A spatio-temporal request extents Q_s by an additional time constraint $[\tau_l, \tau_u]$. In the case of an abstract request Q, more complex temporal restrictions are included, such as daytypes, specific hours, months or years. □

2.1 Contraction Hierarchies

The algorithms used in the main retrieval data structure internally relies on a Contraction Hierarchy (CH), which is the result of augmenting a weighted input graph G with additional *shortcut edges* every time a *node contraction* operation is performed. The output graph is denoted as G_{CH}, having the same vertex set, the extended edge set with added shortcuts, a cost function c, and a depth d. A shortcut edge $e = (u, v)$ can be decomposed into two edges $e_1 = (u, w)$ and $e_2 = (w, v)$ where each e_i can again be a shortcut edge. Shortcut edges are added iteratively during the process of CH creation. Both cost function c and the depth d are defined recursively: The cost of a shortcut edge is the sum of its children's cost, the depth equals the by one incremented depth of the maximum of both children's depths. Non-shortcut edges have the same cost as in G and depth 0. A node w from G is contracted by removing it and all adjacent edges and inserting shortcut edges e_1 and e_2 if uwv was part of a shortest path. Eventually, no vertices are left. The ordering in which nodes are chosen for contraction heavily influences the maximum depth and also the length of the shortest paths in G_{CH} [25, 27]. Non-adjacent nodes can be contracted in paralleled.

The level of a node is defined by the number of rounds it passed until it got removed. All nodes contracted at the same time do have the same level. $l_{\max}$ denotes the maximum level, which also represent the total number of rounds. The level of a shortcut edge is defined by the level of the contracted node w. Non-shortcut edges have level 0.

Definition 4 (parents) Let e be a shortcut edge in the graph having the children e_1 and e_2. Then, e is called the *parent* of e_1 and e_2. □

Since the two edge types must be well distinguishable, non-shortcut edges are defined:

Definition 5 (plain edge) An edge e is denoted as *plain edge* iff. it has no shortcut. □

Consequently, by definition, an edge is either a parent edge (shortcut) or plain, but never both. The CH's input graph only consists of plain edges, i.e., they were all part of G.

Path Compression

Augmenting the graph G allows *compression* on paths by using the newly added shortcut edges. The compressed representation π' of a path π is obtained by repeatedly checking if two consecutive edges e_{i-1} and e_i are part of a shortcut edge $\hat{e}$. If so, the subsequence $e_{i-1}e_i$ is replaced by $\hat{e}$ and the search continues with $\hat{e}e_{i+1}$. Trivially, this can be done in time linear in the length of π [25].

Definition 6 (root edge) The edges of the compressed instance π' of path π are called root edges.□

The compression scheme is lossless and guarantees $|\pi| \geq |\pi'|$, where $|\pi'|$ is typically much smaller which allows to store trajectory data very efficiently. On the other hand, retrieved paths are compressed and cannot be used directly. Therefore, some post-processing during the export into a format which can be used for drawing has to be done.

Path Unpacking

The process of path compression can be reverted easily, since each shortcut edge keeps track of its children. Algorithm 2.1 shows the complete decompression operation, where the paths π and π' are represented as edge lists and the union operation $\cup$ concatenates two lists. The reversed stack method first reverses the list ordering and interprets the new most right element as stack top.

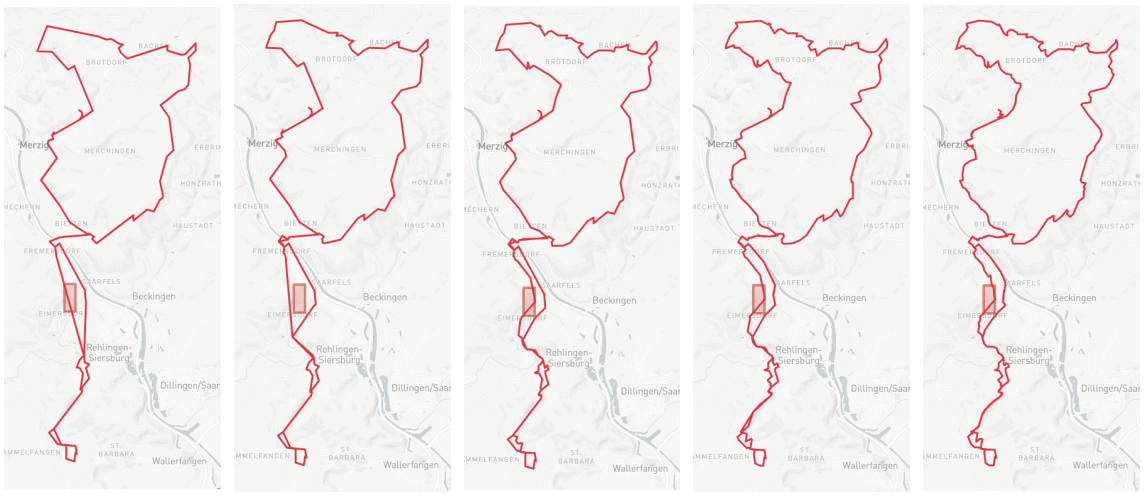
Algorithm 2.1 Unpacking a path (reversing the compression)

```
1:  procedure UNPACKPATH(π′)
2:      π ← []
3:      open ← TOREVERSEDSTACK(π′)
4:      while |open| > 0 do
5:          e ← open.pop()
6:          if e.is_shortcut then
7:              e1, e2 ← e.children
8:              open.push(e2)
9:              open.push(e1)
10:         else
11:             π ← π ∪ [e]
12:         end if
13:     end while
14:     return π
15: end procedure
```

Since unpacking reverses the original path by undoing the shortcut replacements in the reversed order, it has the same time complexity $\Theta(|\pi|)$.

To get a simplified version of π, one can stop the unpacking process at a certain depth. To do so, an upper bound on the edge level is defined: $l_{\sqcap}$. The algorithm for fine-tuneable unpacking steered by the choice of $l_{\sqcap}$ is explained in Algorithm 2.2 and slightly modifies the original. By definition, $l_{\sqcap}$ cannot be smaller than the lowest edge level 0 and all choices for $l_{max} \leq l_{\sqcap}$ results in π', i.e., no unpacking is performed.

Figure 2.1 demonstrates the simplification range by varying $l_{\sqcap}$. The smaller $l_{\sqcap}$ was chosen, the closer the shape approximates the original path by transmitting more data.



(a) π' (3.80kB) **(b)** $l_\sqcap = 25$ (5.14kB) **(c)** $l_\sqcap = 15$ (9.34kB) **(d)** $l_\sqcap = 5$ (16.56kB) **(e)** π (36.8kB)

Figure 2.1: (b) - (d) show results after partially unpacking for varying $l_\sqcap$. (a) shows π' ($l_\sqcap = l_{\max}$) and (e) for $l_\sqcap = 0$, therefore π. The corresponding transmission sizes are attached in brackets.

As Figure 2.2 suggests, there's no linear relationship between the choice of $l_\sqcap$ and the data being transmitted: The transmission sizes decreases exponentially for small levels. Later in this work it will be shown that $l_\sqcap$ must be chose carefully.

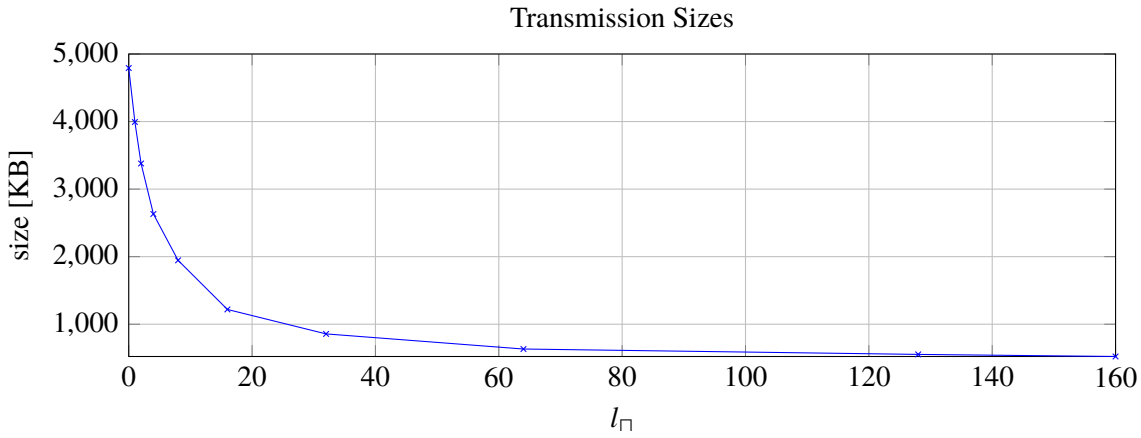


Figure 2.2: Total response size for requesting the area depicted in Figure 2.7 for different unpacking thresholds.

Compressed Path Traversal

A central concept in this work is the interpretation of the top-down view for a root edge as an unpacking tree. Given a root shortcut node e with child edges e_1 and e_2, this corresponds to a binary tree with root node e and children e_i. Unpacking the children recursively results in a saturated

Algorithm 2.2 Partially unpacking a path

```
1:  procedure UNPACKPATH(π′, l⊓)
2:      π ← []
3:      open ← TOREVEREDSTACK(π′)
4:      while |open| > 0 do
5:          e ← open.pop()
6:          if e.is_shortcut ∧ e.level then          // Only unpack edges above threshold
7:              e1, e2 ← e.children
8:              open.push(e2)
9:              open.push(e1)
10:         else
11:             π ← π ∪ [e]
12:         end if
13:     end while
14:     return π
15: end procedure
```

binary tree having either two children in case of a shortcut edge, or no child in case of a plain edge. The latter are the tree's leaves. The input π' can be interpreted as list of trees, namely a forest with the root edge being the root node for each tree. An interesting property, which is implicitly used by the unpacking procedure, is that traversing the forest from left to right and the trees in-order, the fully unpacked path π is obtained if only leaves are reported. This connection will become important later.

When a compressed path π' is returned from the PF, it is unknown how many leaf nodes it will represent, i.e., how large $|\pi|$ will be – it is not even known which root edge will contribute how much or which of the two root edge's leaves do have more (grand-) children. It can only be estimated based on the (children's) edge levels.

2.2 PATHFINDER Algorithm

PATHFINDER (PF) is a data structure for efficiently answering complex requests R by reporting all matching trajectories. It was developed at the University of Stuttgart in 2019. This section only focuses on giving an introduction and highlighting the important concepts, which will be used later in this work. A comprehensive method evaluation and speedup techniques explanations are given in the original paper [25].

At its core, PATHFINDER builds up and extends a contraction hierarchy for both storing and efficiently retrieving of trajectories. The reporting of results is done in three steps, which are shown in Algorithm 2.3.

Algorithm 2.3 PATHFINDER Algorithm (High Level)

1: **procedure** RUNQUERY(Q)
2: $E_O \leftarrow$ FINDEDGECANDIDATES(Q)
3: $E_r \leftarrow$ REFINEEDGECANDIDATES(Q, E_O)
4: $\mathcal{T} \leftarrow$ GETASSOCIATEDTRAJECTORIES(E_r)
5: **return** $\mathcal{T}$
6: **end procedure**

PF utilizes the CH hierarchy as spacial search structure by storing additional information: For each edge, a bounding box containing all it's vertices is (in case of a shortcut edge recursively) created. Additionally, bounding boxes are linked with each vertex containing all nodes reachable from it using *down paths*, which only consists of *down edges*. A down path of v is a path in G_{CH} starting at v, where the contraction levels of all subsequent nodes decrease. In a first step, the CH is traversed level-by-level in a top-down fashion by starting with all vertices v which don't have higher direct neighbors. Latter are called CH's top nodes Each adjacent down-edge (v, w_i) is added to the list of candidates E_O, if its edgebox intersects the query's box. The recursion continues with all vertices w_i, whose downgraph has non-empty intersection with Q's box.

To speedup the traversal, besides parallelization, efficient top-node look-ups, and skipping of unused or redundant edges, search paths can be pruned if a w_i's downgraph box is fully contained in Q, because all (grand-)children's down path boxes will intersect too, due to transitivity.

So far, only edges were collected, whose bounding boxes intersect Q. This, however, represents a superset of all valid solutions only, since a non-intersecting edge can have non-empty intersecting bounding boxes. To check for real intersections, candidate edges are unpacked until an unambiguous decision can be made, because either the unpacked edges e_i do not intersect the query any longer or a child with a clear intersection is found. Unambiguously intersected edges are added to E_r. According to [25], it is almost never necessary to completely unpack an edge for a definite decision.

To allow Q to have temporal restrictions, the spacial-only variant has to be extended: Further time interval information is added to edges and vertices, similar to the downgraph- and edgebox-constructions, but representing time intervals now. The adding of time slice bit vectors allow for periodic time event requests.

In the last step, the joined set of all referenced trajectories for all edges in E_r are returned after all duplicates have been removed.

2.3 Maps and Tiling

This section focuses on plotting real-world trajectory data, therefore, this last section briefly touches on map, navigation and tiling.

Visualizing a path π comes down to plotting a line stroke of several points. Each point is described by a latitude-longitude (lat-lon) tuple (λ, ϕ) with $\lambda \in (-180, +180]$ and $\phi \in [-90, +90]$, where $\phi = 90$ is reserved for north and $\lambda = -90$ for south pole. Fixing λ yields a vertical line, fixing ϕ a horizontal one. From Figure 2.3 it is evident that while two points on a vertical line do always have

the same distance to each other regardless of the choice of λ, distances between two points (λ_1, ϕ) and (λ_2, ϕ) have different distances, depending on the choice of ϕ [41, 43]. This becomes important when distances between points have to be calculated.

The embedded map layer has zoom layers $1-18$, where the first is very coarse and the last represents very small map sections. In this work, levels $1-6$ define continent, 7 and 8 country, $9-11$ metropolitan, and $12-18$ city scales, defined similarly to [43].

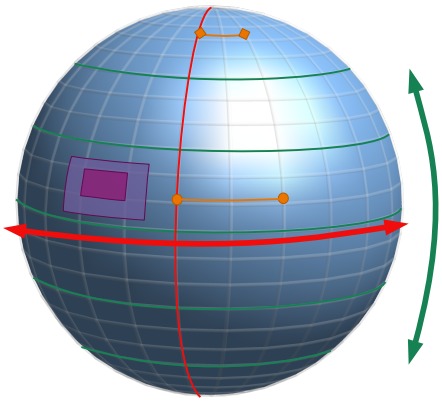


Figure 2.3: Varying longitude λ (red) and latitude ϕ (green). Distances between point pairs for different ϕ values are colored orange. Two example zoom views are depicted in light and dark purple showing smaller and larger zoom scales, receptively.

This upcoming chpater briefly introduces the preliminary work all implementations base on and presents the main data processing pipeline which was in used prior to this work.

2.4 PATHFINDER Data Pipeline

The earliest PF code release[1] was published after submitting its corresponding paper [25] in 2019. The PF application is embedded in a pipeline of processing steps which in total allow the user to answer trajectory data requests and plot the result using a heavyweight graphics card render engine. The pipeline is depicted in Figure 2.4

[1] https://gitlab.com/anusser/pathfinder/

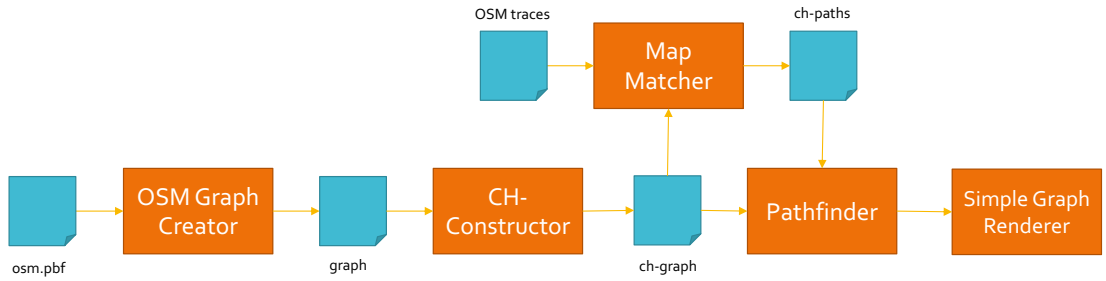


Figure 2.4: The PF's data- and optional visualization pipeline architecture.

The input graph data originates from *Geofabrik*[2], a company providing open source map data from OpenStreetMap (OSM). The input data is parsed by the `OSM Graph Creator`[3], which was developed by the Institute for Formal Methods of Computer Science (FMI)'s algorithmic group. The temporary output graph serves as an input for the CH construction program which, in turn, saves the modified shortcut enriched graph to file. The FMI's `Map Matcher` (see [51]) parses the OSM traces and links each trajectory with a set of corresponding pairwise adjacent graph edges. Each real world trajectory is now represented by a list of CH-edges. Both the CH-Graph and the matched traces are finally fed into the PATHFINDER. To test and visualize its correct functionality, a local plotting application named `Simple Graph Renderer`[4] (again, FMI) was utilized. As the name suggests, its input processing capability is limited to a list of 2D coordinates defining the points, and edge definitions as pairs of nodes. Additionally, for each node and edge, a color (r,g,b,a)-tuple has to be set. The picture Figure 2.5 shows a typical map excerpt where the edges, colors were calculated in advance to visualize usage frequencies.

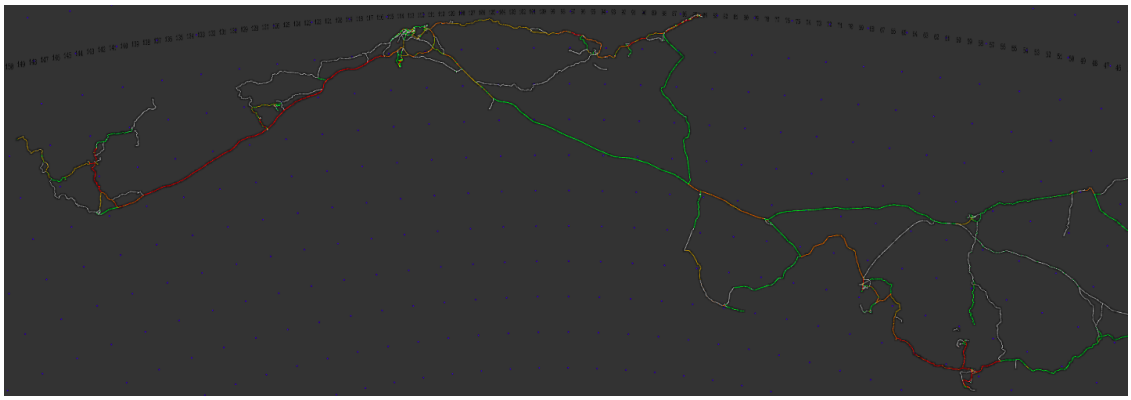


Figure 2.5: The project's predecessor: Simple Graph Renderer.

[2] https://www.geofabrik.de/
[3] https://github.com/fmi-alg/OsmGraphCreator
[4] https://github.com/invor/simplestGraphRendering

2.5 PATHFINDER^{web}

The aforementioned visualization solution renders the already unpacked paths in full detail. Once the path-plotting process has finished, navigating through the graph cannot be done smoothly, which has already been confirmed by [8]. Further, it lacks support for interactive requests, since the input rectangle has to be drawn by setting coordinate parameters.

To improve usability, a client-server infrastructure named PATHFINDER^{web}[5] was developed at the University of Stuttgart to send requests by drawing a bounding box and answering them plotted as an overlay of a set of polylines in a map. Each trajectory is plotted on its own and meta information is attached. This allows for iteractive hover- and on-click options to display further information.

To improve responsiveness - especially on larger zoom scales - a partially unpacking approach was implemented, which only unpacks paths down to a certain level threshold, as explained in Algorithm 2.2 This threshold is chosen dynamically with respect to the current zoom level (as defined in Section 2.3) and the graph's maximum unpacking level.

The unpacking level is obtained by the following formula:

$$a = \frac{1}{l_{max}}, \quad b = \frac{\log(l_{max}/\eta)}{\log(z_{max})}$$
$$l(z) = \left\lfloor \frac{1}{a \cdot z^b} \right\rfloor$$

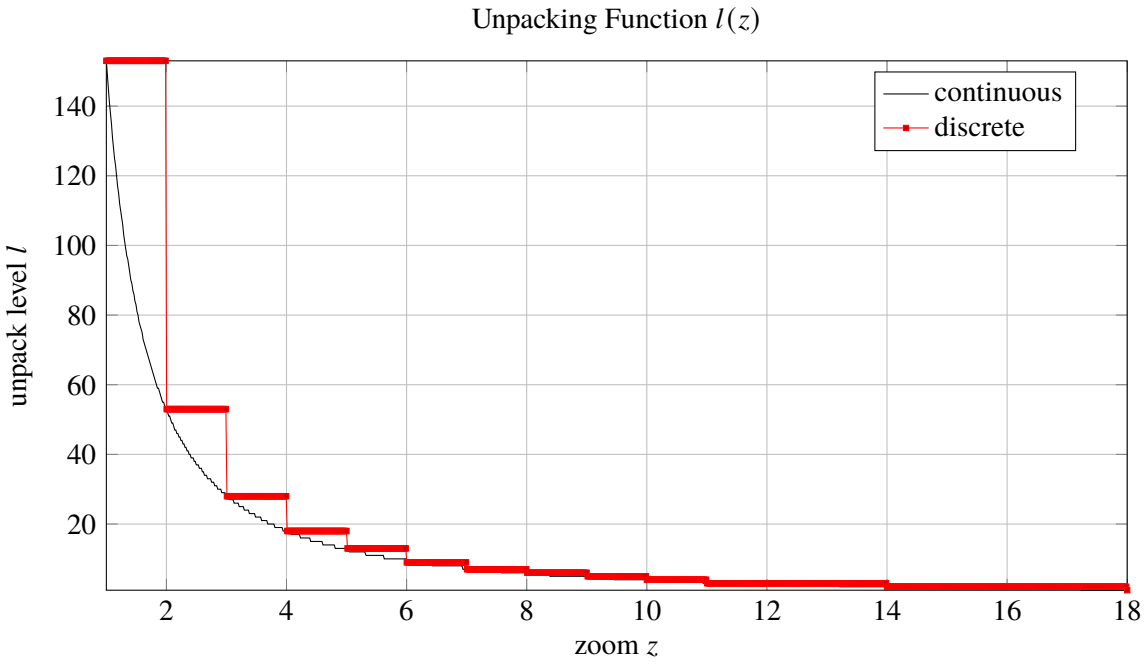


Figure 2.6: Level unpacking function in case of the Saarland graph with default parameters and $l_{max} = 153$.

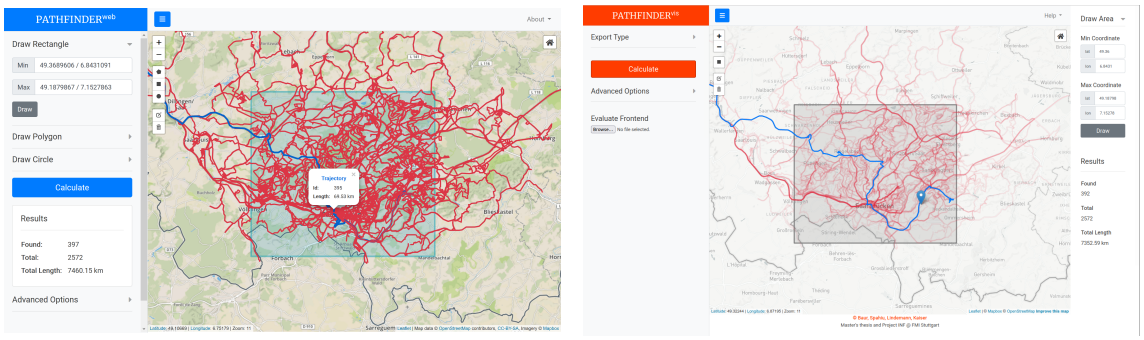
[5] https://bitbucket.org/baurls/pathfinder-web/src/master/

Here, η is a smoothness parameter, l_{max} the CH-graph's maximum level and z_{max} the maximum possible zoom scale. The default values proposed by the authors are $z_{max} = 18$ and $\eta = 1.8713$. Figure 2.6 shows an example function for the Saarland graph. Because the embedded map support only allows discrete map zooms, only 18 possible unpacking levels are required.

All work done base on this implementation and extends it by various plotting algorithms motivated by the different user groups as introduced in Definition 1.

Basic Enhancements

The very first improvement enhances the plain trajectory plotting by introducing semi-transparent paths. This results in path sections of high usage will have more color saturation, due to the overlaying of multiple polylines. In addition, a prototypical heatmap has been implemented by first (partially) unpacking the resulting trajectories and transmitting the points only, including with a weighting which depends on the number of trajectories passing through.



(a) **(b)**

Figure 2.7: (a) Screenshot of the successor project PATHFINDER^{web}. Image is taken from the original publication [52]. (b) shows the equivalent scene using the transparent-sensitive extension with a different tiling embedded.

3 Segment Graph

To illustrate the motivation for the following methods, it is worth to further investigate the behavior of shared edges: When increasing the number of trajectories defined on a fixed-sized graph, the number of edge reuse also increases. Assuming the requested trajectories have a lot of edges in common, it might be worthwhile spending some preprocessing time in reducing duplicated edges from the resulting response. To further illustrate this idea, the edge properties of common datasets are listed in Table 3.1: The path-edges column describes the full number of edges obtained when unpacking all paths in the dataset. The unique column accounts for the fact, that some edges are shared among the paths and only lists the total number of edges used once. The ratio between all path-covered and uncovered graph edges is denoted as coverage. Finally, the uniqueness-score represents the ratio between edges which were used only once and all edges covered by some path.

	paths	edges	path-edges	unique path-edges	coverage	uniqueness
Cuba	233	$4.76 \cdot 10^6$	$1.02 \cdot 10^5$	58,751	1.24	57.34
Saarland	472	$2.73 \cdot 10^6$	$1.69 \cdot 10^5$	$1.16 \cdot 10^5$	4.25	68.5
Germany	92,501	$2.48 \cdot 10^8$	$3.13 \cdot 10^7$	$1.37 \cdot 10^7$	5.53	43.87

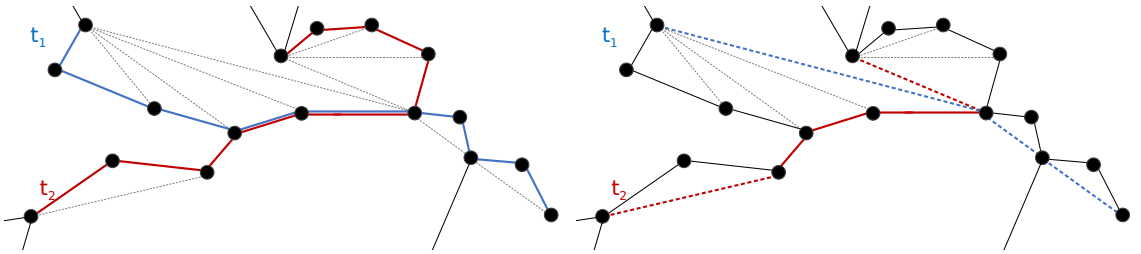
Table 3.1: Dataset properties on fully unpacked paths on different CH compressed graphs. Uniqueness and coverage are given in percent.

From the edge covering data, two conclusions can be drawn: Firstly, a fairly small amount of edges is in use. By numbers, in our real world examples, approximately 94 – 99% off all graph edges are not part of any path. Note though, that the CH has introduced a decent amount of new edges, which are not in used due to unpacking. Moreover, a fair amount of edges is used more than once: Only around 44-69% of the covered edges are used exactly once.

	path-edges	factor	unique path-edges	factor	coverage	uniqueness
Cuba	7,532	13.6	5,684	10.34	0.12	75.46
Saarland	23,393	7.24	19,956	5.82	0.73	85.31
Germany	$3.27 \cdot 10^6$	9.59	$2.29 \cdot 10^6$	5.99	0.92	70.23

Table 3.2: Dataset properties on compressed paths on the different CH compressed graphs. Uniqueness and coverage are given in percent.

From the data of Table 3.2, one can see that the number of edges per path (columns path edges) reduces massively due to the CH path compression. We obtain reduction factors between 7 and 14, which also affects the coverage, since multiple single edges are now represented (and therefore



(a) Uncompressed paths **(b)** Compressed paths

Figure 3.1: Showing two trajectories in two different settings: Using non-shortcut edges only (fully unpacked) and its compressed representation. Dotted edges indicate shortcuts. While the two paths share some edges in their unpacked representation, they do not in their compressed one.

replaced) by shortcuts. At the same time, the number of unique path edges decreases less vigorously, meaning that there are less edges being shared. This behavior was expected, because not all shared edges are mapped to the same shortcuts. This effect is illustrated visually in Figure 3.1.

3.1 Edge Index

To show the potential of sharing edge information, before the message is transmitted, the response is post-processed by listing all edges uniquely and store the trajectories as lists of indices. Instead of sending a list of coordinate-lists, i.e., one list for each trajectory, a list of all unique edge-coordinates and a list of index-lists is sent. This makes sense since indices are less space consuming than latitude-longitude coordinate pairs: A coordinate pair needs two doubles to be stored, while a reference only takes one integer. For an edge to be sent, four doubles are required.

Algorithm 3.1 shows the transformation of a list of edge-ID-lists as output of the PF to a unique list of edge-IDs and the correct index translation. The fact, that at least each input entry has to be read once already sets an upper linear time bound. By design, the algorithm has linear run time in the input data and is therefore optimal, assuming the map is implemented only consuming amortized $\Theta(1)$ for each operation. Considering the space consumption, $|R| = \sum_{\texttt{path} \in R} |\texttt{path}| = \sum_{\texttt{path}' \in R'} |\texttt{path}'| = |R'|$ and $|E| < |R|$ holds true, meaning that the algorithm consumes at most twice the input space. If needed, the above algorithm can be easily transformed into an inplace-variant.

A minimalistic example is shown in Listing 3.1. At time of transmission, each edge Identifier (ID) gets finally converted into its lat-lon representation, e.g. ID 783844 becomes $[[48.912192, 8.893239], [48.913612, 8.894719]]$. In principal, the same technique can also be applied to the vertices of the edges, to further reduce redundancy. Since this method only serves as demonstration, further details are omitted.

Algorithm 3.1 Simple Edge Indexing Algorithm

```
1:  procedure transformResponseList(R)
2:      E ← [], R′ ← []
3:      map : EdgeID ⟶ Index
4:      for all path ∈ R do
5:          path′ ← []
6:          for all edge_id ∈ path do
7:              if edge_id ∉ map then                    // Check for shared or unseen edge
8:                  map(edge_id) := |map|
9:                  E′ ← E′ ∪ [edge_id]
10:             end if
11:             path′ ← path′ ∪ [map(edge_id)]           // Replace Edge Identifier by Index
12:         end for
13:         R′ ∪ [path′]
14:     end for
15:     return E, R′
16: end procedure
```

```
R =  [
        [783844, 348733, 348734, 348735, 8293, 123442, 123443],
        [2623, 348734, 348735, 8293, 2893]
     ]
R' = [
        [0, 1, 2, 3, 4, 5, 6],
        [7, 2, 3, 4, 8]
     ]
E =  [783844, 348733, 348734, 348735, 8293, 123442, 123443, 2623, 2893]
```

Listing 3.1: Example illustrating the index-edge-splitting

Comparison Between Plain Text and Indexed Transmission

From a theoretical point of view, the more trajectories are added to the fixed graph, the more edges can be reused and the greater the benefits over the naive strategy are. On contrary, for inputs of (nearly) adjacent edge sets, e.g., each edge transmitted is used only once, and the index structure introduces unnecessary overhead. At this point an interesting research question is how *dense* the trajectories have to overlay the graph in order to make use of an edge-indexed result.

Setup This thesis cannot cover all scenarios, since there are many graph types, different zoom levels for unpacking, and parameter choices for generating or reusing realworld trajectories. To still give a reasonable intuition, different paths of maximum length 400 km were generated on the Saarland Graph. In a next step, requests were sent and both plain and indexed result sizes are recorded and plotted. The results are shown in Figure 3.2. Two exporting scales were tested, one using bounding boxes of half the graph's size, the other a quarter. For both scenarios, 100 bounding boxes were requested.

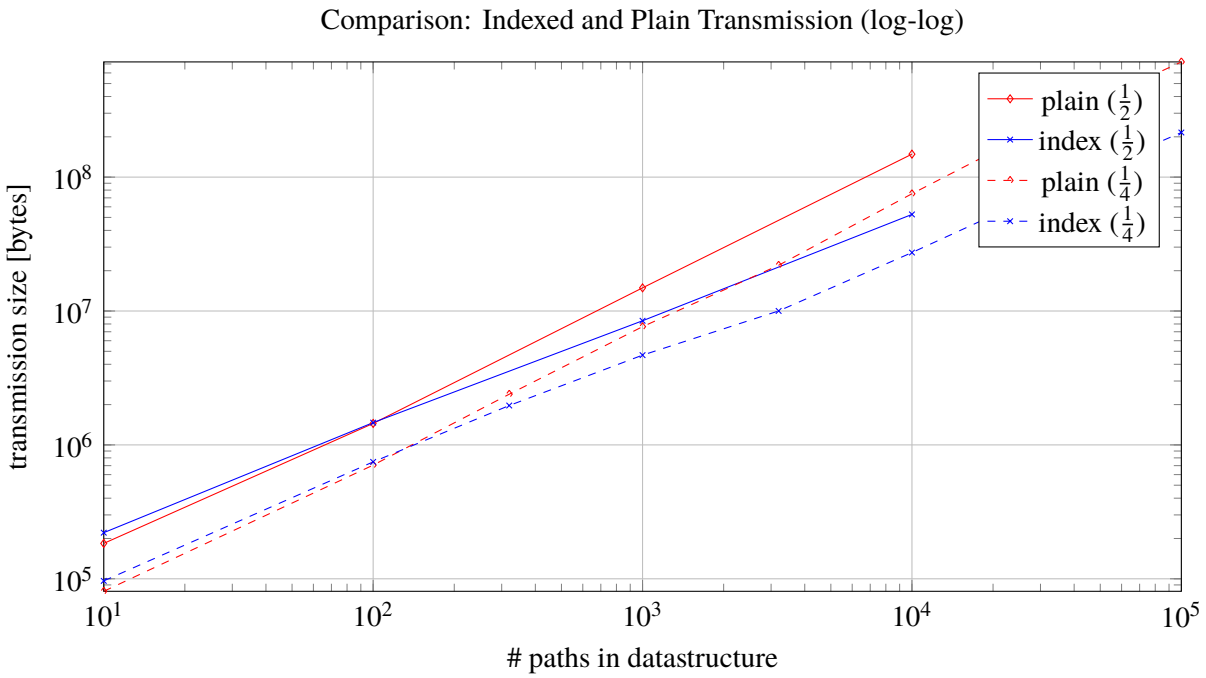


Figure 3.2: Transmission comparison between the plain transmission (red) and its indexed version (red) on different exporting levels. The fractions indicate the request box border lengths compared to the original graph's bounding box.

Algorithm 3.2 Segment Graph Algorithm (High Level)

```
1: procedure ConstructSegmentGraph(𝒯)
2:     V, map_v ← CreateVertices(𝒯)
3:     E, map_e ← CreateEdges(V, 𝒯)
4:     V_S, S ← MergeToSegments(V, E)
5:     return Graph G(V_S, S)                    // The Segment Graph
6: end procedure
```

Conclusion In all cases, the number of data being sent increases as more paths are added. Regardless of the zoom size, the inverted index performs worse on less paths, but outperforms the plain method for denser input graphs. Tp conclude, it is worth investigating aggregation methods to reduce the overhead instead of transmitting trajectory data isolatedly. The following method does not only aggregate partially unpacked trajectories edgewise but also groups path-strokes.

3.2 Segment Graph Algorithm

The Segment Graph Algorithm (SGA) first converts the input into a suitable graph and an efficient lookup data structure. In a second step, the temporary created graph gets transferred into the final segment graph. A high level algorithm sketch can be found in Algorithm 3.2.

3.2.1 Input

The input data is a list of r trajectories $\mathcal{T} = [t_1, t_2, \ldots, t_r]$ stored as a list of paths π_i. Since we only care about exporting the already found trajectories and temporal annotations are not relevant any longer, t_i and π_i are be used interchangeably.

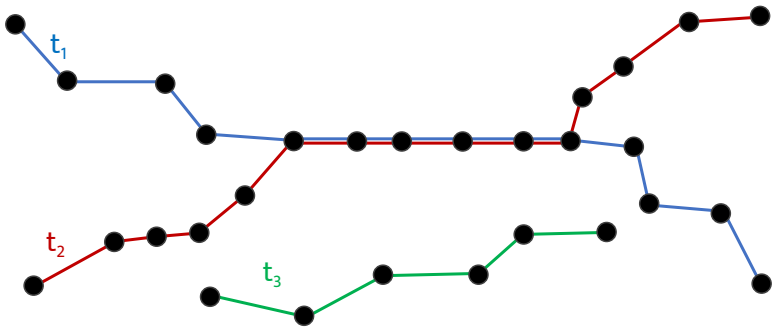


Figure 3.3: Example input $\mathcal{T}$ having three trajectories t_1, t_2 and t_3.

3.2.2 Preprocessing

Since each t_i represents a list of edges and therefore a sub-graph of the original one, $\mathcal{T}$ can also be interpreted as undirected graph $G(V, E)$ as visualized in Figure 3.3. The graph creation is done in the first two steps. Note that each edge is only added once even if it is used by multiple trajectories. The result is depicted in Figure 3.4.

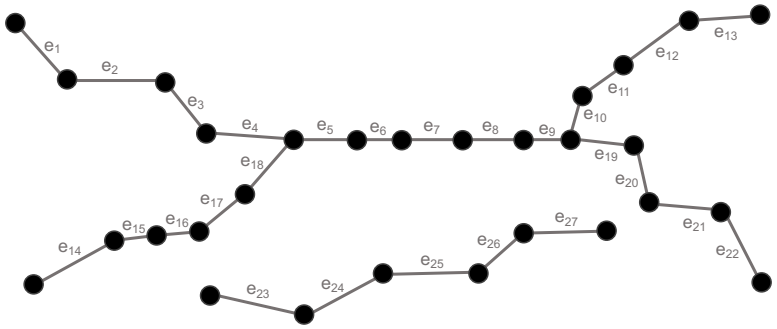


Figure 3.4: Graph $G(V, E)$. Extracted from the input $\mathcal{T}$ defined by Figure 3.3.

3.2.3 Edge-Labeling

To keep track, which edge represents which trajectories, a list at each edge storing this information could be introduced. This, however, is not efficient for the following reasons:

1. **Memory**: A list of trajectory references has to be stored for each edge. The list grows with the number of trajectories passing the edge. Adding a new trajectory (consisting of k edges) into the graph results in adding k list entries.

2. **Time**: To compute whether two edges are part of the exact same set of trajectories, iterating over all list entries and executing pairwise comparisons is required, which takes worst case linear time in the number of referenced trajectories.

To resolve both issues, an external data structure is proposed, to keep track of trajectories passing each edge while only using pointers for each edge: The list of paths traversing an edge is managed using a concept similar to a finite-state automaton. An empty state machine is initialized, only consisting of one starting state which we call *empty state* s_ϵ. Implicitly all edges are in state s_ϵ before they are added. Every time a path edge e from a trajectory t is traversed, a check for graph-existence is performed. If present in the graph, one *transition* from the current state s to the followup state s' which represents the same set of trajectories it already had plus the new trajectory t. If no such transition $s' = \delta(s, t)$ exists, first both the new state s' and the pointer linking from s to it is created. Then, the edge gets labeled s'.

An example is shown in the Figure 3.5 below.

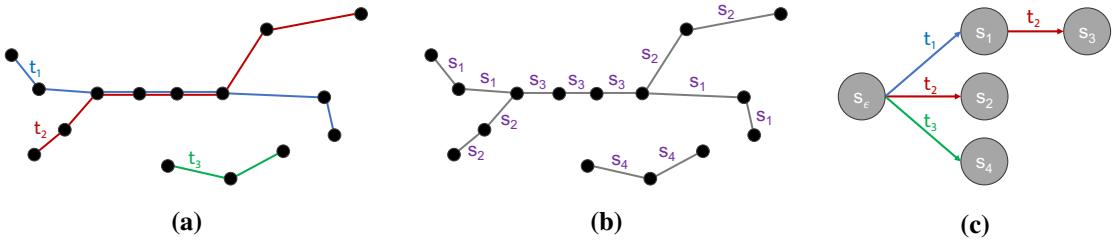


Figure 3.5: Example edge adding: (a) shows the input graph, (b) its corresponding edge states after inserting, and (c) the created transition data-structure.

Only storing a pointer for each edge which links to an internal *trajectory passing state* will ensure small storage requirements per edge. Also, checking two edges for having the same list of trajectories passed can be done in constant time by simply comparing their state-pointers.

3.2.4 Merging Edges

The data structure created during the edge-adding procedure will significantly reduce the complexity for merging edge strokes to paths, since only neighboring edges have to be checked for their state pointers. The merging approach is straight forward: All graph components are marked as *not merged yet* on initialization. Then, all vertices are iterated through to check if a merge is applicable. A merge is possible if the current vertex has only exactly two outgoing neighbors in the same state (this node is then called a *link node*) and it has not been merged yet.

If a link vertex allows a merge, a search along the outgoing paths to the left and right is performed to find the first vertices which are no longer merge-able. A sketch for such a node v is depicted in Figure 3.6. They are called s and t respectively. Along each path every merge-able vertex is marked and stored in a list. Once the path merging is competed, a new segment consisting of all inner nodes (those which just got marked) sorted in correct order is created having the end-nodes s and t .

(a) (b)

Figure 3.6: Example: Single merge step. (a) shows the search starting from link node v, marking all nodes on the way as invalid. In (b), a new s-t-segment is added and its traversed vertex list is stored.

After all nodes have been processed, the updated graph is returned. The edges which have not been removed during a merging phase are kept as segments without inner nodes. This results in the graph G which only uses a subset of input vertices. After the merging step, the initial example from Figure 3.3 would look like the following:

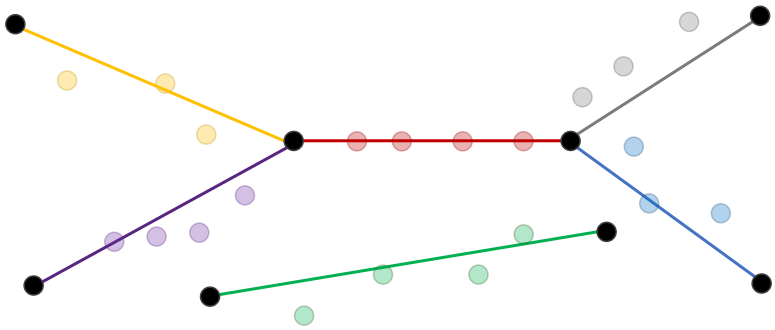


Figure 3.7: SGA's resulting segmented connecting graph. The transparent nodes are not part of the graph any longer, but stored in the correct order at each segment-edge.

3.3 Running Time Analysis

The size of the input PF output $\mathcal{T}$ is denoted as follows: $|\mathcal{T}| = \sum_{t \in \mathcal{T}} |t|$ with $|t|$ being the number of vertices the path t visits. Nodes can appear several times, since round trips or self-intersections are not excluded explicitly. Note that the number of edges of t equals $|t| - 1$, hence it suffice to proof an upper runtime bound for $|\mathcal{T}|$.

For the analysis it is assumed that maps have amortized constant reading and writing times.

Lemma 1 *The segment graph construction can be done in $O(|\mathcal{T}|)$ time.*

Proof The vertex creation is realized by two loops, traversing each node exactly once, and therefore $\Theta(|T|)$ time is guaranteed. For adding the edges, each edge is sequentially loaded once, both ends are first sorted and inserted into the map, if the vertex is not stored yet. Again, only constant time per iteration is needed. The labels are obtained similarly: A new edge gets a new state assigned, an

Algorithm 3.3 Segment Graph Creation (Low Level)

```
1: procedure createVertices(𝒯)
2:     V ← []
3:     map_v ← {}
4:     for t ∈ 𝒯 do
5:         for v ∈ t do
6:             if v ∉ map_v then
7:                 map_v(v) := |V|
8:                 V ← V ∪ createVertex(v)
9:             end if
10:        end for
11:    end for
12:    return V, map_v
13: end procedure
```

```
1: procedure createEdges(𝒯)
2:     E ← []
3:     map_e ← {}
4:     M = (S, Σ, s_ε, δ, F) ← ({s_ε}, {0, 1, ..., r − 1}, s_ε, δ, ∅)      // Initialize the trajectory state machine
5:     for t ∈ 𝒯 do
6:         for e′ = (u′, v′) ∈ t do
7:             e = (u, v) ← (min{u′, v′}, max{u′, v′})
8:             if e ∉ map_e then
9:                 map_e(e) := |E|
10:                s′ ← M.makeTransition(s_ε, t)                    // Creates s′ and δ(s_ε, t) := s
11:                E ← E ∪ createEdge(e)
12:            else
13:                s ← state(e)
14:                s′ ← M.makeTransition(s, t)    // Creates s′ and δ(s, t) := s′ (if not present yet) and returns δ(s, t)
15:            end if
16:            state(e) := s′
17:        end for
18:    end for
19:    return E, map_e
20: end procedure
```

Algorithm 3.4 Merge Edges to Path-Segments

```
1:  procedure mergeToSegments(V, E)
2:      E ← []
3:      S ← []
4:      V_S ← []
5:      markAllVerticesReal(V)
6:      markAllEdgesReal(E)
7:      for v ∈ V do
8:          if canMerge(v) then
9:              v.isVirtual ← True
10:             s, π_s ← findPathEndAndMerge(v, v.neighbors[0])
11:             t, π_t ← findPathEndAndMerge(v, v.neighbors[1])
12:             π ← CreateSegment(s, t, π_s, π_t)
13:             S ← S ∪ π
14:         end if
15:     end for
16:     for {u, v} ∈ {e ∈ E|e.isVirtual = False} do
17:         S ← S ∪ CreateSegment(u, v, [], [])
18:     end for
19:     V_S ← {v ∈ V|v.isVirtual = False}
20:     return V_S, S
21: end procedure
1:  procedure findPathEndAndMerge(v, w)
2:      π = []
3:      e ← {v, w}
4:      while canMerge(w) do
5:          w.isVirtual ← True
6:          π ← π ∪ w
7:          e.isVirtual ← True
8:          x ← getNextNeighborOnPath(v, w)
9:          v ← w
10:         w ← x
11:         e ← {v, w}
12:     end while
13:     e.isVirtual ← True
14:     π ← π ∪ w
15:     return π
16: end procedure
1:  procedure canMerge(v)
2:      return v.isValid ∧ v.isLinkNode
3:  end procedure
```

existing edge updates its label by creating a new state originating from the previous edge state. A neighbor counter and a map of outgoing edge pointers to each vertex are also added to allow for constant merge-comparison- and access times later.

The merge process checks each vertex for being merge-able and traverses its neighbors if so. Since an already merged node has been marked as *invalid*, a second merge process when visiting the vertex a second time is circumvented. Each inner node is therefore visited at most twice, each round using constant time. An intersection node v is visited at most $deg(v) + 1$ times (iff. it has $deg(v)$ many adjacent merge-able paths connected), but since each merge-able path has to have at least one inner node, which contributes to two intersection-node comparisons, the total number of intersection-node-visits is upperbounded by $2 \cdot |V_{inner}|$, where V_{inner} describes the total set of inner nodes. Latter is again upperbounded by the number of vertices. Creating a new segment requires to copy all its corresponding inner nodes (amortized $|V_{inner}|$), and linking the two end nodes. After creation, the path edges entering s and t gets replaced by the created segment-edge. Updating the links also requires constant time.

Hence, the total graph creation can be done in amortized $O(|\mathcal{T}|)$ time. ■

Theorem 1 (Runtime Optimality) *The SGA's asymptotic running time is optimal.*

Proof Since each input vertex has to be read at least once, a lower runtime bound of $\Omega(|\mathcal{T}|)$ is obtained. This, together with Lemma 1, yields the statement. ■

3.4 Discussion

A conclusive discussion based on transmission size and timing measurements using real world datasets follows. The chapter closes with a use case suggestion based on the evaluation's results.

3.4.1 Implementation

Before the transmission and creation performance is analyzed in Section 3.4.2, a few typical output examples are shown in Figure 3.8. Besides the choice for the bounding box, temporal restrictions can be set using either time interval or weekday filters. When the user hovers over a segment, its color changes to light blue. Clicking a segment colors all trajectories passing through it darker blue, and additional information like vertex, edge and trajectory IDs are shown. The *Trajectory Super-Imposition ID* is a reference to a correct edge state $s \in S$, i.e., this value is identical for all segments having the exact same trajectories in use. By default, the Segment Graph's unpacking level is determined by the zoom level on request time, but can be changed by switching the *packing level*.

3.4.2 Validation

This section briefly explains the experimental setup including the hardware and parameter range choices which were used for creating the measurements shown in this chapter's second half.

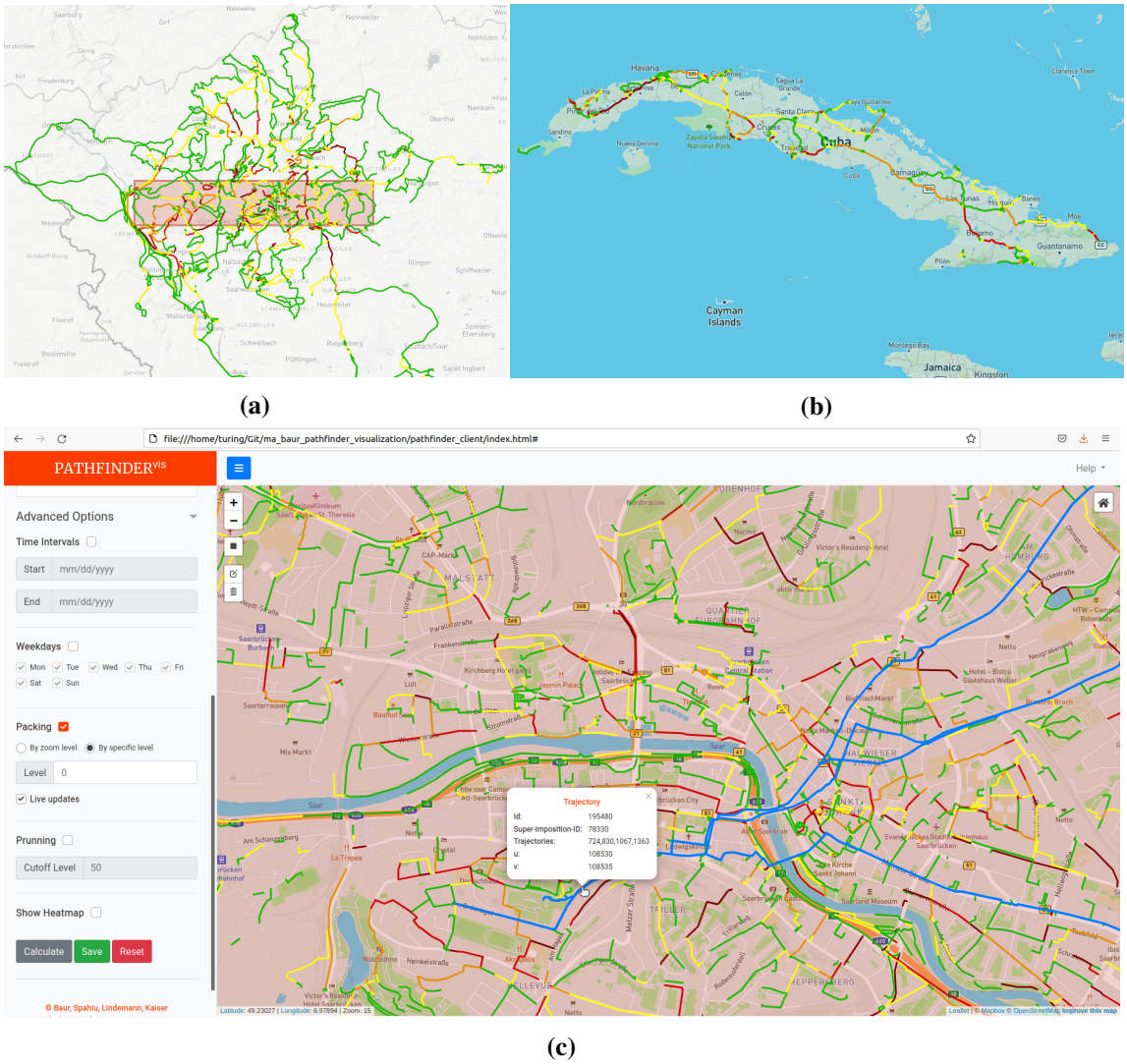


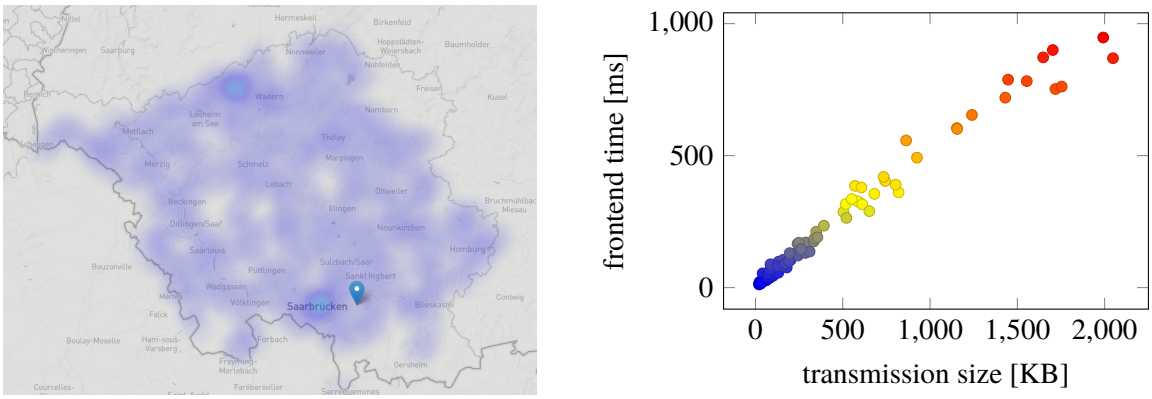
Figure 3.8: Typical SGA outputs: (a) visualizes real world data from the Saarland Dataset, (b) from the Cuba Dataset. (c) shows a user interaction on a Trajectory Graph segment in the center of Saarbrücken, where all trajectories passing through this segment are highlighted using the internal graph structure for traversal.

Setup

To evaluate the implementation, two devices of different performance were used:

(1) Threadripper: AMD Ryzen Threadripper 1950X (16-Core) with 256 GB RAM and Toshiba OCZ RD400 NVMe SSD (2.6 GB/s)

(2) ThinkPad: Lenovo ThinkPad T440s (2-Core) with 8GB RAM and SanDisk SDSSDA240G (479 MB/s)



(a) Saarland Trajectories' density map

(b) Positive plotting time correlation

Figure 3.9

For all frontend evaluations, Firefox 94.0 (64-bit) [13] running on the ThinkPad machine was used. Besides the times needed to process the frontend and backend data, the transmission size was recorded.

For evaluation, five different bounding box sizes were chosen: Each bounding box has a randomly chosen center and its size is determined by the graph's bounding box, down scaled by a constant factor between of $1/2$ (half the original's side lengths), $1/4$, $1/8$, $1/16$ and $1/32$. The zoom scale varied between 1 and 18, where the former returns a graph on a continental and latter a street-scale zoom. On the backend side, each measurement results from creating 20 requests and averaging the sizes and timing results. Frontend evaluation measurements were calculated using the 20 backend's request responses, each plotted five times and averaging over all values.

For this work, a reasonable response time is defined to be smaller than two seconds on average. This reflects the maximum tolerable waiting time found in [38]. For further argumentation, package transmission sizes are weighted by 0.279 s/MB, which is the average Germany internet transmission download speed of 28.7Mbps, according to a survey from 2020 [20] in which 87 524 176 592 connections were evaluated.

A total processing time of one second, for example, together with a package size of 3.5MB would be a response within approximately two seconds for an average user.

Measurements

Results for exporting Segments Graph data from the Saarland Dataset are listed in Table 3.3 and were calculated on the ThinkPad. The transmission sizes increase for larger bounding boxes and tend to increase for higher zoom levels. Closer inspection shows that the response times for smaller request boxes fluctuate more than for larger boxes, which can be explained by the non-homogeneous distributions of paths, meaning areas of different path densities, e.g., high density around cities and low densities in rural areas, large lakes or unpaved terrains. For illustration, the different densities are shown in Figure 3.9a. Backend calculation takes at most 0.52, frontend up to 0.95 seconds. Figure 3.9b shows that there is a strong positive correlation between the data sent and the needed plotting time.

(a) Intra-city analysis

(b) Macroscopic analysis

Figure 3.10: Use case analysis: While (a) shows good support for intra-city analysis, SGA's output helps only little for small zoom scale based macroscopic analysis.

For zoom boxes of up to $1/4$ the original graph box's edges length, responses do not exceed 0.85MB and processing takes less than 0.7 seconds on average. Request boxes of size up to a scale factor $1/2$ take up to 1.5 seconds processing and approximately 2MB in size. Therefore, all request boxes in the covered parameter domain are within the allowed limit.

Measurements for the Germany Dataset, which ran on the Threadripper, are shown in Table 3.4. Analogously to the Saarland set, times and network traffic rose for either enlarging the request box or increasing the zoom level. Since the graph is larger in size, even small boxes yield large transmission sizes. For the smallest request parameter pair (zoom = 1, size factor = $1/32$), for example, 42 times more data is sent compared to its Saarland equivalent. Different from the Saarland's requests, there are some parameter combinations for which the backend exceeded a generation time of five seconds. In this cases, the generation process was stopped and the final time was calculated by averaging the results sampled so far. Also, the frontend was not able to plot all recorded data instances, especially for larger request sizes, the client simply stops working or freezes.

3.4.3 Summary

The Segment Graph exporter module nicely visualizes street usages and supports highlighting of single trajectory groups. Streets of about the same usage are quickly identified using the five color scale. Lookup for trajectories which pass a single edge stroke is straightforward.

For larger bounding boxes, however, the views becomes less meaningful and transmission costs rise. Already on graph views with size about half of Saarland's side lengths, the network traffic and processing times reaches the defined limit.

Regarding the use cases listed in Definition 1, while the SGA well supports microscopic analysis and partially assists inter-city analysis work, the method is not suitable for macroscopic tasks due to its limited filtering and abstraction options. Two representative outputs are shown in Figure 3.10.

	Zoom																	
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Factor 1/32																		
Frontend [ms]	12.1	14.8	20.9	15.5	20.7	18.7	14.9	34.1	41.4	70.1	37.5	50.4	36.9	64.0	46.8	27.6	44.9	34.1
Backend [ms]	8.3	12.6	11.1	12.5	14.7	16.5	14.6	27.0	31.2	52.3	25.1	43.6	33.1	52.3	32.0	22.8	40.0	26.2
Size [KB]	20	22	23	27	36	32	29	61	71	122	44	96	81	126	81	62	101	64
Factor 1/16																		
Frontend [ms]	27.6	37.3	54.0	19.4	67.2	41.1	50.3	39.5	79.1	55.5	98.3	48.6	68.0	104.4	91.7	56.6	76.1	80.7
Backend [ms]	13.1	16.7	15.0	15.3	40.1	30.5	35.7	26.3	52.6	38.9	49.1	37.3	51.4	71.0	62.8	52.4	67.1	56.4
Size [KB]	44	48	41	35	103	73	87	68	137	97	132	95	132	187	169	131	179	152
Factor 1/8																		
Frontend [ms]	52.3	69.3	88.5	105.3	81.8	106.1	92.5	126.4	132.2	105.5	171.2	104.8	121.5	174.5	100.2	131.5	136.5	185.8
Backend [ms]	25.7	31.1	27.9	52.4	44.9	75.6	63.1	88.7	89.8	79.0	107.6	75.1	101.3	115.6	67.6	101.4	115.9	120.9
Size [KB]	72	85	86	158	126	200	170	235	237	191	289	201	243	333	193	285	310	338
Factor 1/4																		
Frontend [ms]	130.7	166.3	170.2	146.0	211.7	190.8	234.6	287.2	385.6	323.7	327.9	315.4	403.9	264.0	355.1	289.8	360.8	389.7
Backend [ms]	54.4	75.8	79.8	92.2	120.8	123.9	141.6	173.1	201.9	226.0	203.4	224.4	273.7	183.6	227.1	219.1	264.9	286.1
Size [KB]	196	241	249	261	348	355	392	504	570	595	581	612	743	521	681	652	820	803
Factor 1/2																		
Frontend [ms]	316.5	335.5	379.3	419.4	557.5	492.3	604.6	601.1	787.9	654.1	872.2	719.6	752.4	900.1	782.3	761.7	869.1	947.5
Backend [ms]	150.8	161.0	189.6	236.7	277.6	295.5	352.0	373.5	435.7	385.8	495.8	435.8	526.4	483.3	465.4	502.3	589.3	561.9
Size [KB]	517	551	607	734	863	925	1154	1155	1447	1240	1648	1431	1718	1703	1554	1754	2048	1992

Table 3.3: Segment Graph measurements: Saarland Data.

	Zoom																	
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Factor 1/32																		
Frontend [s]	0.54	0.55	0.52	0.65	0.55	0.76	0.62	1.78	3.05	0.99	1.05	1.53	1.47	1.12	0.95	1.38	1.77	1.56
Backend [s]	0.12	0.13	0.13	0.23	0.20	0.32	0.27	0.79	0.86	0.49	0.53	0.87	0.81	0.64	0.50	0.83	0.91	0.90
Size [MB]	0.82	0.91	0.82	1.29	1.08	1.62	1.45	3.76	3.80	2.30	2.56	3.91	3.39	2.91	2.48	3.61	4.24	3.95
Factor 1/16																		
Frontend [s]	1.02	0.97	1.34	1.73	1.67	1.72	2.65	1.82	2.14	2.56	**4.46**	**-**	2.64	**-**	**-**	-	**-**	**-**
Backend [s]	0.25	0.25	0.41	0.51	0.64	0.65	1.20	0.81	1.10	1.26	**3.09**	**3.60**	1.58	**2.33**	**2.54**	1.67	**2.17**	**2.44**
Size [MB]	1.84	1.80	2.56	2.86	3.56	3.64	6.01	4.06	5.25	5.89	**11.63**	**12.26**	6.70	**9.60**	**10.79**	7.80	**9.46**	**10.84**
Factor 1/8																		
Frontend [s]	2.67	-	-	-	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**-**
Backend [s]	0.69	1.39	1.05	1.51	**2.30**	**2.25**	**2.93**	**2.97**	**4.87**	**5.40**	**3.27**	**3.69**	**11.32**	**6.81**	**4.08**	**4.32**	**5.77**	**13.25**
Size [MB]	4.99	9.51	6.66	9.00	**12.45**	**11.68**	**14.53**	**14.55**	**21.71**	**23.89**	**15.84**	**16.96**	**42.15**	**31.78**	**18.29**	**19.44**	**25.63**	**52.69**
Factor 1/4																		
Frontend [s]	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**-**
Backend [s]	**3.90**	**4.46**	**6.30**	**8.79**	**3.68**	**5.96**	**14.48**	**17.30**	**7.01**	**8.16**	**8.75**	**10.98**	**14.65**	**5.13**	**14.85**	**13.75**	**31.90**	**20.91**
Size [MB]	**27.20**	**29.63**	**37.71**	**48.50**	**20.23**	**31.00**	**70.28**	**77.03**	**31.59**	**38.46**	**41.77**	**51.12**	**67.83**	**25.37**	**67.87**	**65.13**	**131.21**	**96.17**

Table 3.4: Segment Graph measurements: Germany Dataset. Bold values indicate generation timeouts resulting in a smaller number of samples. Missing values indicate plotting timeouts.

4 Batched Transmission

As discussed earlier in chapter 1, the number of points for a single trajectory might be arbitrary large. Especially for larger zoom scales it intuitively makes sense to omit less important points for transmission instead of posting all data points for each trajectory immediately. In Figure 4.1 a very rough approximation is sent first improving over time. If all points are sent at once, the user has to wait until the whole data transmission process completes to see any of the results or updates. The system's response time is much higher than necessary since always a fully unpacked path is shown although at large zoom levels a rough trajectory is sufficient. Clearly, users do not like waiting, and websites with long response times produce greater frustration [50] However, this frustration is eased if some form of vague sketch is shown first, which improve over time [38]. Analogously, it is state of the art for normal websites to first load the structural template and then fill it with content little by little.

This chapter touches on transmitting path data in chunks. While the other chapters focus on CHs, this chapter isolatedly shows some key concepts of batching mechanisms as a basis for future implementations and works with an unpacked edge data structure. This allows for constant access time to the ith node of a trajectory. Note that without preprocessing this is not possible for compressed paths: The lower level edges are only accessed through an unpacking procedure.

This chapter is split into four sections. In the first part, the main architecture and communication dependencies are explained and a metric is introduced to allow for a transparent and fair comparison of the methods. The subsequent second and third parts, present different transmission mechanisms. They differ in the order of traversal: Section 4.2 researches the more general case, where nodes are stores in a random-access manner index by the the order in which the path visited the nodes. To connect the transmission idea, the CH structure is included in Section 4.3. It still requires a full unpacking on the server side, but takes information about the unpacking hierarchy into account. Eventually, the chapter closes with a discussion of the results and findings.

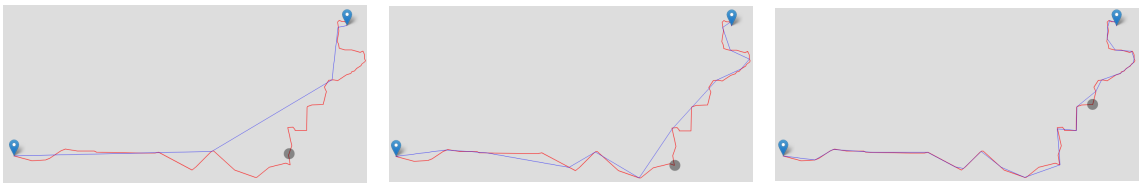


Figure 4.1: Example for continuously improving the plot using updates: simplified path (blue) approximating the original path (red) over time after 1, 3 and 4 received updates respectively.

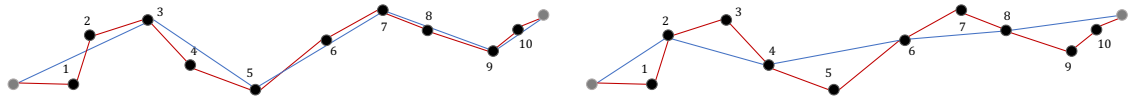
4.1 General Setting

Before the implemented methods are shown, a more abstract overview about the architecture and the evaluation methods is presented. The latter will become useful when analyzing the methods in the discussion. For the sake of simplicity, only one trajectory is considered in this chapter and time annotations are ignored meaning, meaning $\mathcal{T} = [t] = [\pi]$. The generalization to a set of trajectory is straight forward.

4.1.1 The Order of Transmission

Intuitively it makes sense to transmit data partially to allow for short interaction times. The choice order in which nodes should be transmitted is, however, not intuitive. From a theoretical view point, there are $n!$ possible orderings in which a length-n-path can be sent. Also, some methods require additional meta information that allow to reconstruct the path given only a stream of points.

Finding a good ordering, meaning a quick visual convergence to the original path even on only a fraction of points is not the only aspect that matters. Figure 4.2 shows, by way of example, intermediate results of varying quality for different order choices. The size of meta information which to be transmitted to recombine the packages must also be taken into account and should be sufficiently small. One of the goals of this chapter is to find a method that balances this trade-off by discussing the proposed methods in a transparent and measurable way.



(a) **(b)**

Figure 4.2: Example of different transmission order choices for ten inner nodes after transmitting four nodes. (a) chooses $[3, 5, 7, 9]$ and (b) $[2, 4, 6, 8]$ first. The original path is colored red.

4.1.2 Architecture

In our client-server-landscape, only three components have been added to allow for iterative refinement. On the server side, an update request module listens for incoming connections and sends either an initial package or an update batch. The client's *Batch Decoder* modifies the current path with the incremental update using the meta information sent along with the new nodes. Finally, the plotter updates the view.

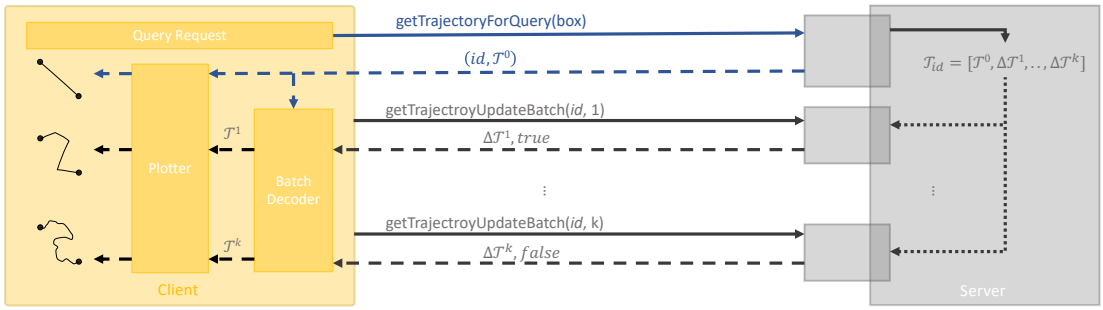


Figure 4.3: Main batching overview: The server receives a request and sends a response containing the trajectory and a very rough sketch. Iterative requests are used to refine the shape, until eventually the full point list is loaded.

Figure 4.3 shows the overall communication sketch: The client starts by requesting a trajectory within a defined bounding box from the server. After doing the calculation using the PF algorithm (see Section 2.2), the server immediately responds with the found trajectory ID and a rough sketch of the trajectory $\mathcal{T}_0$ using only a few points. A refined shape is obtained on the client's side by iteratively requesting update batches from the server which are built on top of each other. Wherever possible, the update policy is enforced to only rely on stateless server communication to ensure scalability and integrate the representational state transfer (REST) pattern [22].

4.1.3 Quality Measure

The goal of transmitting the data in batches is to obtain a quick and good estimate of the original trajectory line shape quickly and omit a large waiting time by iterative loading of details. To measure the quality of a partial solution after integrating i batches, the path distance measure *offset error* is used. It measures the largest gap between the original path $\mathcal{T}$ and its simplification $\mathcal{T}^i$. To do so, the same idea as presented by Douglas and Peucker in [19] is used, i.e., calculating the distance between each original point and its nearest point on the respective path $\mathcal{T}^i$. This distance is called maximum offset error (MOE). In the paper, the authors were mostly interested in finding a point violating a certain *offset tolerance*. In this work, however, finding the maximum offset tolerance for which all points of $\mathcal{T}$ are still covered, is of interest.

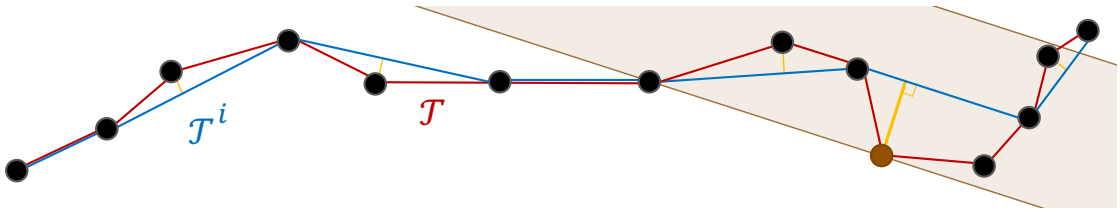


Figure 4.4: Visualization of the maximum offset error between $\mathcal{T}$ and its approximation $\mathcal{T}^i$. Distances are drawn in orange. The maximum offset point and its corresponding tolerance area are highlighted in mocca.

To compare different paths with each other, the absolute offset measurements can be scaled w.r.t different path properties. In this work, the relative maximum offset error (RMOE) is defined as the fraction between the absolute offset error and the path's length. Scaling a path in size does not change its relative offset error.

4.1.4 Naming Conventions

To declare a naming for the input path $\mathcal{T}$, a general constructive method for any path is given: Start with an initial graph without shortcut edges and mark an arbitrary u-v-path made up of $n-1$ plain edges. Label the nodes according to their sequential order, starting with 1 for u and ending with n for v. After inserting shortcut edges, i.e., compressing the path, the new u-v-path returned by the PF instance has only $n'-1$ edges, where n' is typically much smaller than n, but at most the same size. This naming scheme will be used throughout the whole chapter.

The abstract update rule is defined as follows:

$$\mathcal{T}^{i+1} = \texttt{process\_update}(\mathcal{T}^{i}, \Delta\mathcal{T}_{i+1})$$

where $\mathcal{T}^{i+1}$ is an improved version of $\mathcal{T}^{i}$ after applying update information $\Delta\mathcal{T}_{i+1}$. $\mathcal{T}^{0}$ is an initial path representation returned as a response for the very first server request. The batch decoder on the client side processes the update patches and includes the new information into the existing rendered structure. It yields a plain point list $\mathcal{T}^{i+1}$ based on the initial points and all $i+1$ update batches.

As mentioned earlier, a root edge implies a tree structure. Unpacking a list of root edges yields a list of vertices in correct order of traversal at recording time. Given that list, any other (search-) tree can be built on top of this ordered vertex list. In the following chapters, either artificial trees are built in case of the unpacking independent versions, or the given edge tree is utilized in case of unpacking dependent CH-implied trees.

Starting from that convention, the only topic that still remains open is the traversal of trees. While Section 4.2 used artificial search trees, 4.3 deals with the already given CH edge hierarchy.

In the following, a rough summary of hierarchical-traversal methods is given. The methods below are classified according to their iteration order, meaning its characteristic hierarchy traversing.

Sequential Ordering

The sequential order traversal simply processes, i.e., prints, the leftmost element which has not been processed yet. This results in a visiting order $[1, 2, 3, \ldots, n]$, hence its name. It can also be seen as an inorder traversal of any tree spanned on top. An CH-Inorder-Traversal iterator can be implemented efficiently by always unpacking the currently left-most edge and pushing the traversed edges to a stack until a non-shortcut edges is found. If so, its left node reported. Continue with the stack top, until there are no more edges left. Make sure to also report v.

Level Ordering

The level order reports nodes from top to bottom and left to right, where a node's level is defined by its distance to the root node. Even though the traversal is trivial once a tree is given, the output highly depends on the tree-construction and its resulting topology. Therefore, the tree shape algorithms below are defined in great detail.

Further Orderings

For the sake of completeness the traversing methods post- and preorder are also mentioned. The former processes the left and right child nodes first, the latter starts with processing the current node and then continues in left-right child order. These methods will not pursued any further, since they neither show as simple representation describing properties as inorder nor have coarse-to-fine properties as in level-order.

4.2 Unpacking-Independent Approaches

In this section the hierarchical structure which is implicitly given by the edge-unpacking process as described in Section 2.1 is ignored. Instead, a trajectory is defined as an ordered list of points where the first point is called u and last point v:

$$\mathcal{T} = [p_0, p_1, \ldots, p_{n-1}] = [u, p_1, \ldots, p_{n-2}, v]$$

Further it is assumed, that the first response returns the first line approximation (u, v), representing a straight line connecting the first and the last point:

$$\mathcal{T}^0 = [u, v]$$

Since u and v are already known to the client, they are ignored by the further transmission-algorithms. All subsequent responses will refine that list until the final update batch $\Delta\mathcal{T}_k$ will stop the refinement process and $\mathcal{T}^k = \mathcal{T}$ holds.

4.2.1 Sequential-Order Update

The most obvious but qualitatively poor conditioned method is the transmission of the remaining points in sequential order, grouped into $\left\lceil \frac{n-2}{b} \right\rceil$ at most length-b sized batches (with equality iff. $n \equiv 2 \mod b$). We call this method Sequential Order Update (SOU). Accordingly, the following update-batches and -rule apply:

$$\Delta\mathcal{T}_i = \begin{cases} [p_{(i-1)b+1}, p_{(i-1)b+2}, \ldots, p_{ib}] & \text{, if } i \leq \frac{n-2}{b} \\ [p_{(i-1)b+1}, p_{(i-1)b+2}, \ldots, p_{n-2}] & \text{, else} \end{cases}$$
$$\mathcal{T}^{i+1} = \texttt{process\_update}(\mathcal{T}^i, \Delta\mathcal{T}_{i+1}) := (\mathcal{T}^i \setminus [v]) \cup \Delta\mathcal{T}_{i+1} \cup [v]$$

Since all transmitted points are already in the correct order without any gaps, the update routine on the client side becomes trivial. The only requesting parameters for querying the next batch are the trajectory ID and an offset, where to start the next data chunk.

This transmission method obviously suffers from a small update locality: While the first segments along the path gets refined right at the beginning, loading the end path nodes close to v is postponed until the very end. This updating scheme is visualized in Figure 4.5.

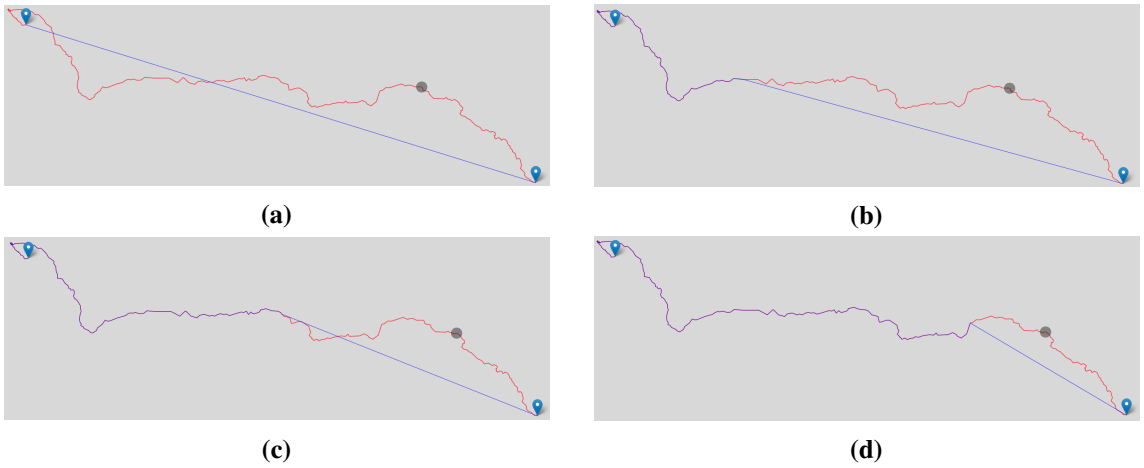


Figure 4.5: Example update process using SOU batches with a batch size of 300 points each applied to path #166 from the *cuba* dataset: (a) initial batch, and path resulting from (b) 5, (c) 7, and (d) 9 consecutive updates.

Sending a single point at each update is not desired, because there's a lot of overhead introduced by the Hypertext Transfer Protocol (HTTP) protocol [21] and routing communication. Hence, in practice we send a batch of $b > 1$ points each time. As one will see later, only compatibly little overhead for sending several smaller point lists compared to the other methods is introduced. Especially for larger batch sizes (such as $n/2$ or $n/4$), the overhead is asymptotically constant and therefore negligible.

To introduce a more global update variant, level-ordering is applied next.

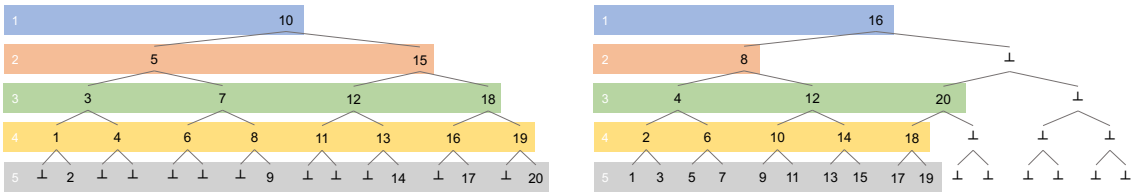
4.2.2 Level-Order Update

Labeling the $\hat{n} = n - 2$ inner-nodes (those which are not u,v) in order of their visit, p_1 gets label 1, p_{n-2} label $\hat{n}$. According to their label, a binary search tree is then constructed from the nodes. If there are exactly $\hat{n} = 2^k - 1$ nodes ($k \in \mathbb{N}$), a search tree can be constructed which is both saturated

and balanced. A formal proof can be found in Appendix A.1. In all other cases, various methods for construction can be used each having different properties. Two approaches will be presented in the following assuming to have $\left(2^{l-1}-1\right)+t$ nodes, where l is the largest power of 2 for which $2^{l-1}-1 \leq \hat{n}$. This bounds t to be in $\left[0, 2^{l-1}\right) \subseteq \mathbb{N}$.

Center aligned ordering: As an intuitive strategy, the most central node could be determined being tree-root. From that, the left and right neighbors are determined recursively. In the case where there is no central node, say there are $2k$ nodes, the kth is chosen. This ensures a tree with at least $l-1$ entirely full levels (and the lth layer being full if and only if $t=0$). An example for this ordering is depicted in Figure 4.6a. To transmit the level-ordered nodes, one could send nodes layer-by-layer (as it is done in the example: $[10], [5,15], [3,7,12,18], \ldots$), or one could group layers together to better equalize the sizes and reducing the number of batches send, which especially makes sense at the beginning, when levels are small (e.g. $[(10), (5,15)], [(3,7,12,18)]$). While each of the $l-1$ levels is obvious to de- and encode, transmitting the last layer, however, is more expensive in the (likely) case where $k>0$. This is because the information where empty leaves of the tree are stored needs to be encoded, too. This can be done by using an index list, add some skipping elements in-between blocks or by appending a bit-array indicating on which position to store a value. Alternatively, a different centering element to start the binary recursion with could be chosen, as is done in the following method.

Left aligned ordering: To convert the input to the pleasant case where the tree is completely balanced and saturated, appending virtual elements is key. In this case, $\hat{n}$ is already one less than a power of two, there's nothing to do. Otherwise, we add a virtual nodes, such that $\left(2^{l-1}-1\right)+t+a=\left(2^{l}-1\right)$, meaning the last layer is now complete. If we then perform the same algorithm as stated above, we obtain a tree with the special property that no *gap* emerges on any layer, some levels are just not filled up completely. A short proof for this as well as a more formal definition of a gap can be found in the appendix (Theorem 7, Definition 11).



(a) center-aligned level order

(b) left-aligned level order

Figure 4.6: Ordering strategies applied to the input $[1, \ldots, 20]$. Thus, $l=5$, $t=5$, because $20=\left(2^{5-1}-1\right)+5$ and $\left(2^{5-1}-1\right) \leq 20 < \left(2^{5}-1\right)$. The batches are highlighted in color.

To emphasize on the fact that the search tree serves as a concept for visualizing and understanding (while implementing, no tree has to be built), this method is called Plain Level Order Update (PLOU).

When the update step is defined in more detail below, a virtual nodes are assumed.

$$
\begin{aligned}
I_i &= [2^{l-i} + 2^{l-i+1}m | 0 \leq m < 2^{i-1}], && 1 \leq i \leq l \\
\Delta\mathcal{T}_i &= [p_j | j \in I_i \wedge 1 \leq j \leq \hat{n}], && 1 \leq i \\
\mathcal{T}^{i+1} &= \texttt{process\_update}(\mathcal{T}^i, \Delta\mathcal{T}_{i+1}) := \texttt{zip}(\mathcal{T}^i, \Delta\mathcal{T}_{i+1})
\end{aligned} \tag{4.1}
$$

The *and* condition in the definition of $\Delta\mathcal{T}_i$ assures not to include the virtual nodes, since they do not encode any useful update information. The `zip` operation merges two lists by alternately picking the next element in the list, starting with the first argument. If one list becomes empty, the algorithm stops by appending the remaining other list. An example is shown below in Figure 4.7.

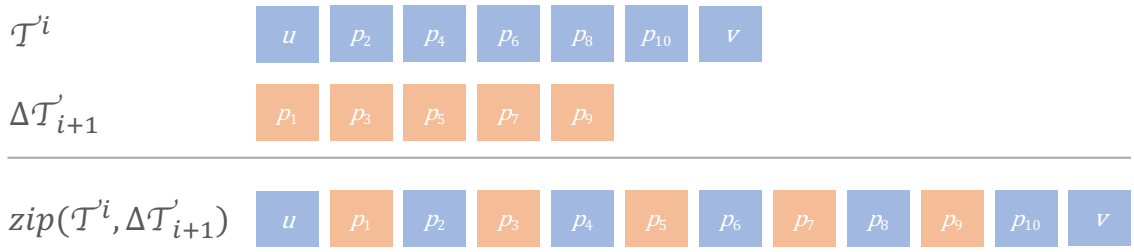
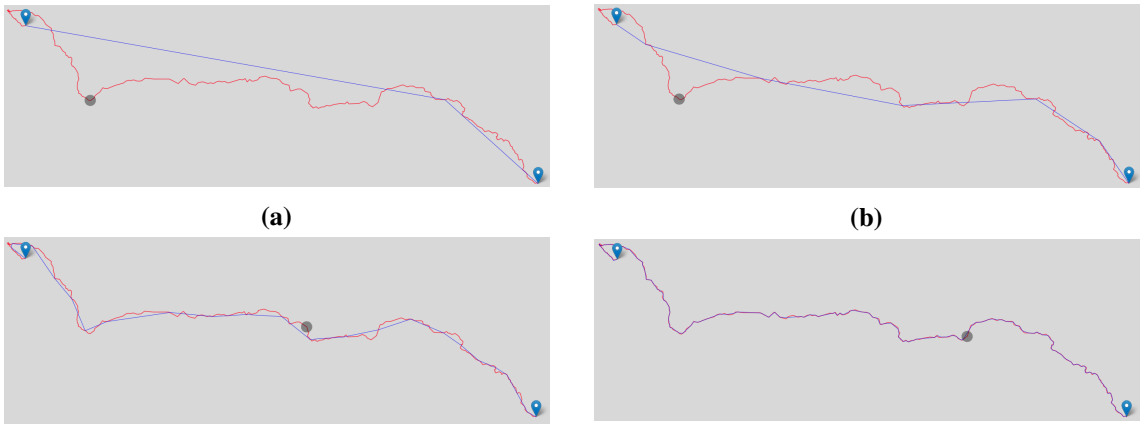


Figure 4.7: Example update processing the zip-operation.

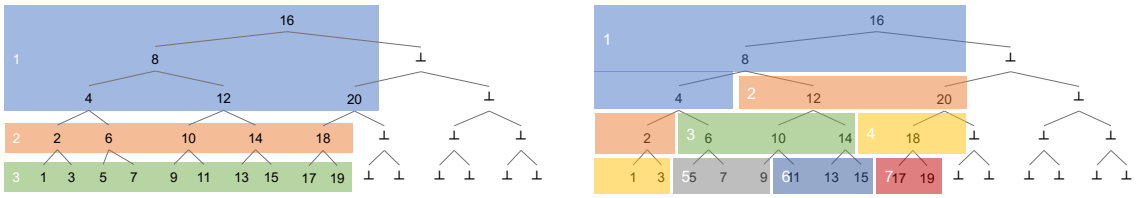
Sending nodes in level-order sequence matches a coarse-to-fine approach: Sub-sampled path points are sent first, details are added over time. Figure 4.8 shows the process with an example trajectory from the Cuba Dataset. This ordering updates edges top-down, resulting in a more global updating behavior.



(a) **(b)** **(c)** **(d)**

Figure 4.8: Example update process using PLOU batches (left aligned) applied to path #166 from the Cuba Dataset: (a) initial batch with first update, and path resulting from (b) 3, (c) 5, and (d) 8 consecutive updates.

Requesting and Transmission Using the indexing from Equation (4.1), the client only has to transmit the level i to receive the next batch, so no additional overhead compared to SOU is introduced. As the level increases, usually the number of elements also does. In the worst case, namely $t = 0$ meaning that the tree is fully saturated and balanced, the number of transmitted elements increases every request by a factor of two. To avoid having batches of fundamentally different sizes, grouping of layers becomes important. Each time a length-b batch is created, either many (for i being small) or just a few layers (for i close to l) are grouped together. Again, only the next requesting level has to be transmitted and a minimum transmission size must have been defined in advance. This idea can be further improved by specifying a fixed number of points to be transmitted: From the current level-index pair, the next b elements are retrieved. To do so, a marking has to be added to let the client know, which elements belong the which level, since not all levels are fully occupied. In terms of transmission cost, however, this only leads to an overhead of l elements in total, which is in the order of $O(\log(n))$.



(a) min-3-sized batches **(b)** fixed-sized batches

Figure 4.9: Showing an example for min-b-sized and fixed-b-sized batching. In this example, b is fixed to 3.

4.2.3 Randomized Order Update

Another sending order, named Randomized Order Update (ROU), is obtained by randomly sampling a permutation for the $\hat{n}$ remaining nodes. The random ordering also sub-samples the input list globally, because each of the remaining positions have the same chance of being picked next. This avoids point clustering as mentioned for SOU.

A downside is that the server has to maintain some kind of list of all already traversed vertices, to avoid repeating nodes – a simple counter value as with the previous shown methods does not suffice on server side. For requesting on the client side, however, storing an index is enough. Nevertheless, to correctly decode the stream of incoming nodes, some kind of meta information needs to be sent. In the easiest case, the server sends the absolute index for each node.

4.3 Unpacking-Dependent Approaches

The methods so far fully unpacked the path, ignoring its hierarchical structure of shortcut edges induced by the CH. The following methods integrate the edge-composition into traversing. To better distinguish them syntactically from the previous ones, the names are prefixed with *unpack* to emphasize the process of decompression.

4.3.1 Partially Unpacked Sequential Update

A straight-forward generalization of the method shown in Section 4.2.1, i.e., SOU, unpacks the path level-by-level to a certain depth to obtain k consecutive edges, hence $k-1$ new inner vertices. u, v and the inner vertices are then send to the client as an initial batch. For each of the k edges, each batch returns a constant number of level-order successors, until there are no vertices left. Due to similarity to SOU, it is called Partially Unpacked Sequential Order Update (PU-SOU).

As a special case the initial path gets *unpacked* only to its root-edges (no unpacking is performed), whose respective vertices are part of the initial batch. Then, a sequential iterator is used for each of the edges on server side to interactively refine the edge. To keep track of where to insert the new vertices, we also pass the inserting positions along with the current round's vertices. Since this method serves well as demonstration, it will be used later having the name Rootedge Sequential Order Update (R-SOU).

Figure Figure 4.10 shows the algorithm applied to a sample artificially generated path, which consists of five root edges.

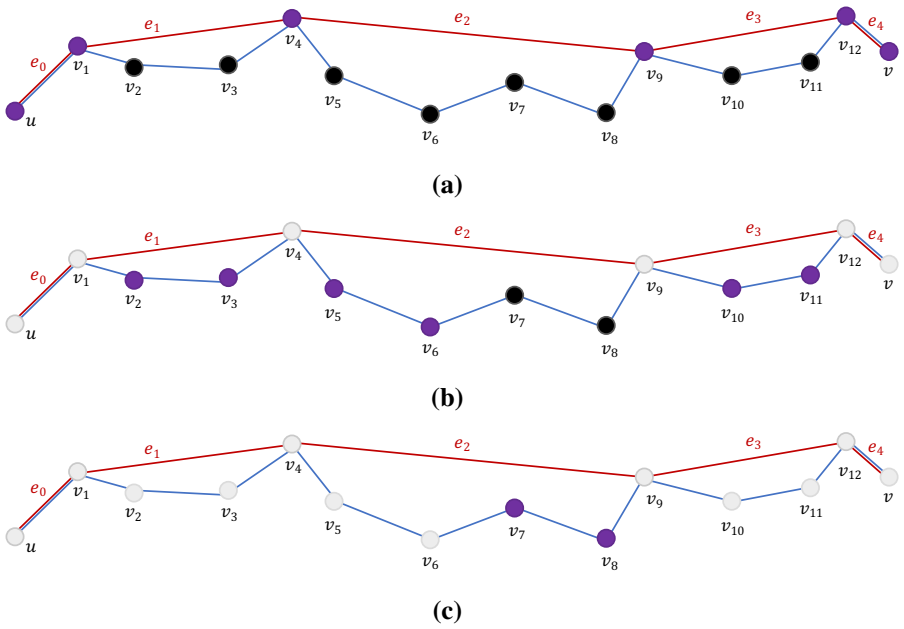


Figure 4.10: Example update process using R-SOU batches with a batch size of $b = 2$. The five root edges are colored in red, the unpacked path in blue. (a) Initial batch, (b) first, and (c) third update batch. The color codes are: black: not sent yet; magenta: part of the current batch; gray: already sent.

As part of the respond, the root-edges-vertex list $\mathcal{T}_0 = [u, v_1, v_4, v_9, v_{12}, v]$ is transmitted. Since each edge is iterated in sequential order, the next batch contains a mapping, to link the updates for each edge: $\Delta\mathcal{T}^1 = [e_1 : [v_2, v_3], e_2 : [v_5, v_6], e_3 : [v_{10}, v_{11}]]$. The final update is $\Delta\mathcal{T}^2 = [e_1 : [v_7, v_8]]$.

Because the method extends the simple sequential one by running multiple iterators in parallel, similar characteristics are inherited. In the extreme case of choosing $k = 1$, assuming there exists a shortcut edge between u and v, the methods are in fact identical.

The choice of k remains a matter of further investigation. Choosing a larger value results also in a larger initial batch, but more global improvement. A small k leads to similar locality problems as SOU. The next method adapts to CH's unpacking order.

4.3.2 Unpacking Level-Order Update

Starting from a set of root edges, unpacking yields a lower-level edge representation. When there is no more shortcut edge left, the path is fully unpacked and uses all $n-1$ edges which entail all vertices, too. Unpacking one shortcut edge results in two new (shortcut) edges and one new node connecting them. Thus, unpacking an edge gives one new vertex.

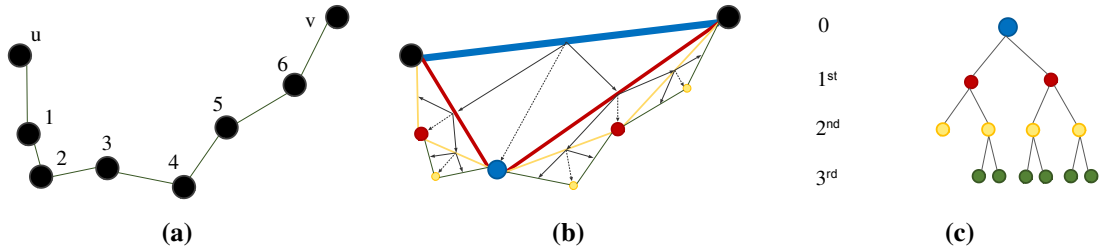


Figure 4.11: Obtaining unpacking level-order by an example path: (a) shows the path using plain edges only with sequential labeling. In (b), all shortcut edges have been added, the compressed path only consists of the (u, v)-shortcut-edge. Each shortcut edge breaks up into two edges (see solid arrows) and new vertex is obtained (dotted arrows). The unpacking yields a hierarchy (see (c)). traversing this tree in level-order outputting the linked vertices for each edge, gives the final ordering: 3,1,5,2,4,6.

To get a unique node ordering w.r.t. the unpacking procedure, the root edges have to be merged together. The details will be explained later in the scope of the proof for Lemma 2. From now on, one can assume that each path can be represented by a single (artificially created) root edge. Recursively unpacking yields the traversing order for underlying nodes. The important difference between this and all other methods is, that the final absolute position for each new node is not known to the server while unpacking: When an edge is unpacked and a new vertex emerges, it is initially unclear, how many edges will be left (or right) to it. Therefore, the methods need to encode the current level updates relative to the nodes one level before. Algorithm 4.1 shows the encoding in greater detail. Offsets are set according to the parents offset and gaps occurring on the previous layer. Transmitting the vertices according to their unpacking order is denoted as Unpacking Level Order Update (ULOU).

4.4 Simulation and Discussion

To investigate the theoretical properties in practice of the methods explained earlier, a testing environment has been set up using the Cuba dataset (see appendix B) and ´batches were compared with each other. To make the results more meaningful, very short paths of size less than ten vertices have been removed. For each method type, all paths are transmitted and the sizes for each update batch and the resulting improvements are measured.

Algorithm 4.1 Unpack-Level-Order Batching

```
1:  procedure rootEdgeToBatches(e_root)
2:      l ← 0
3:      current ← [(0, e_root)]
4:      next ← []
5:      while |current| > 0 do
6:          ordering(l) := []
7:          left_nbrs ← 0
8:          for all (offset, e) ∈ current do
9:              if e.is_shortcut then
10:                 e_1, e_2 ← e.children
11:                 next ← next ∪ [(offset + left_nbrs, e_1), (offset + left_nbrs + 1, e_2)]
12:                 v_e ← e_1.target
13:                 ordering(l) ← ordering(l) ∪ [(offset, v_e)]
14:             end if
15:             left_nbrs ← left_nbrs + 1
16:         end for
17:         current ← next
18:         next ← []
19:         l ← l + 1
20:     end while
21: end procedure
```

Accumulated Overhead

At first, the overhead introduced by the methods is analyzed. As a baseline, we transmit all uncompressed data points at once. This clearly gives a lower bound on the total transmission size. To begin with, for SOU, an appropriate value for b needs to be chosen´. The overhead behavior changes fundamentally regarding this choice: For $b = 1$ the maximum possible overhead is attained, because each vertex is requested separately. For $b \to \infty$, eventually all points fit within the first batch and the overhead becomes zero (meaning the ration gets one), as the path lengths increase. A similar behavior is expected for ROU. Simulating the overhead using SOU on the Cuba dataset for different batch sizes is shown in Figure 4.12.

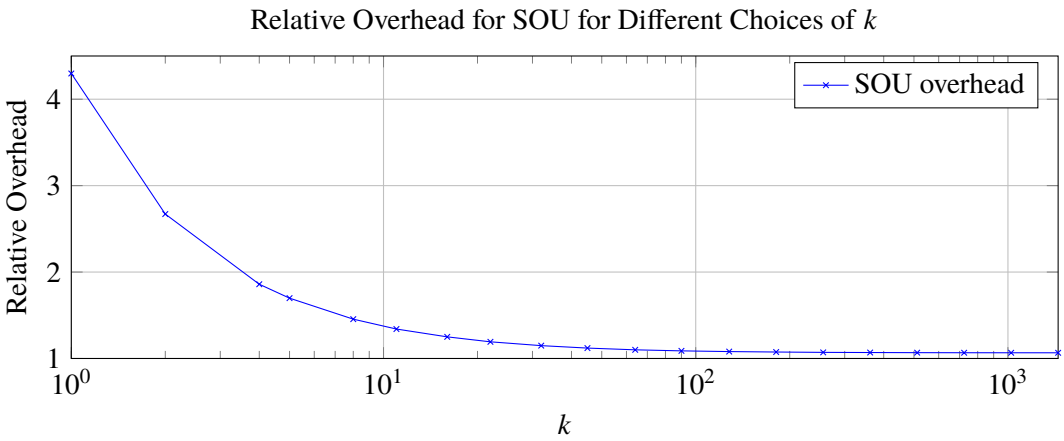


Figure 4.12: Overhead for SOU compared to baseline for different static batch sizes.

To adjust the parameter k reasonably, a fair setting can be achieved by equalizing the number of packages sent on average. The level-order method sends order $\log_2(n)$ many packages, having an average of around 360 edges, this indicates $b = \log_2(358) \approx 8.5$. Hence, $b = 10$ is used in the following, which is a reasonable approximation according to Figure 4.12.

The experimental results measuring the overhead for the different methods are shown in Figure 4.13. Regarding the overhead deviation, there is only very little variation for SOU and ROU. This originates from the fact that the number of packages for a fixed transmission size correlates with the length of the path, and therefore with the benchmark size. Level-order based methods transmit small data chunks first, improving their overhead-per-sent-node ratio over time. For small paths, the overhead is greater in relative terms. On average, the unpacking level-order introduces more overhead than its plain variant. This can be explained by the more complicated node encoding, which requires more meta information to be transmitted. Analogously, SOU outperforms the random order update. On the first glance, R-SOU introduces very few overhead. This is because on average ten percent of all edges involved in a path are root edges, meaning that ten percent of all data is sent without any batching. From that point on, each edge gets refined using $b = 10$ points, which yields very large batches in general. Sending only a few but larger badges implies, as already mentioned, less overhead but longer updating times.

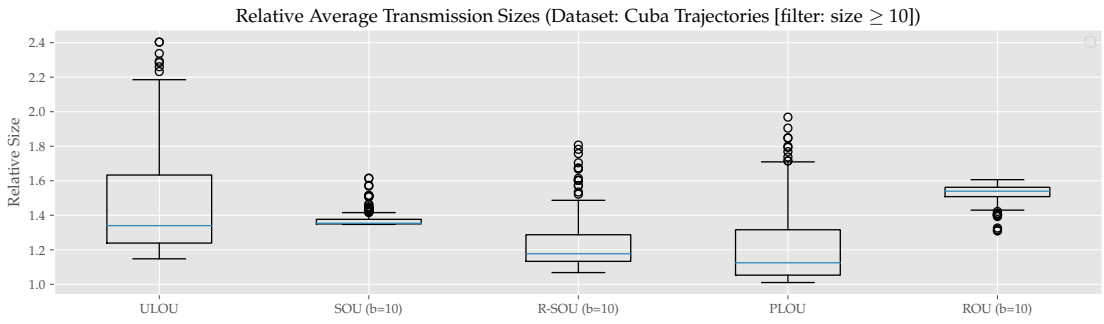


Figure 4.13: Comparison of overhead sizes for the different methods. For SOU and ROU, the batch sizes were fixed to $b = 10$. The blue markers represent the averages, the boxes separate the upper and lower quartiles, and the circles indicate outliers.

As an intermediate result it can be noted that on average no level-order method transmits more than 35% overhead for the Cuba dataset, PLOU even stays within 20%. The overheads for random and sequential order can be adjusted arbitrarily by varying k, but SOU needs slightly less overhead in general.

Error Over Update Size

So far, only the package sizes for transmitting the batches has been analyzed. However, methods to improve the visual representation fast (using few information only), are preferred – the visual quality during the path-assembly has been ignored completely yet. To measure the visual quality of an intermediate result at a given point in time, the RMOE metric from Section 4.1.3 is utilized.

To measure this time-dependent behavior, the quality is measured after each batch decoding and plotting step. In addition, the simulation keeps track of update sizes to argue about the total data sent.

To make more general arguments, it is necessary to average over all paths. The aggregation and averaging procedure explained in the following is supported by the Figure labeled 4.14. In a first step, the update sizes are normalized w.r.t the benchmark size. It is evident, that a measurement can only be taken after a batch has been sent entirely. Concludingly, the error-reduction-over-size function for each path t and each method m is given by a constant number of measurement points $s_{i,m}^{(t)}$. To average over multiple paths for a fixed method, the function $\phi_m^{(t)}$ is constructed by linearly interpolating between sampling points and rescaling the domain to $[0, 1]$. Averaging gives a meaningful aggregated function $\Phi_m(x) = \frac{1}{\#\text{paths}} \sum_t \phi_m^{(t)}(x)$. For plotting the final results, all $\Phi_m(x)$ are again sampled on an equidistant grid.

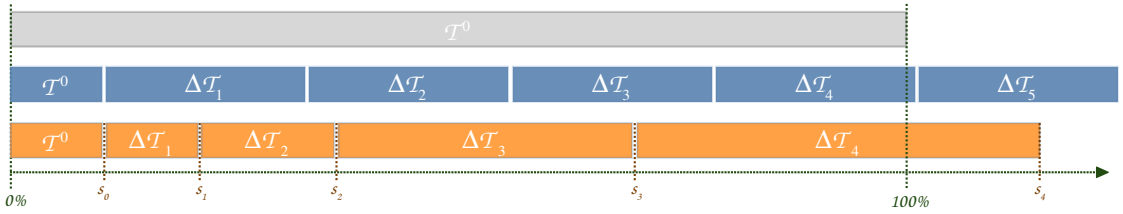
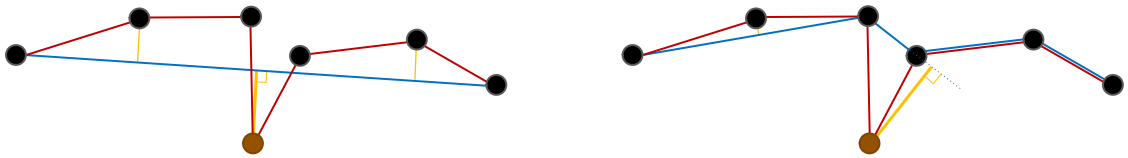


Figure 4.14: Sampling sketch explained for two methods (blue, orange). The bars indicate batches and the bar's widths represents data sizes needed for transmission. First, the sizes were determined by scaling w.r.t the baseline transmission size (gray). Each path and method yields different sampling points. For the sake of simplicity, in this example only the sampling points for the orange method have been added.

Before the simulation results are shown, an interesting remark about $\phi_m^{(t)}$ is made: While the error decreases to a final value of 0 after the last update has been transmitted, it is not guaranteed to do so monotonically, i.e,. the error can increase after applying an update batch. Using more points may shift the already well approximating rough line which results in a larger maximum distance. This phenomenon is illustrated in Figure 4.15.



(a) Before update **(b)** After update

Figure 4.15: Example to show non-monotonic quality function: In (a), an early transmission state is shown, only using the start end end point. After some updates, the blue approximation partially covers original path (red) closer, but the overall max error measurement increased, since the base line got shifted upwards.

The simulation results are shown in Figure 4.16. Obviously, none of the methods reduces its error with less data needed. The most prominent difference is the error drop shape: Different from ULOU, PLOU, and ROU, the error decreases linearly with the number of nodes being sent in case of SOU. The other methods show a steep improvement in the beginning which flattens out between 20 to 40%. After transmitting abount 60% of the data, compared to the benchmark set, they linearly decrease the error. In a direct comparison, the plain level-order transmission outperforms all other methods, especially in the early stage of transmission.

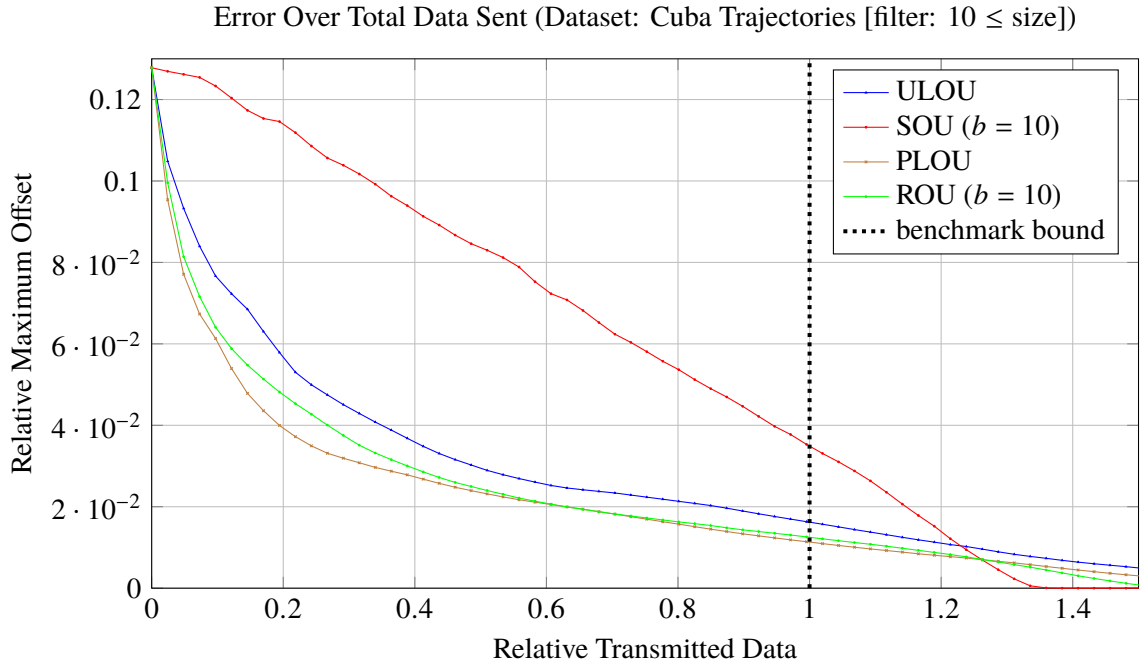


Figure 4.16: Comparing of the average error over the (normalized) number of transmitted data batches for the different methods.

Varying the value of b in case of sequential order steers the error reduction slope: Figure 4.17 shows some simulation runs for different values of b. This intuitively makes sense, since the same data stream is transmitted, but having fewer (in case of larger choices) or more overhead involved. An analogous statement applies to the ROU method (see Figure C.1): Root edges are transmitted

first, and further updates refine the gaps, where the a better performance compared to transmitting sequentially is achieved. Still, ROU performs worse than transmitting in level-order (see Figure C.2 for more details). Since the method transmits root edges at the beginning, it inherits its properties. In a later transmission state, ROU resembles SOU, because edges are reminded one after another.

Only taking the quality of a transmission process based on a fixed implementation into account might miss some important details. There might be more efficient encodings or better compression schemes. To abstract from the underlying implementations and encoding methods, the orderings are directly compared with each other: Firstly, transmit the batches and keep track of the order in which the methods transmit the nodes. Secondly, recalculate the offset errors, but ignore the transmission size and only include the number of vertices which has been transmitted already. The results are shown in Figure 4.18. Interestingly, the overall shapes mach the previous results. Choosing nodes in plain level-order still outperforms all other methods, even though expensive encoding costs for ULOU and ROU are ignored.

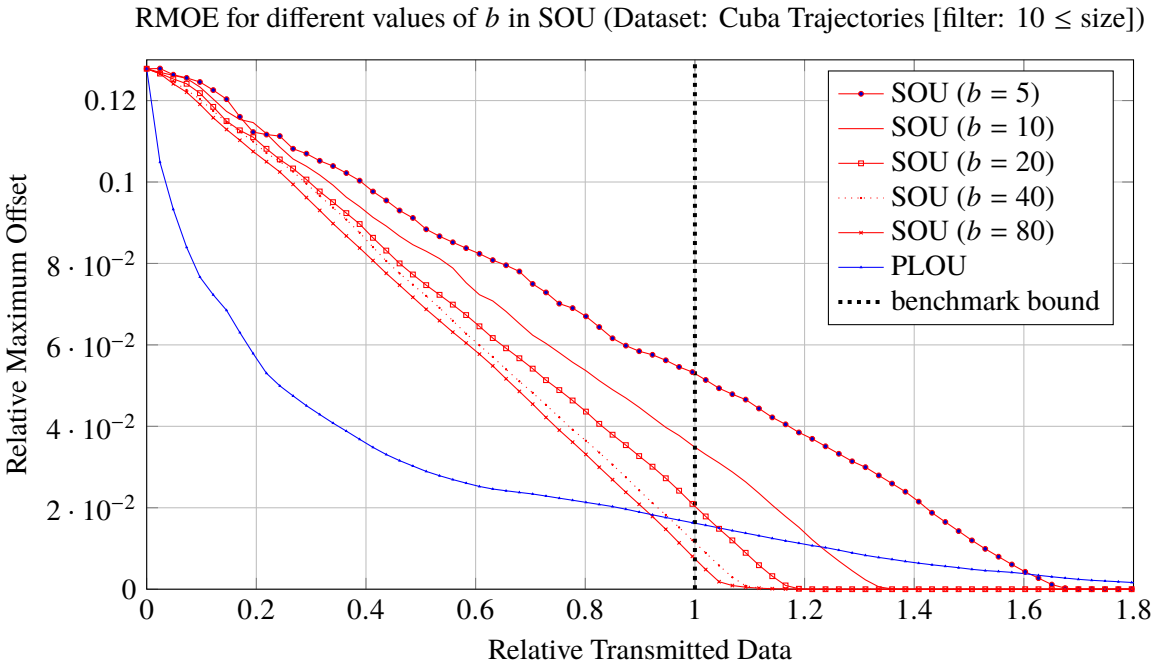


Figure 4.17: Plot showing the average errors over the (normalized) number of transmitted data batches for different values of b and PLOU for reference.

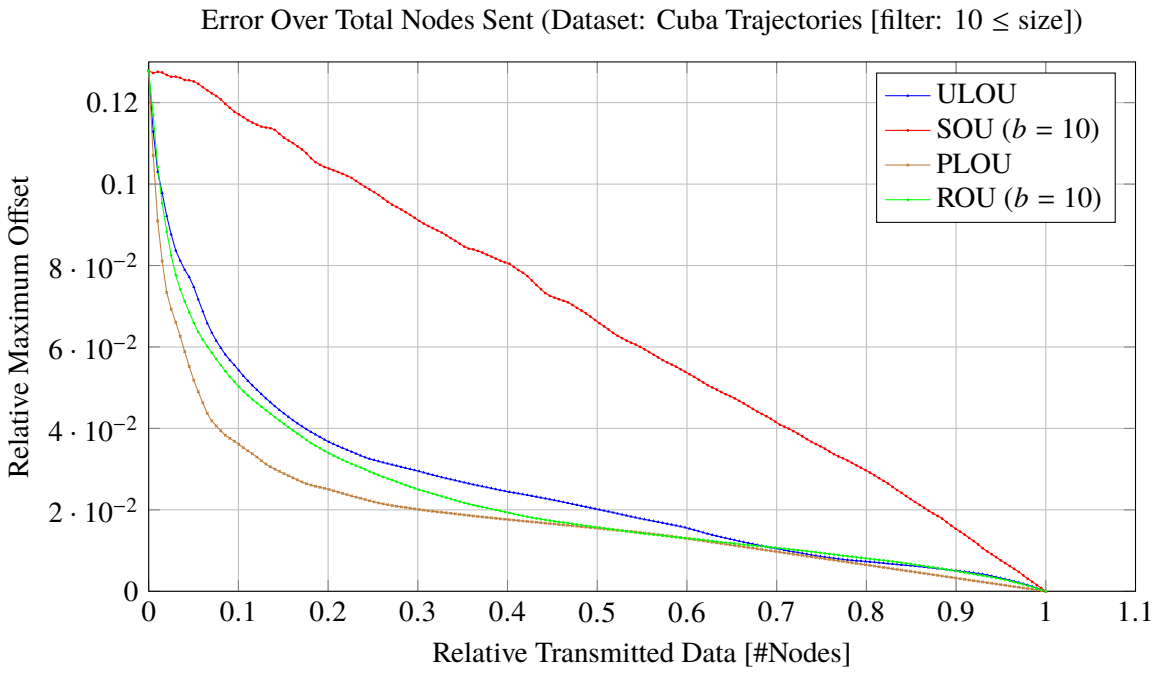


Figure 4.18: Plots showing the average errors over the (normalized) number of transmitted nodes for the different methods.

Summary

In this chapter some basic algorithms for transmitting trajectories over a network were presented, as well as the corresponding implementation architecture. The first section dealt with CH independent implementations, in the second part, the structure of unpacking paths was utilized. To argue about their performance, a simulation has been implemented and different use cases were tested using the Cuba dataset. Clearly, the presented methods only show a subset of the $n!$ possible batch-ordering types.

The simulations confirmed the earlier stated property that the batch size parameter negatively correlates with overhead being generated by the transmission process, as well as for de-, and encoding. Moreover, transmitting a small amount of data already approximates the true trajectory quite well if the method chooses points globally. This means, it avoids choosing points close to each other. The average overhead differs from method to method, but generally stays within an acceptable range of 10-60%. Therefore it is reasonable to transmit 20-40% of the data globally in a batched fashion to create good results early in the transmission having only a moderate amount of overhead.

The best method, namely PLOU, chooses points level-order-wise induced by a left-aligned search tree ordering.

All methods were designed in a way such that each node or edge can be accessed in constant time, especially unpacking is required for most of the methods. Also, all simulations are based on the fact that all points are transferred. For practical usage, however, none of the two assumptions apply. Still, the architecture could be reused easily for further batching tasks, by simply extending the parser and encoder modules.

An example transmission for the methods PLOU and SOU is accessible at [7] as videos.

5 Graph Layer-Pruning

The methods discussed so far work on a list of either partially or fully unpacked paths. For larger zoom scales, however, transmitting or even partially unpacking all requested edges is not possible given the usual waiting time restrictions. Edgesets responses which are required to produce outputs similar to Figure 5.1 for example would need multiple megabytes in size and intense unpacking.

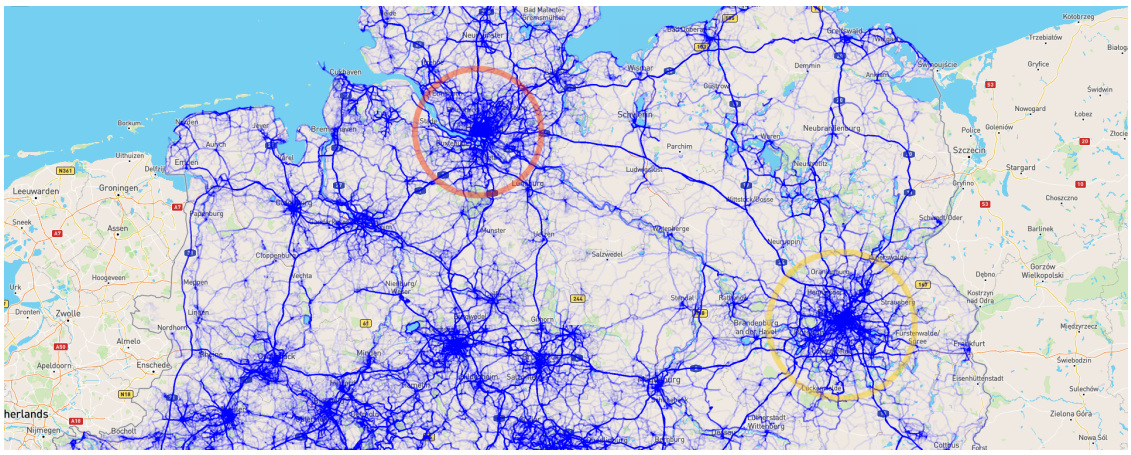


Figure 5.1: An example showing a high-resolution result on a large zoom scale. Markings of the cities Berlin (yellow) and Hamburg (red) has been added externally.

By requesting map sections on such large zoom scales, it would be sufficient to plot the most important trajectory parts only and replace high-detail information by some abstraction. In the example map above, it might be interesting to see the connections between the two highlighted cities even on high zoom levels, but hide very detailed trajectory sections in the respective downtowns. Therefore, some kind of edge selection and information reduction has to be applied. The following method called Graph Layer-Pruning (GLP) categorizes the response set into either relevant or dispensable parts and returns a approximate edgeset and an accumulated low-level reduction layer respectively.

5.1 Implementation

Plotting a set of trajectories comes down to visualizing a set of edges. Each of the edges has an associated level, which originated from the CH construction process. The larger the level, the later the edge has been created. This in turn means, that the edge is more abstract because it groups together multiple subparts. Hence, In this section, we leverage the edge level as a measure of importance.

5.1.1 High-/Low-Level Split

The input is, analogous to Section 3.2.1, a list of compressed trajectories $\mathcal{T} = [t_1, t_2, \ldots, t_r]$ given by the output of the PF algorithm. In a very first step, an iterator visits all root edges $e_j^{(i)}$ for all input trajectories t_i and decides if the edge will be kept for or rejected from refining. The decision is guided by a user predefined cutoff parameter l_{cut}. As a result, two sets of edges emerge:

$$\begin{aligned} E^+ &= \left\{ e_j^{(i)} \in t_i, t_i \in \mathcal{T} | level(e_j^{(i)}) \geq l_{cut} \right\} \\ E^- &= \left\{ e_j^{(i)} \in t_i, t_i \in \mathcal{T} | level(e_j^{(i)}) < l_{cut} \right\} \end{aligned} \tag{5.1}$$

Note, that duplicate edges have been removed already, which required linear time. l_{cut} is chosen in such a way, that E^+ becomes small enough to work with. This usually requires l_{cut} to be significantly larger that the user's preferred unpacking target level $l_\sqcap$ (see Section 2.1 for definition). In a second step, these edges are unpacked next. This is necessary, because in general, edges with very high levels don't approximate their unpacked version well anymore. Figure 5.2 illustrates this claim for edges above $l_{cut} = 75$.

(a) Edges E^+ **(b)** Edges E^+ after unpacking to $l_\sqcap$

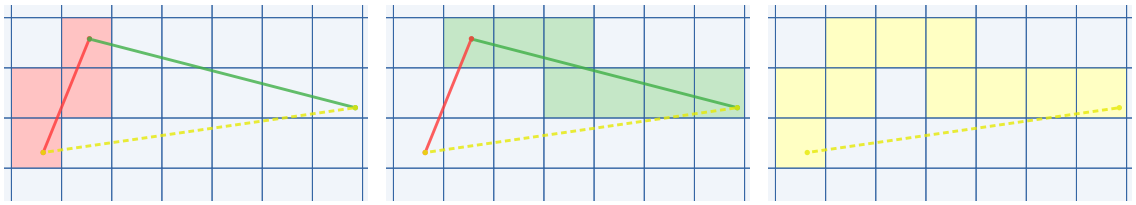
Figure 5.2: Naively returning the filtered edges E^+, as depicted in (a) does not give meaningful results.

So far, a subset of important result edges got extracted and *pushed downwards* to enhance quality to a satisfactory level. When only filtering the edges, the retrieved information from the lower level edges would get lost. For integrating E^- while keeping the aggregated output small, a heatmap is created. A simple approach would extract all nodes from the fully unpacked edge in E^- to build a heatmap. Clearly, this operation would be far too timeconsuming. Instead, calculations for building the heatmap will be made during a preprocessing phase on startup, which will be explained next.

5.1.2 Heatmap Grid Hierarchy

To quickly generate a heatmap given a set of (potentially high-level) edges, a global grid is introduced. On startup, a $m \times n$ sized grid is defined, where $n = 2^q$ and $m = 2^p$ are powers of two ($p, q \in \mathbb{N}$). The grid boundary matches the bounding box of the graph exactly. Given the bounding box and

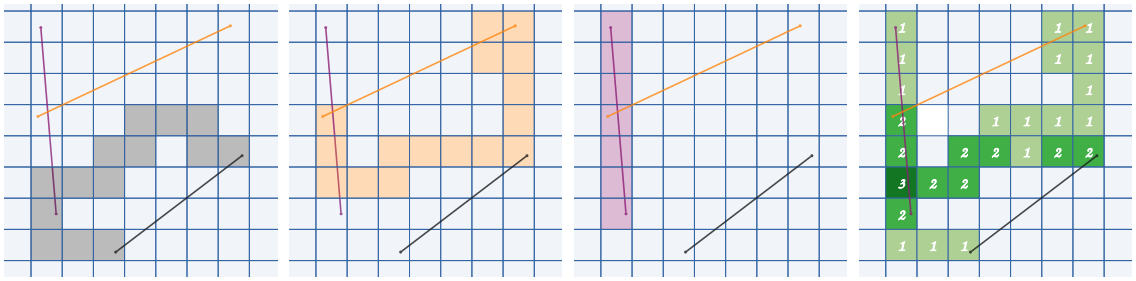
the grid width and height parameters, each cell $c_{i,j}$ $(0 \leq i < m, 0 \leq j < n)$ is defined uniquely. A list stores the intersected cells for every edge. The marking process can be calculated recursively: On a shortcut edge the list is obtained by merging the cell entries for both children. The cells of a plain edge is obtained by the retrieval of all intersected cell boundaries. An example is shown in Figure 5.3.



(a) Markings of e_1 **(b)** Markings of e_2 **(c)** Markings of e

Figure 5.3: Showing the iterative merge process of shortcut edge e (yellow) having the children e_1 (red) and e_2 (green).

When the heatmap for a set of edges has to be calculated, all edge-lists are merged, but a count for each cell is maintained. This allows to quantify the region-based usages. An example is shown in Figure 5.4. Note that for plain edges, typically only a few marked cells have to be stored.



(a) Markings of e_1 **(b)** Markings of e_2 **(c)** Markings of e_3 **(d)** Final grid

Figure 5.4: Showing the process of building the heatmap on the (shortcut) edges input list $E^- = [e_1, e_2, e_3]$.

To get reasonable results, the heatmap must be fine-scaled by choosing moderately large values for p and q, depending on the graph's size. This, however, results in a growth of the response size, because $g = 2^p \cdot 2^q = 2^{p+q}$ counter values have to be transmitted in the worst case. Especially for larger zoom scales, such a fine grained grid is not required. To downsample the response quickly, each edge already stores its downsamled versions. This ensures fast merging times, since counting takes place on the donwsampled versions, too. By grouping exactly 2×2 cells together, in theory $r = \min(p, q) - 1$ sub-sampled versions can be precalculated, until only one row or column (for $p = q$ a single cell) remains.

This gives us a pyramid-shaped grid hierarchy, similar to a Gaussian Pyramid[56], but using an *or* operator on boolean values. Hence, by storing the complete hierarchy, the total number of additional cells is upperbounded by $\sum_{l=1}^{r}(\frac{2^p}{2^l} \cdot \frac{2^q}{2^l}) = \sum_{l=1}^{r} \frac{g}{4^l} = \left(\sum_{l=1}^{r} \left(\frac{1}{4}\right)^l\right) \cdot g \leq \frac{1}{3} \cdot g$. Depending on the requested zoom level, the service choses a suitable hierarchy level to perform the merge on and returns the heatmap.

5.1.3 Running times

Algorithm 5.1 shows the preprocessing on a rather high level. The hierarchy, consisting of $r+1$ layers is built iteratively. For the initial layer, an empty grid for each edge is initializes which takes $O(|E_{G_{CH}}| \cdot g)$ time. For each of the r following layers, sub-sampling based on the previous one is performed, which traverses each of the recently created edges-indices and assigns a new index. Duplicated indices are ignored. An example for re-indexing in shown in Figure 5.5. In the actual implementation, no tuple is stored. Instead, a global grid index idx (integer) is used, but the conversion can be done in constant time:

$$\begin{aligned} j &= idx \mod n \\ i &= (idx - j)/n \end{aligned} \qquad idx = i \cdot n + j \tag{5.2}$$

Theorem 2 *GLP's preprocessing takes $O\left(g \cdot |E_{G_{CH}}|\right)$ time.*

Proof In the worst case, $O(|E_{G_{CH}}| \cdot g)$ steps are required to build $\mathcal{L}_1$, but its size is upperbounded by $\frac{1}{4} \cdot |E_{G_{CH}}| \cdot g$. The latter term again (asymptotically) upperbounds the running time for building $\mathcal{L}_2$. More abstractly spoken, the size of the previous layer asymptotically upperbounds the running time of the current, meaning that the total running time is upperbounded by the total size which is. Hence, preprocessing take $O\left((1+\frac{1}{3})g \cdot |E_{G_{CH}}|\right) = O\left(g \cdot |E_{G_{CH}}|\right)$. ∎

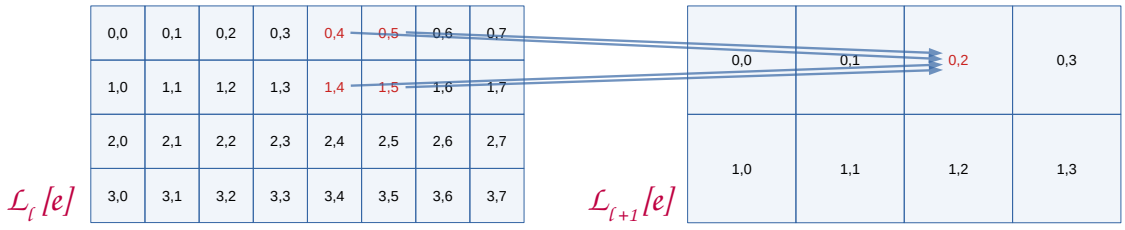


Figure 5.5: Showing the re-indexing step on a 4×8 sized input grid.

A high level sketch for exporting the pruned results including the heatmap are sketched in Algorithm 5.2. The input consists of the set of root edges returned by PF E, a cutoff threshold l_{cut} a request zoom level z, a target unpacking level $l_{\sqcap}$ and a view request box V.

Splitting the input set takes linear time in $|E|$, each edge is visited once. Unpacking is output sensitive and takes at most $O(|E_{ex}|)$. Mapping the input zoom size to a hierarchy level is up to a user-defined function, but constant in time. Creating the heatmap needs to define the boundary and grid sizes, which can be obtained by the original cell size, modified based on the current level

Algorithm 5.1 Graph Pruning: preprocessing

procedure BUILDHIERARCHY($p, q, E_{G_{CH}}$)
  $r \leftarrow \min(p, q) - 1$
  $\mathcal{L}_0 \leftarrow$ INITEMPTYLAYERGRID($2^p, 2^q$)
  for all $e \in E_{G_{CH}}$ **do**
    $\mathcal{L}_0[e]$ = GETINTERSECTINGCELLS($e, \mathcal{L}_0, G_{CH}$)
  end for
  for $l \in 1, \ldots, r$ **do**
    $\mathcal{L}_i \leftarrow$ SUBSAMPLELAYER($\mathcal{L}_{l-1}$)
  end for
  $\mathcal{H} \leftarrow \mathcal{L}_0, \mathcal{L}_1, \ldots, \mathcal{L}_r$
  return $\mathcal{H}$
end procedure

procedure SUBSAMPLELAYER($\mathcal{L}_l$)
  $n_l, m_l \leftarrow \mathcal{L}_l$.size
  $\mathcal{L}_{l+1} \leftarrow$ INITEMPTYLAYERGRID($\frac{m_l}{2}, \frac{n_l}{2}$)
  for all $e \in E_{G_{CH}}$ **do**
    for all $(i, j) \in \mathcal{L}_l[e]$ **do**
      $\mathcal{L}_{l+1}[e] := \mathcal{L}_{l+1}[e] \cup (\lfloor i/2 \rfloor, \lfloor j/2 \rfloor)$
    end for
    $\mathcal{L}_{l+1}[e] \leftarrow$ REMOVEDUPLICATES($\mathcal{L}_{l+1}[e]$)
  end for
  $\mathcal{H} \leftarrow \mathcal{L}_0, \mathcal{L}_1, \ldots, \mathcal{L}_r$
  return $\mathcal{H}$
end procedure

hierarchy l. Still, the initialization can be done in constant time, because no edge has been inserted yet. For each edge in E^-, counts are updated by going through all linked cell entries. This are at most $g/4^l$ cells each. Hence, $O(|E^-| \cdot \frac{g}{4^l} + |E_{ex}|)$ time is needed in total. Converting the result into a text representation clearly scales linearly in the conversion input, which does not change the running time's asymptotic behavior.

In addition, not all edges are reported, but only those which are visible by the requesting view V. A decision is made by checking each edge's bounding box for intersection with V while unpacking is performed. Whenever the view changes, i.e., after zoom- or panning-operations, a new request is generated, while the last edgeset is stored in cache for faster follow-up requests. This ensures quick response times since only actually visible edges are transmitted each time.

5.2 Discussion

In the following, the pruning process is examined from various points of view, including a use case analysis, run time measurements and method limitations. To start with, typical results are shown and explained in the following section.

Algorithm 5.2 Graph Pruning: exporting

```
1: procedure exportPrunedGraphAndHeatmap(E, l_cut, z, l_⊓, V)
2:     E⁺, E⁻ ← splitEdgeset(E, l_cut)
3:     E_ex ← []                                          // Unpack highlevel
4:     for e ∈ E⁺ do
5:         E_ex ← E_ex ∪ unpack(e, l_⊓)
6:     end for
7:     l ← chooseHierarchyLayerLevelByZoom(z)
8:     H ← createHeatmapByLayer(L_l, E⁻)
9:     return exportEdgelistAndHeatmapToJSON(E_ex, H, l_⊓, V)
10: end procedure
1: procedure createHeatmapOnLayer(L, E⁻)
2:     L ← initEmptyCountingGrid(L)
3:     for e ∈ E⁻ do
4:         L.incrementCounts(L, L[e])
5:     end for
6:     H ← normalizeCounts(L)
7:     return H
8: end procedure
```

5.2.1 Implementation

Figure 5.6 shows typical results using GLP on various graph and path data. Strokes colored in black indicate edges of high level from E^+ which passed the edge filtering and got partially unpacked depending on the choice of $l_\sqcap$. Low-level edges below l_{cut} have been pruned away and replaced by a heatmap. According to the density and weights, areas are colored on a blue to red scale, indicating low and high trajectory densities respectively. Typically most high-density areas are within the request box. This is because only trajectories passing this request box have been retrieved and further intersections become less likely as trajectories are of limited length.

5.2.2 Measurements

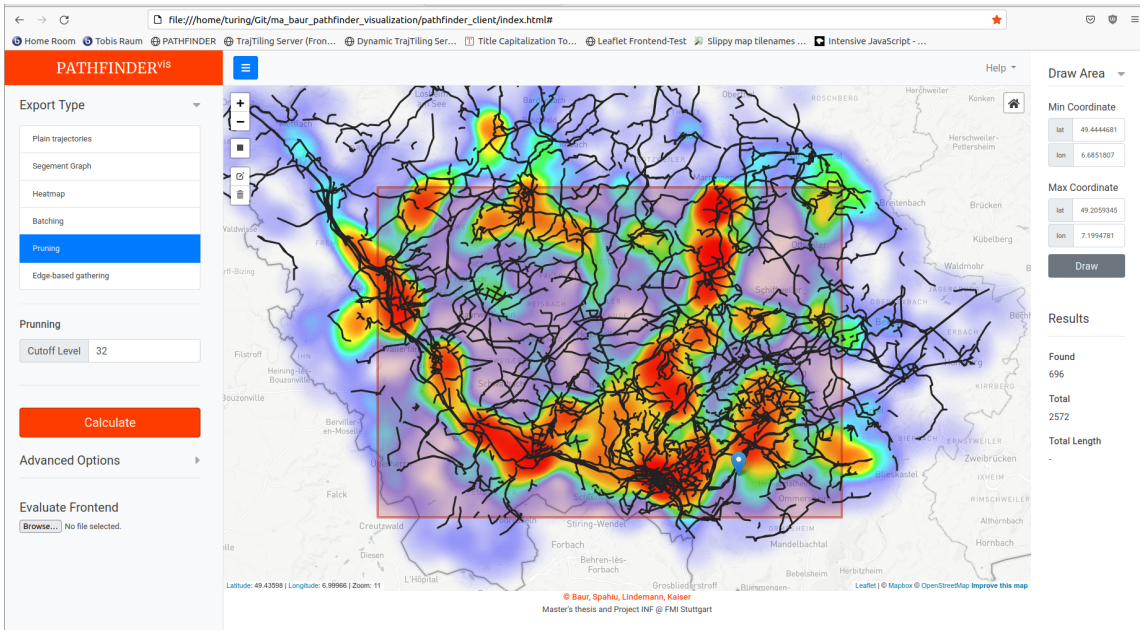
For evaluation, there are many degrees of freedom steering the calculation times, response quality, and size:

Cutoff level l_{cut}: influences the filtering process, the larger the value the more edges contribute to the heatmap and the smaller the value the more edges are kept.

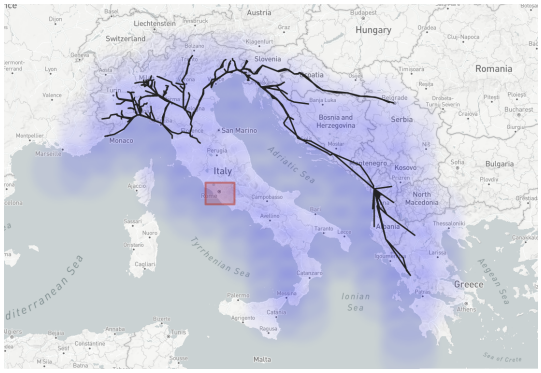
Unpacking target level $l_\sqcap$: parameter defining the quality for kept edges. The smaller $l_\sqcap$, the more unpacking is performed, increasing both the response size and quality.

Request bounding box Q_s : determines area and size of the request. Larger request bounding boxes typically yield more matches, resulting in higher processing times and return sizes.

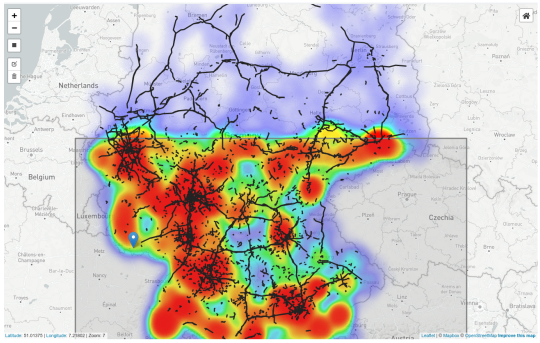
Request view box V: another bounding box containing the description of the view displayed to the user. The behavior is similar to Q_s.



(a)



(b)



(c)

Figure 5.6: Typical GLP outputs on different graph and path data: (a) depicts a response on the Saarland Dataset, (b) a request around Rome on the Europe500K Dataset, and (c) the southern part of Germany on the dataset of the same name.

Zoom level z: in addition to the view box V, the zoom level z describes the user's map zoom state. For larger zoom scales, usually more detailed are desired.

Grid size powers p and q: define the quality and resolution of low-level edges' E^- heatmap grid. The larger the choice for p and q, the finer the are result-heatmps, at the cost of higher merging times and larger transmission packages.

Naively iterating through the whole parameter space is unreasonable because not all combinations make sense or are feasible to calculate. Hence, for evaluation, the following simplifications are made: The request box Q_s equals the view box V, i.e., the user is always interested in the parts visible on the current screen. The size of the request boxes Q_s are defined similarly as described in Section 3.4.2 using the factors 2^{-i} with $i \in \{1, 2, 3, 4, 5\}$. Since the request box and the screen view

	Saarland					Germany				
request size factor	1/32	1/16	1/8	1/4	1/2	1/32	1/16	1/8	1/4	1/2
corresponding zoom level z	15	14	13	12	11	11	10	9	8	7

Table 5.1: Mapping from the size of request box Q_s, determined by the graph's bounding box and the size factor, to the zoom level used for validation.

are related, not all zoom scales are possible any longer. Therefore, each request bounding box comes with a respective zoom resolution determined by manually plotting bounding boxes on the screen and adjusting the zoom accordingly. The mapping is given by Table 5.1. If not stated differently, the unpacking level l_{cut} is derived from the request zoom z as it has been implemented in the PATHFINDER$^{\text{web}}$ project and explained in Section 2.5. The influence of the grid size is investigated later in this section, but will be fixed to $(p, q) = (6, 6)$ for now.

Given this parameter sets, requests have been send using both the Saarland and the Germany Dataset, the results are presented in Table 5.2 and Table 5.3 respectively. For each measurement, 20 boxes were sampled at random and timings and sizes were averaged. In case of very large requests where calculations on the backend side already exceed a threshold of five seconds, sampling was stopped and averaging of the results sampled so far was performed. Those special cases are highlighted using bold font. Frontend validation and Saarland backend processing was done using the ThinkPad, backend processing for the Europe and Germany Dataset ran on Threadripper.

	Cutoff level l_{cut}					
	128	64	32	16	8	4
Factor $1/32$						
Frontend [ms]	11	11	13	10	10	10
Backend [ms]	2	2	1	2	4	2
Size [KB]	1	5	3	7	19	9
Factor $1/16$						
Frontend [ms]	15	14	12	12	11	12
Backend [ms]	2	2	5	4	4	4
Size [KB]	1	6	21	20	21	19
Factor $1/8$						
Frontend [ms]	18	12	12	12	11	10
Backend [ms]	4	9	12	11	9	24
Size [KB]	4	38	63	59	58	141
Factor $1/4$						
Frontend [ms]	25	22	18	16	14	14
Backend [ms]	11	22	29	41	40	46
Size [KB]	10	92	159	251	271	341
Factor $1/2$						
Frontend [ms]	51	46	49	31	20	13
Backend [ms]	29	47	70	101	97	111
Size [KB]	27	203	441	768	794	955

Table 5.2: Graph Pruning measurement: Saarland Dataset.

	Cutoff level l_{cut}				
	512	256	128	64	32
Factor $1/32$					
Frontend [ms]	37	27	15	12	11
Backend [ms]	34	31	31	61	54
Size [KB]	2	34	131	442	512
Factor $1/16$					
Frontend [ms]	128	88	52	20	9
Backend [ms]	107	101	140	174	230
Size [KB]	6	208	886	1512	2334
Factor $1/8$					
Frontend [ms]	493	228	108	45	9
Backend [ms]	223	289	340	482	835
Size [KB]	22	595	1882	3939	8465
Factor $1/4$					
Frontend [ms]	405	190	95	40	19
Backend [ms]	794	801	799	1394	1916
Size [KB]	13	933	3187	10287	18442
Factor $1/2$					
Frontend [ms]	2877	1145	**503**	**106**	**17**
Backend [ms]	2196	2374	**3202**	**3734**	**7661**
Size [KB]	55	2176	**10778**	**24657**	**67369**

Table 5.3: Graph Pruning measurement: Germany Dataset.

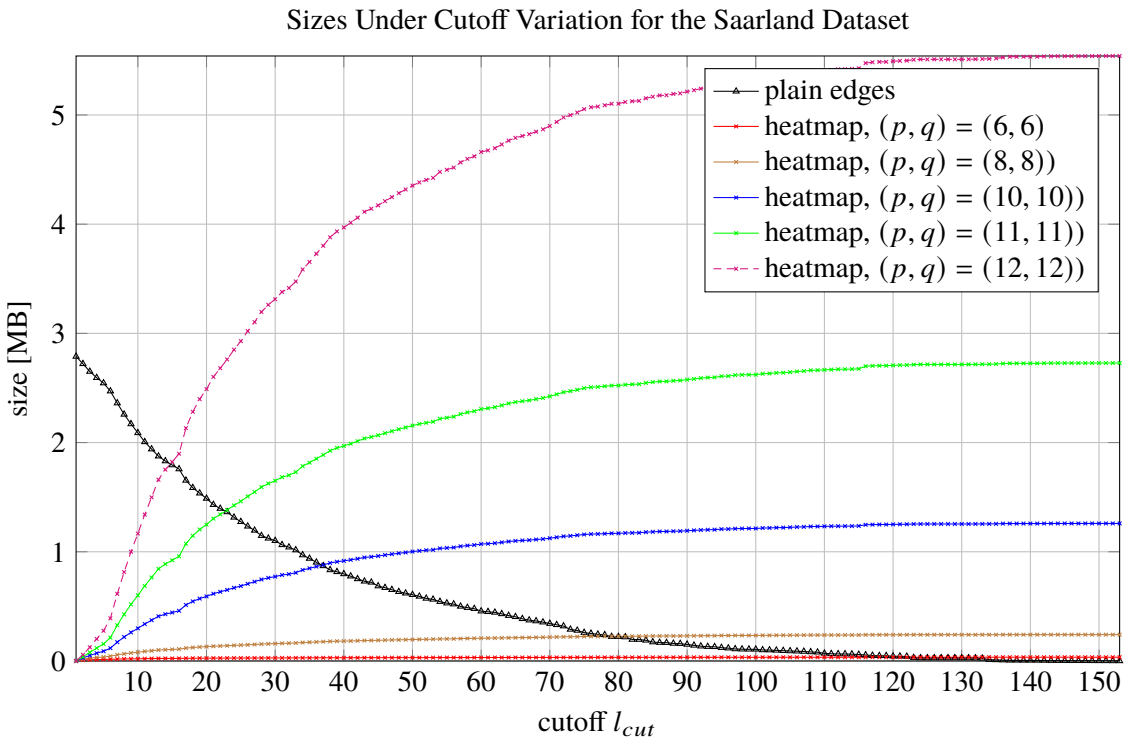


Figure 5.7: Varying the cutoff on different grid resolutions.

Some of the theoretical properties are confirmed by the measurements, most importantly the negative correlation between the factor size and the response sizes and calculation times. In general, the same holds true for the cutoff level in relation to the package size, except from very few outliers for rather small request boxes.

The measurements indicate that pruning performs very well on the smaller Saarland Dataset, but still acceptable for the Germany one if the request size does not exceed $1/4$th of the graph's bounding box original scale and a cutoff of 64.

The data shows an interesting relation between front- and backend times: For a fixed bounding box size, client- and server-workload measured by time influence each other reciprocally: The larger the cutoff was chosen, the faster the backend finished work, but the longer the frontend needs to display the results and vice versa. This can be explained due to the fact that the backend edge aggregation is slower than unpacking edges, but results in smaller responses which are faster to visualize on frontend side.

For further investigation, the whole Saarland Dataset was requested for all possible cutoff levels ranging between 1 and $l_{max} = 153$ under variation of the grid size. The result is shown in Figure 5.7.

Regardless of the heatmap resolution, in all cases the edge proportion decreases and the heatmap sizes increases for letting l_{cut} grow. All heatmap functions have in common that they eventually saturate and there is no influence when increasing the cutoff size further. This can be explained by the heatmap grid: eventually all cells are transmitted and from that point on, only the counts change The latter procedure does not influence the heatmap size. For to large choices of p and q,

dataset	Saarland						Germany	Europe
	100k	1M	real paths					
grid	$2^6 \times 2^6$			$2^{12} \times 2^{12}$	$2^{14} \times 2^{14}$	$2^{16} \times 2^{16}$	$2^6 \times 2^6$	$2^7 \times 2^7$
time [min]	0:14	0:21	0:05	0:06	0:11	0:47	2:05	8:38

Table 5.4: Recorded preprocessing times: edge hierarchy calculation for various graph types and path inputs. On all graphs, the first four layers have been build. The Saarland measurements were taken on the ThinkPad, the Europe and Germany ones on the Threadripper.

the transmission of heatmap data even exceeds the size for transmitting all edges, meaning that the encoding of low level edges using heatmaps is more expensive than transmitting the edges themselves.

The determination of good heatmap size parameters turns out to be subtle: Choosing to large (p, q)-values results in high-resolution heatmaps at expensive costs. To small values either produce dotted lattice-based patterns or blurry outputs, even for regions including very few trajectories. Examples are shwon in Figure 5.8a and Figure 5.8b accordingly.

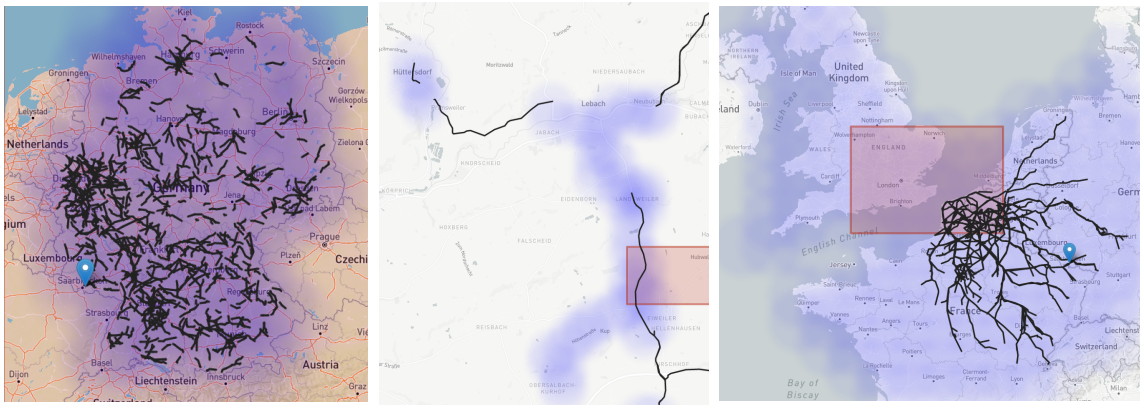
(a) **(b)**

Figure 5.8: Responses indicating that p and q were not chosen large enough. In (a), single grid points are visible while in (b) traversed streets are blurry.

Increasing the grid size does not only slow down the request processing time because more cells have to be merged, but also contributes negatively to preprocessing. Table 5.4 summarizes typical setup times for graphs of different size and varies the grid size in case of the Saarland Dataset.

Utilizing the CH-graph levels' as a measure of edge-importance allows for constant threshold check and usually returns satisfactory results, especially in city regions: While main streets and routes connecting city districts are often part of the shortest paths and therefore gets removed late yielding high levels, dead ends and paths, which are difficult to reach, are removed early. A concrete example will be shown later when this chapter is summarized. Nevertheless, pruning based on a fixed level is not optimal, especially for large cutoff levels. A real world data path consists of multiple root edges of different level. During threshold filtering, paths are not necessarily kept as a

whole. In the optimal case, small end segments are pruned away and the main routes remain. The counterexamples in Figure 5.9a and Figure 5.9b show, however, that the output can have unpleasant edge holes. In case of Figure 5.9b, no useful insight is gained.



(a) **(b)** **(c)**

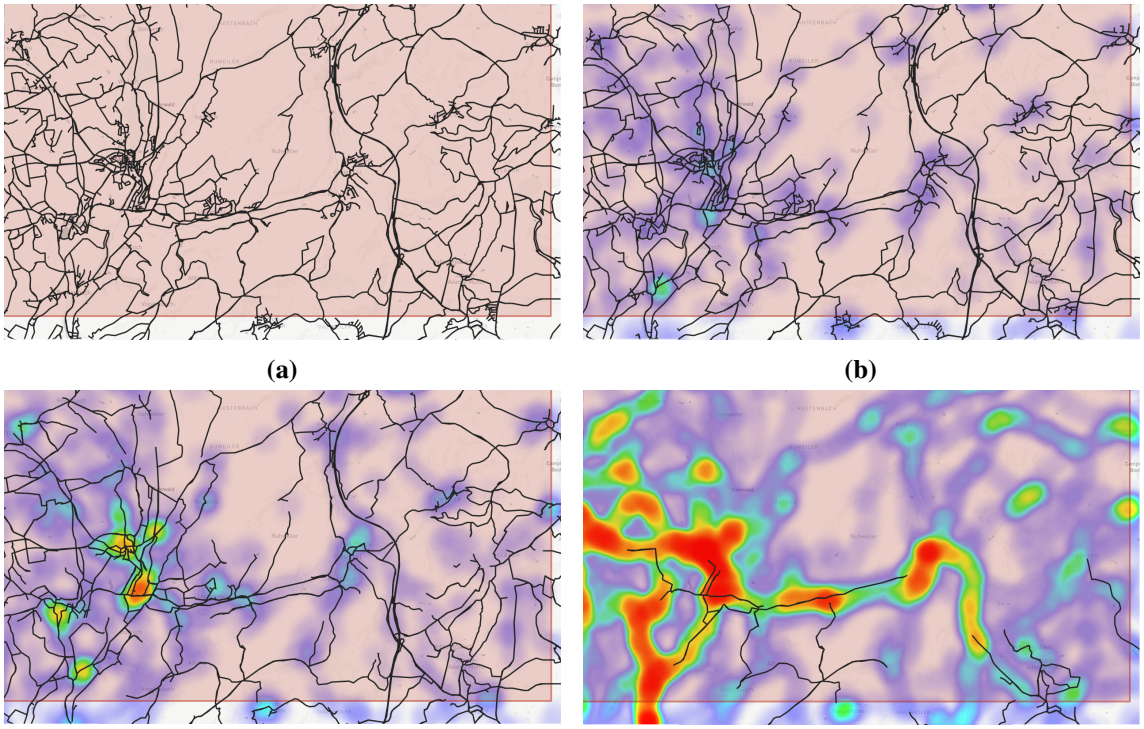
Figure 5.9: Examples showing limitations of GLP.

Another limitation is that graph-connected components differ in the number of nodes. If fewer nodes are given, fewer contraction operations have to be performed during CH construction. Hence, important edges on a smaller graph or on a independent sub-graph have lower levels than important edges of graphs differing in size by multiple factors. In the specific scenario of requesting the Europe500K Dataset, for example, islands' important streets have low levels compared to important streets on the Europe mainland. An example for unpleasant results obtained when requesting independent components is shown in Figure 5.9c: The UK's sub graph is much smaller, hence, important routes are only visible on smaller cutoff levels.

The presented method returns very fast results on small datasets or request boxes and nicely emphasizes trajectory-hotspots and their connections. GLP is ideally suited for inter-city analysis and partially applicable to macroscopic tasks. Microscopic analysis tasks are not well supported, since no trajectory-related information is linked with the edges.

5.2.3 Summary

Graph-pruning based edge filtering enables the highlighting of important edge strokes and the appealingly coloring of less meaningful graph parts at the same time. While it is not clear yet how to automate the finding of an optimal grid size choices, once manually specified, informative responses are obtained. The static edge filtering may causes problems when multiple graph components exist and further postprocessing for avoiding unpleasant gaps is needed. The quasi-continuous filtering parameter can be utilized to optimize the output with respect to the user's intentions and inter-city analysis tasks are perfectly covered. Concludingly, the dynamic of varying l_{cut} is outlined in Figure 5.10.



(a) **(b)**

(c) **(d)**

Figure 5.10: Varying l_{cut} results in a quasi-continuous filtering scheme. In this examples based on the Saarland Dataset. In (a), no filtering was applied and results for thresholding with parameter values $l_{cut} = 8, 32, 64$ are visualized in sub-figures (b), (c), and (d).

6 Edgebased-PATHFINDER

In the previous section we utilized the PF's speed to quickly obtain all compressed paths which at least partially intersect the bounding box. Especially for requests where the graph is sparse and matching paths are mainly located within the request box, graph pruning is a powerful tool. For very dense graphs or if a decent amount of paths are only partially within the bounding box, e.g., they intersect at the request's boundaries, many trajectories have to be collected and transmitted compared to the edges which are actually contained uniquely.

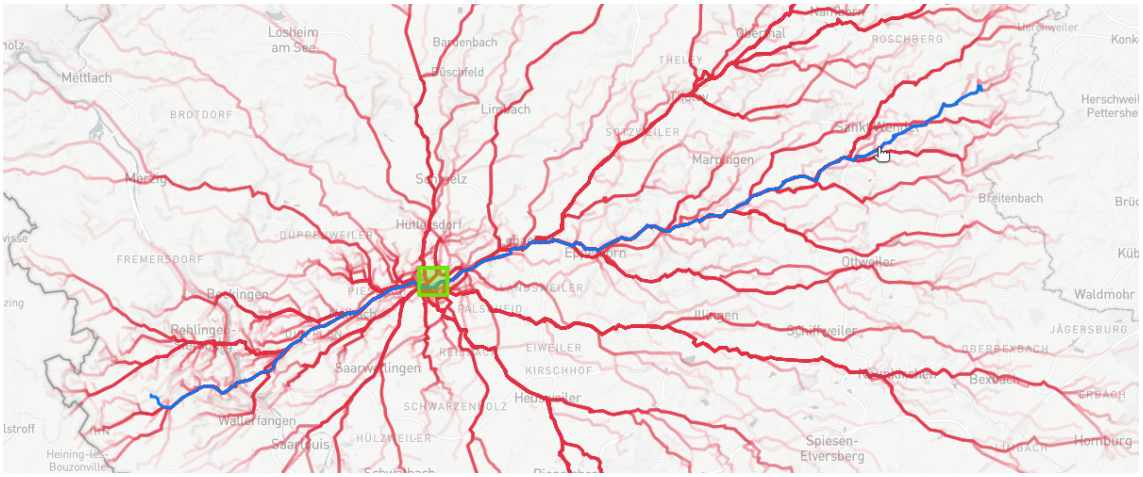
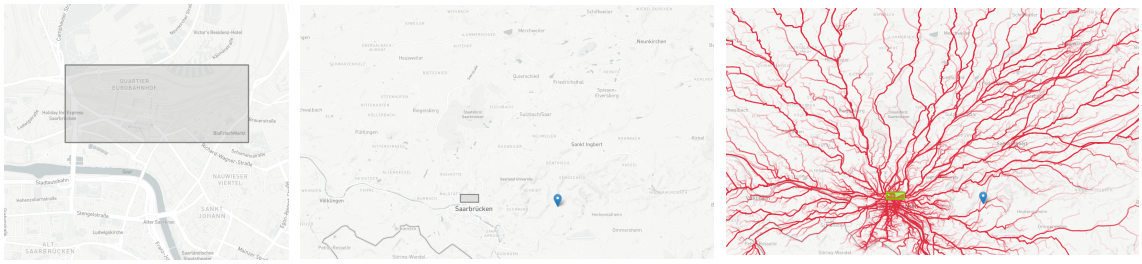


Figure 6.1: A typical result when (at least some) recorded paths are long: Even a small request (green box) produces a considerable amount of output data.

An example is shown in Figure 6.1: A small request box yields many trajectories to be displayed. While it might be very interesting to see, where the trajectories start, end, and which areas are covered more frequently, it is impossible to take a closer look at trajectories within the bounding box at higher unpacking levels or for significantly larger request boxes. To give a more practical example, a realistic use case is shown in Figure 6.2: Assume, one wants to analyze traffic within a city district. By drawing a bounding box covering the city quarter, a busy highway partly intersects the requested area. As a result, a lot of data is loaded outside of the relevant bounding box and the client hardly responses. In this example, the server (here on the Thinkpad, see Section 3.4.2) needed more than 13 seconds to retrieve, export and send all the data. The total transmission size added up to 184 MB. Apparently, there is a need for loading only data within the requested area.



(a) (b) (c)

Figure 6.2: Practical use case of analyzing intra-city traffic. All three images show the same request box. (a) represents the area of interest, e.g., a city center, (b) its zoomed-out version for better comparison to (c) which has the matching response set included. Clearly, most of the transmitted data is not relevant for accomplishing the given use case task. The request box in (c) was colored green to enhance visibility.

A naive solution would reuse the implemented PF code, retrieve all trajectories, unpack and send only those root edges, which intersect the requesting bounding rectangle. For two reasons, this is not efficient: On the one hand, all edges outside the box also have to be checked. A similar strategy was already shown to be problematic in section Section 5.1.1, where root edges are grouped into either the category high- or low-level. On the other hand, if an edge inside the requested area has been used by k trajectories, it is also reported k times. After iterating through all paths, there are many duplicates which have to be removed subsequently.

To solve both issues, the original PF algorithm was modified and an adjusted version, which is called Edge-Based PATHFINDER (EBPF), carefully operates on edges only. EBPF will be explained in detain in the next section.

6.1 Implementation

The initial implementation for PF, as presented in Algorithm 2.3, starts with collecting bounding box intersected edges. In a second step, trajectories for these edges are collected, merged and a duplicate-free list gets returned. Trajectories are linked with edges using an inverted index list. On request time, all trajectories linked with an edge can be obtained by first reading the start and end index for an edge and copying the respective trajectories-entries within that range from the index. By skipping the last steps and refining the postprocessing, only relevant edges within the requesting region are kept. This retrieval method will be called EBPF, with its pseudecode attached:

Algorithm 6.1 Edgebased PATHFINDER Algorithm (high level)

1: **procedure** RUNQUERY(Q)
2: $E_O \leftarrow$ FINDEDGECANDIDATES(Q)
3: $E_r \leftarrow$ REFINEEDGECANDIDATES(Q, E_O)
4: $E \leftarrow$ POSTPROCESSEDGECANDIDATES(E_r)
5: **return** E
6: **end procedure**

Algorithm 6.2 Full unweighted duplicated-free edge exporting, naive approach

```
1: procedure exportFullDuplicateFreeNaive(E)
2:     E' ← []
3:     for e ∈ E do
4:         E' ← E' ∪ unpack(e)
5:     end for
6:     E_ex = removeDuplicates(E')
7:     return exportEdgelistToJSON(E_ex)
8: end procedure
```

Due to the early stopping when performing a PF search, both edges of high and low level are returned. In addition to the fast edge retrieval, counts for each edge can be obtained in constant time. For each edge, the start- and end indices linked to the inverted trajectory index are read. The difference of both values yields the number of trajectories passing the respective edge.

Plotting the result edge set directly does not give satisfactory results, due to edges' different level representations. In the following, exporting techniques are presented to postprocess the initial edge set for better visual appearance.

6.2 Exporting

6.2.1 Input and Output

The input is a length-k list of edge-IDs $E = [e_1, e_2, \ldots, e_k]$ returned from the EBPF edge refining. For the sake of clarity, edge-indices (IDs) are used interchangeably with their respective edge objects. Therefore, edge attributes and methods can be used. Clearly, this is not a real restriction, since objects can be retrieved from IDs in constant time and vice-versa. In the case of weighted edges, an additional weighting function w is available. In practice, this can be realized using a hash list or a supplementary edge attribute. Initially, EBPF returns the edge-weights as vector $C = [c_{e_1}, \ldots, c_{e_k}]$, so $w(e) = c_e \forall e \in E$ and 0 otherwise.

The intermediate output is denoted as E_{ex}, representing a list of edges (or theirs IDs, depending on the context). The exporting size is denoted as o_{ex}. Conclusively, this result list gets converted into an appropriate string representation to allow for web transmissions.

6.2.2 Exporting Basics

The naive way to export edges is to unpack all edges and concatenate the unpacked results (see Algorithm 6.2). To avoid plotting edges twice, duplicates are removed before exporting. The asymptotic running time is dominated by the unpacking and duplicate removal, which both need linear time in their input. To find an upper bound, it suffices to upperbound the size of E'. Trivially, the more edges are shared among the unpacked $e \in E$, the more inefficient the algorithm becomes. In terms of output sensitivity with respect to $|E_{ex}|$, $O(k \cdot o_{ex})$ steps are used asymptotically in the worst case.

Algorithm 6.3 Full unweighted duplicated-free edge exporting

```
1:  procedure exportFullDuplicateFree(E)
2:      E_ex ← []
3:      closed ← Set()
4:      while |E| > 0 do
5:          e ← E.pop()
6:          if e ∈ closed then
7:              continue
8:          end if
9:          closed ← closed ∪ {e}
10:         if e.is_shortcut then
11:             e_1, e_2 ← e.children
12:             E ← E ∪ [e_2, e_1]
13:         else
14:             E_ex ← E_ex ∪ [e]
15:         end if
16:     end while
17:     return exportEdgelistToJSON(E_ex)
18: end procedure
```

The input set E may contain edges e and e' where latter is part of e. To avoid unpacking of identical parts, Algorithm 6.3 keeps track of the unpacking history using a HashSet *closed*. Once an edge is viewed a second time, unpacking is skipped, since it has been added already.

Lemma 2 *The search tree emerged from unpacking a cached edgeset-unpacking-procedure resulting in n non-shortcut edges E_{ex} has size $\Theta(n)$.*

PROOF Showing the lower bound $\Omega(n)$ is trivial and will not be discussed further. W.l.o.g. one can assume that there is only one input edge creating the whole search tree. This is not a restriction, since we could build up a single search tree from multiple trees by iteratively introducing new parent edges merging trees, similar to [29], ignoring the probabilities. If we can show that this tree cannot have more than $O(n)$ edges, the original tree's sizes summed up must have been even smaller.

An upper bound can be obtained by a bottom-up analysis. When the algorithm has been terminated, n plain edges were found. Each edge either entered the result set by an unpacking process from a parent edge or because it was already in the input set. In the former case, a constant amount of time was spent and no other edge has contributed to the search tree. For the latter case, exactly one other edge on the same level was involved in the unpacking. Each shortcut edge created two new edges, therefore no more than n inner shortcuts exist. In total, $O(n)$ edges were involved. This yields the claim. ∎

Theorem 3 *Algorithm 6.3's running time is optimal.*

PROOF The output has size o_{ex}, therefore it remains to proof, that the algorithms runs in $O(o_{ex})$. The loop-body's statements run all in amortized $\Theta(1)$ time. In each iteration, either $|E|$ decrements or increments by one. The latter case happens iff. a shortcut edge is visited for the first time. Since there are o_{ex} edges in the output, according to Lemma 2 not more than o_{ex} shortcut edges can

Algorithm 6.4 Full weighted duplicated-free edge exporting

```
1:  procedure exportFullWeightedDuplicateFree(E)
2:      prioQ ← PriorityQueue(E)          // Build Max-Heap-based Priority Queue w.r.t. w(e)
3:      active ← Set(E)
4:      while |prioQ| > 0 do
5:          e ← prioQ.extractMax()
6:          if e.is_shortcut then
7:              active.remove(e)
8:              for e_i ∈ e.children do
9:                  prioQ.enqueue(e_i)
10:                 w(e_i) := w(e_i) + w(e)
11:                 active ← active ∪ {e_i}
12:             end for
13:         end if
14:     end while
15:     E_ex ← [e : e ∈ active]
16:     return exportWeightedEdgelistToJSON(E_ex, w)
17: end procedure
```

be involved, which in turn bounds the number of total iterations. Hence, the running time of the algorithm is in $\Theta(o_{ex})$, namely linear in the expected output. There's no algorithm writing o_{ex} edges in $\omega(o_{ex})$ time, which concludes the proof. ∎

6.2.3 Weighted Edges

Besides the use case of plotting all unique edges intersected by the requesting bounding rectangle, the user might be interested in usage information, e.g., which edge is used by how many paths. For encoding, a weight function $w : E \to \mathbb{R}^+$ is introduced, storing a count value for each retrieved (shortcut-)edge.

Algorithm 6.4 implements an appropriate top-down approach for extracting all child edges with its correct counts. The pseudo code assumes *prioQ* to not insert elements twice to avoid double-counting. If this is not the case in a practical implementation, it can be circumvented by wrapping the queue with the following logic: Create a (hash)set allowing to check for existence before inserting and remove elements on a pop operation, both constant time. The top-down traversal also makes sure to increment counters of children correctly, because counts of (grand-)parents are aggregated and propagated downwards over time.

Theorem 4 *Algorithm 6.4's running time is in* $O(o_{ex} \log(o_{ex}))$.

Proof Setting up the priority queue and creating *active* takes linear time. Using a Fibonacci heap, inserting elements takes amortized constant time [23]. Hence, each inner loop's command takes constant time. Therefore, the iterations are dominated by the $O(log(|prioQ|))$ term for extracting the minimum [23]. $|prioQ|$ cannot exceed the size of the search tree(s), each bounded by $O(o_{ex})$, according to Lemma 2. Since each element is inserted at most once, no more than $O(o_{ex})$ elements can be added to the queue, for the same reason. This results in the claimed running time. ∎

6.3 Binning

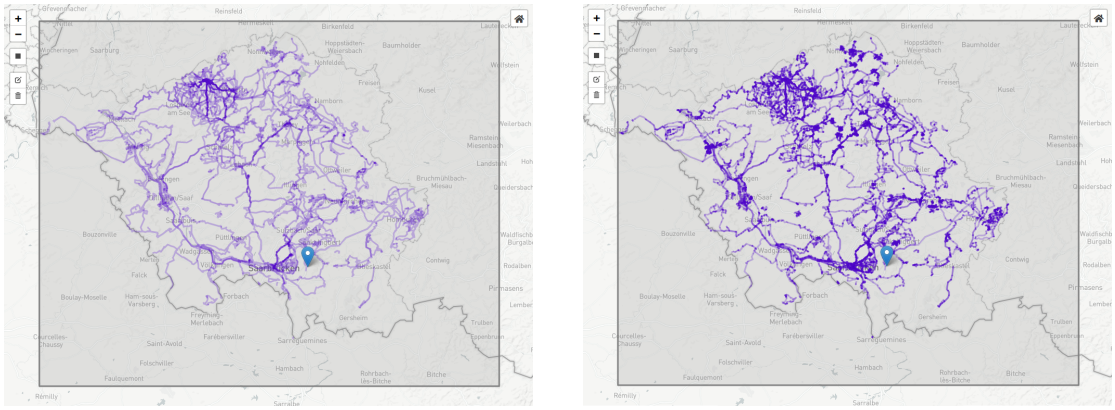
Different from the plotting logic for edge strokes and fully unpacked trajectory lists, EBPF returns a long list of edges represented by two points each. In case of weighting enabled, each edge comes with an additional weight. As a consequence, the frontend plotting library creates a single object for each edge and applies its color according to the weight. It turned out that the plotting time for the weighted case is significantly larger compared to the case where all edges are combined as one object and a color is only set once.

To address this issue, the number of different colors is restricted to b values, which are chosen dynamically in the total color range: For values between the lowest (typically weight 1) and the maximum usage, $b - 2$ gradations in between with equal distances are defined. The server groups the edges together in b edge lists with b respective color values. On plotting time, the client creates one objects for each group, assigns the edges, and sets the colors. This speeds up the frontend calculation to the disadvantage of the server's longer preprocessing phase. To verify that the concept is favorable, the implementation is tested on the Saarland Dataset for five requests of increasing size. A description of the datasets is listed in Table 6.1.

dataset ID	minimum coordinate		maximum coordinate		Zoom	# edges
	lat	lon	lat	lon		
1	49.2993822	7.0357132	49.2438279	7.1201706	13	1001
2	49.2536889	6.9182968	49.2124390	7.0703888	13	1965
3	49.4692396	6.7387390	49.6000304	7.0188904	13	5081
4	49.2310505	6.7497253	49.5510527	7.0868683	11	10067
5	49.6578498	6.3761902	49.0306652	7.5915527	10	199956

Table 6.1: Datasets used for evaluation. All request types were weighted and plotted in transparency mode.

In Figure 6.3, the simplification of Dataset 5 using 25 color bins only is contrasted with its original representation and similar comparisons for the other datasets are appended (Figures C.3 and C.4). In the original Dataset 5 response, there are 22 784 edge strokes objects to be colored, but only 30 unique colors in total, ranging between 1 and 41. Even though a difference in contrast is visible, the key characteristics such as very frequently and rather rarely used edge strokes are still visible.



(a) 25 bins **(b)** no binning

Figure 6.3: Comparison of the original plot of Dataset 5 and its equivalent representation which is limited to 25 color scales.

Five different datasets are chosen to estimate the speedup on various request sizes. For each of the five different request sets, the times for postprocessing on the back- and plotting on frontend side have been recorded. The results are presented in Figure 6.4: Firstly, average calculation times increase as the request size grow, because more edges have to be processed. Moreover, the processing time for exporting only slightly increases compared to the original non-binning version. Lastly and most significantly the frontend times drop drastically: For the largest dataset, for example, the plotting times approximately drop by a factor of six.

To compare the transmission sizes, the original response size is compared to different binned instances. The sizes are outlined in Table 6.2, but there is no significant reduction compared to transmitting a single counter for each edge.

To sum up, the example illustrates the potential for plotting lists of edge lists at once over plotting each edge stoke on its own. The binned representations show the most essential usage characteristics, but better binning strategies would be needed to fully exploit its potential. Small binning sizes show a good separation between frequently and rarely used strokes. While the transmission sizes and backend calculation times do not change noticeably, the plotting times on frontend side are reduced significantly.

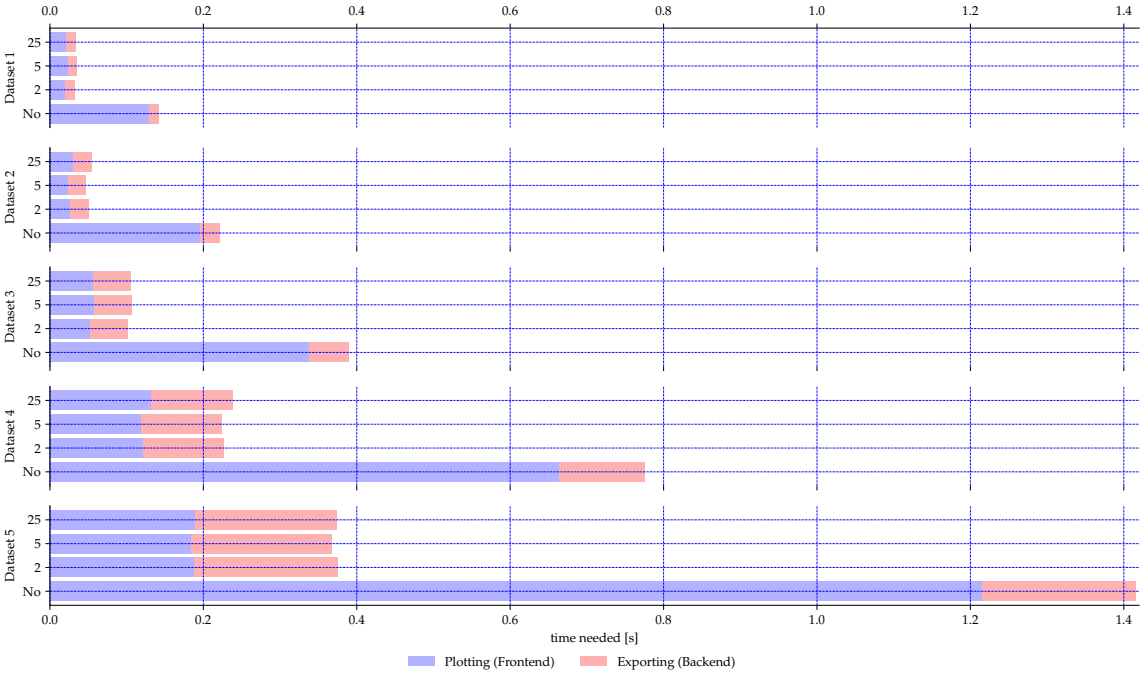


Figure 6.4: Comparison of processing times for different binning types on all five datasets. *No* indicates the case where no binning was used.

Bins	Dataset 1	Dataset 2	Dataset 3	Dataset 4	Dataset 5
-	0.14	0.26	0.58	1.19	2.09
1	0.13	0.26	0.57	1.17	2.05
2	0.13	0.26	0.57	1.17	2.05
3	0.13	0.26	0.57	1.17	2.05
4	0.13	0.26	0.57	1.17	2.05
5	0.13	0.26	0.57	1.17	2.95
10	0.13	0.26	0.57	1.17	2.05
25	0.14	0.26	0.57	1.17	2.05
100	0.14	0.26	0.57	1.17	2.05
200	0.14	0.26	0.57	1.17	2.05

Table 6.2: Transmission sizes (in MB) for all five test datasets under variation of the bin size. The first row refers to the plot without binning.

6.4 Sensitive CH-Unpacking

In the first part of this chapter, EBPF persuasively showed its strength due to its limited catchment area and lightweight implementation by returning matching edges directly. In the weighted case, counts for each retrieved edge are added in constant time each. A central issue is, however, that this

edge set can not be used for plotting directly, as implied by Figure 6.5, because the result entries returned by EBPF are not disjoint in the sense that the pairwise intersections of all unpacked edges will not necessarily be empty.

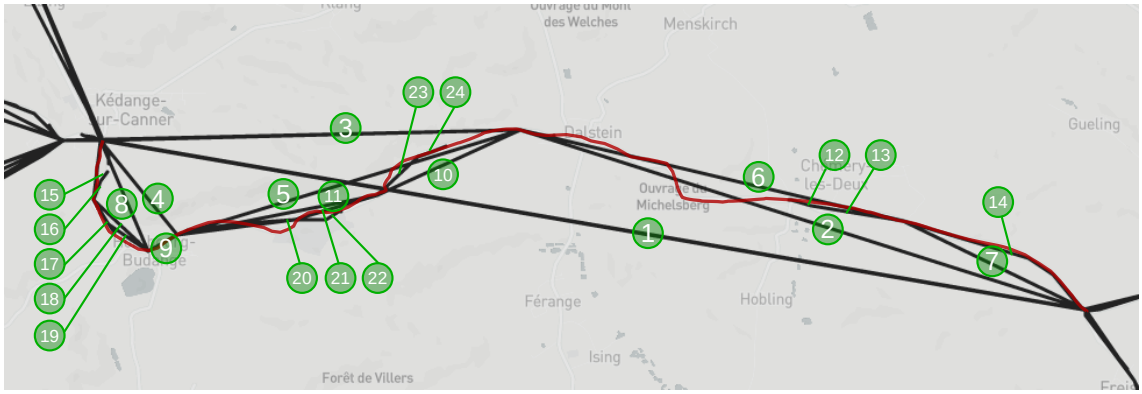


Figure 6.5: Example data set on the real Europe graph showing unpleasant results: Even though there is only one street (red), EBPF returns multiple edges on various levels colored in dark gray. Note that fully unpacking edge with label 1 would result in the original street colored red. Unpacking all other streets labeled 2 – 24 will give partial street segments.

A first approach which unpacks all edges fully and suppresses duplicates was discussed in Section 6.2, but returning level-0 edges is not always desired (see introduction of Chapter 5 and the example in Chapter 1).

6.4.1 Unweighted Case

In the following, an edge-resolution algorithm called Sensitive CH-Unpacking Algorithm (SUA) is presented, which only unpacks an edge if its unpacked path partially intersects an unpacked path of another edge. Child edges are treated similarly in a recursive manner. Before the algorithm is explained in detail, edge intersection definitions are introduced:

Definition 7 (Edge Intersection) Let e_1 and e_2 be (shortcut) edges and π_1 and π_2 their corresponding paths which emerge by unpacking the edges fully. e_1 has non-empty intersection with e_2, i.e., e_1 *intersects* e_2 iff. $\{e|e \in \pi_1\} \cap \{e|e \in \pi_2\} \neq \emptyset$. In short, $e_1 \cap e_2 \neq \emptyset$ is equivalent. □

Definition 8 (Subedge) Let e_1 and e_2 be (shortcut) edges and π_1 and π_2 their corresponding paths which emerge by unpacking the edges fully. Then, e_1 is a *subedge* of e_2, if π_2 can be decomposed into $\pi_2 = \pi_p \pi_1 \pi_q$, where π_p and π_q are (potentially empty) paths. In short, $e_1 \in e_2$ holds true. □

Definition 9 (Cover) Let E be a set of (shortcut) edges. The *cover* $C(E)$ of E is the set of all plain subedges of E: $C(E) = \bigcup_{e \in E} unpack(e)$. □

Definition 10 (Equivalent Edge Sets) Two edge sets E_1 and E_2 are called *equivalent* iff. $C(E_1) = C(E_2)$. It can be abbreviated as $E_1 \equiv E_2$. □

Trivially, $e_1 \in e_2$ implies $e_1 \cap e_2 \neq \emptyset$ and $E_p = C(E_p)$ if E_p already consists of plain edges only.

Given a set of edges E, the goal of the SUA is to return a pairwise intersection free equivalent edge set. A simple solution would be to calculate and return the cover of E. In this implementation, however, edges of high levels are returned where possible. Three full edge loops are included in the calculation which is listed in Algorithm 6.5.

Algorithm 6.5 Sensitive Unpacking Algorithm (High Level)

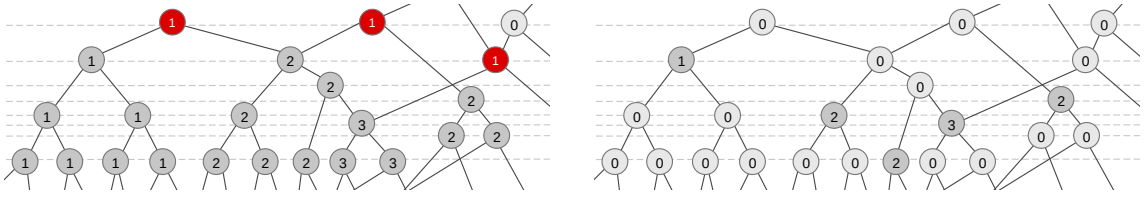
```
1: procedure EXPORTEDGESETSENSITIVELY(E, l_⊓)
2:     count ← [0 | e ∈ E_GCH]
3:     MARKINITIALLY(count, E)
4:     FLOODMARKINGS(count)
5:     PROPAGATEUPWARDS(count)
6:     E_ex ← COLLECTMATCHES(count)
7:     return UNPACK(E_ex, l_⊓)
8: end procedure
```

To propagate information level up or down, the edges are ordered according to their edge level at preprocessing time. In the following, $E^{\geq}_{G_{CH}}$ denotes the total CH edge set in descending order according to the edges' levels, while $E^{\leq}_{G_{CH}}$ denotes its reversed equivalent. Note that for implementing, only one array needs to be stored since the other is obtained by looping backwards.

SUA keeps track of edges being in use by storing counters using the array *count*. Initially, all counts are 0, but the third line increments all input edges counters to one. In the first full loop, the contraction hierarchy is traversed level-by-level in a top-down fashion and parents' counter values are propagated downwards to the child edges. Overlapping subtrees add up their counter values, as indicated in Figure 6.6a, followed by a bottom-up level-traversal where children are removed if the parent edge is kept, in case of inconsistent values, splitting information is propagated upwards by setting the parent to 0. The result is shown in Figure 6.6b. Finally, all non-empty count edges are kept as a final result. Although the algorithm already performs partially unpacking of an edge if a subedge is also in use, on large zoom scales, returning potentially large shortcut edges can give unsatisfactory results. To solve this issue, the resulting edge set can be unpacked further before it gets returned. As a default settling, however, $l_{\sqcap}$ is set to ∞, i.e., no further unpacking is performed.

Algorithm 6.6 Sensitive Unpacking Algorithm (Low Level)

```
1: procedure MARKINITIALLY(count, E)
2:     for all e ∈ E do
3:         count[e] ← 1
4:     end for
5: end procedure
1: procedure FLOODMARKINGS(count)
2:     for all e ∈ E^≥_{G_CH} do
3:         if e.is_shortcut then
4:             e_1, e_2 ← e.children
5:             count[e_1] ← count[e_1] + count[e]
6:             count[e_2] ← count[e_2] + count[e]
7:         end if
8:     end for
9: end procedure
1: procedure PROPAGATEUPWARDS(count)
2:     for all e ∈ E^≤_{G_CH} do
3:         if count[e] = 0 then
4:             continue
5:         end if
6:         if e.is_shortcut then
7:             e_1, e_2 ← e.children
8:             if count[e_1] = count[e_2] = count[e] then
9:                 count[e_1] ← 0                    // Replace children edges by parent edge
10:                count[e_2] ← 0
11:            else
12:                count[e] ← 0                      // Make sure to unpack edges on the up-path
13:            end if
14:        end if
15:    end for
16: end procedure
1: procedure COLLECTMATCHES(count)
2:     return [e ∈ E_{G_CH} | count[e] ≥ 1]
3: end procedure
```



(a) **(b)**

Figure 6.6: Example showing the main two algorithm loops (lines four and five): Initially, only the red input edges are marked. Line four pushes count values downwards, resulting in subfigure (a). Subfigure (b) shows the counters after line five, same-countered subtrees have merged again, inconsistent roots have been kept and merge information have been propagated upwards. Finally, all edges with corresponding non-zero values are returned in line six.

To illustrate a more concrete scenario, an intersection of three streets is given in the graph, where each of the three outgoing paths is complex, i.e., many plain edges are needed for representing the strokes. Shortcut edges have been added during CH-contraction. Further is assumed that two trajectories use the same street initially, but other output paths when leaving the intersection. When the intersection vertex is removed and shortest paths are kept, up to three new shortcut edges are added connecting the three street's endpoints. On request time, EBPF returns this two shortcut edges, as outlined in Figure 6.7a. Naively, when the edges are fully unpacked, all plain edges are returned (Figure 6.7b). Instead, a more abstract representation can be obtained by using shortcut edges instead of expensive plain edge representations. The former is returned by SUA.

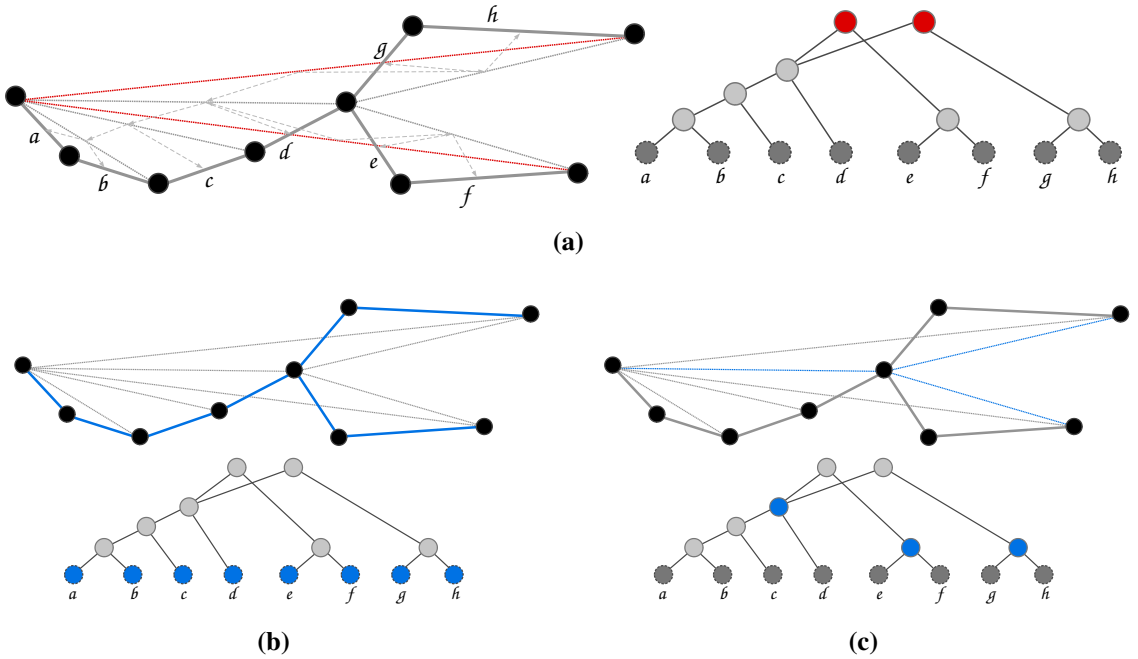


Figure 6.7: Example showing SUA's potential: In the left half of (a), the input scene is given, where three edge strokes merge. Shortcut edges are drawn dotted. Red edges indicate EBPF returned input edges, labeled thicker edges represent plain edges. On the right side of (a), the edge hierarchy is visualized. All blue edges are part the result sets chosen by the different methods: While the naive full unpacking procedure in (b) returns all plain edges as the leaves of the CH, SUA keeps the highest edges only. This is nicely outlined in the edge hierarchy of (c).

Theorem 5 *Sensitive unpacking takes* $O(|E_{G_{CH}}|)$ *time.*

PROOF Marking and returning of edges is based on a $|E_{G_{CH}}|$-sized counting array. Since each step loops through the list at most once with constant time each iteration, the worst case running time asymptotically differs by a constant factor only. For the final (optional) unpacking procedure, no edge is visited and exported twice due to the fact that by construction the set E_{ex} is intersection-free, which proves the claim. ■

The concept of propagating counts to lower children can be generalized for weighted inputs, which will be presented in the following section.

Algorithm 6.8 Weighted Sensitive Unpacking Algorithm (Low Level)

```
1: procedure markInitially(count, E, w)
2:     for all e ∈ E do
3:         count[e] ← w[e]
4:     end for
5: end procedure
1: procedure collectMatches(count, ζ)
2:     E_ex = [e ∈ E_{G_CH} | count[e] ≥ ζ]
3:     w_ex = [count[e] | e ∈ E_ex]
4:     return E_ex, w_ex
5: end procedure
```

6.4.2 Weighted Case

For the weighted version of SUA, namely Weighted Sensitive Unpacking Algorithm (WSUA), counters are initialized with the collected weights instead of static *one* entries. In addition, a minimum intersection count ζ is introduced to only return edges which satisfy a certain minimum usage. Aggregated weights are returned too. The changed code blocks are listed in Algorithm 6.7 and Algorithm 6.8.

Algorithm 6.7 Weighted Sensitive Unpacking Algorithm (High Level)

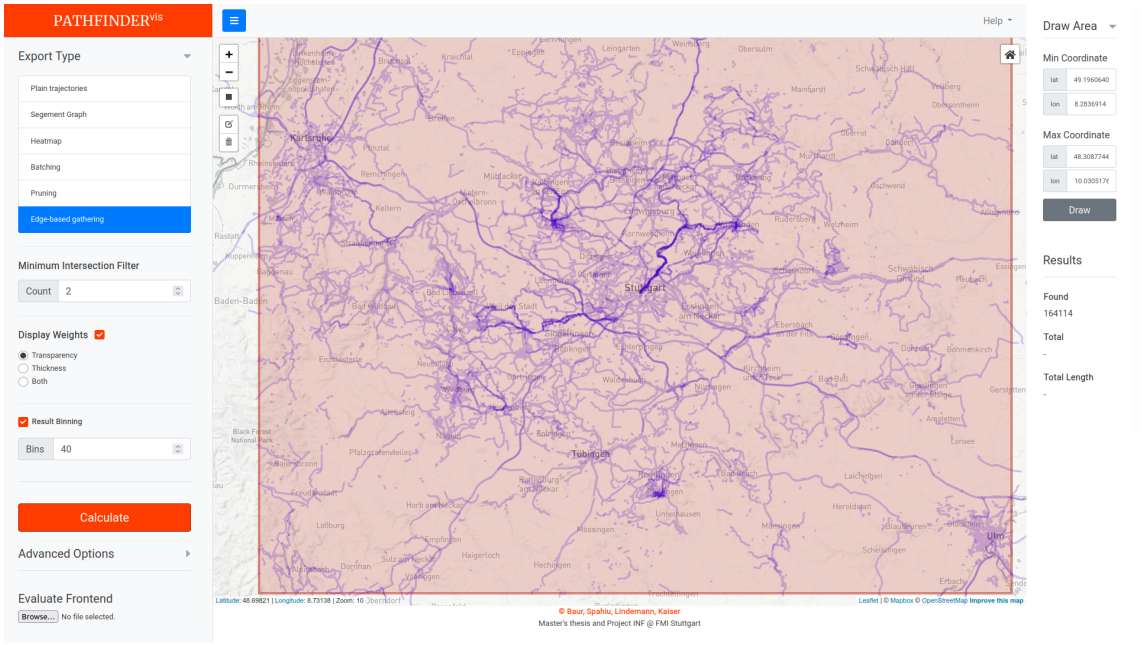
```
1: procedure exportWeightedEdgesetSensitively(E, w, ζ, l_⊓)
2:     count ← [0 | e ∈ E_{G_CH}]
3:     markInitially(count, E, w)
4:     floodMarkings(count)
5:     propagateUpwards(count)
6:     E_ex, w_ex ← collectMatches(count, ζ)
7:     return unpack(E_ex, l_⊓), w_ex
8: end procedure
```

Due to the very same overall merging scheme as SUA, the same running time argumentation holds true for WSUA.

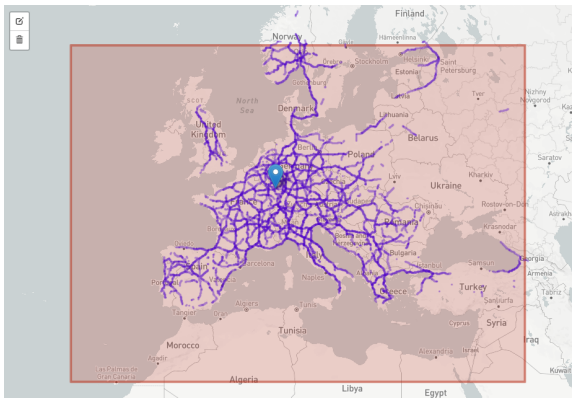
6.5 Discussion

6.5.1 Implementation

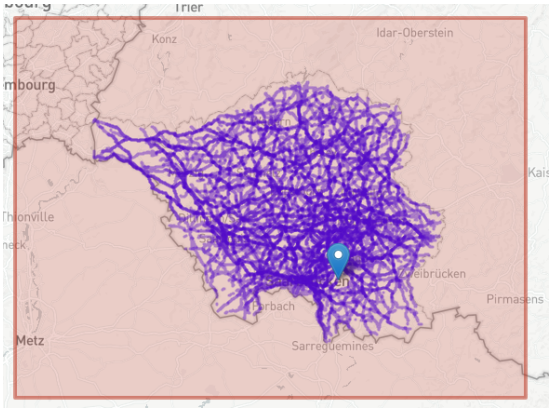
To gain a better understanding for further discussions, some typical outputs are presented first. Figure 6.8a shows the greater surrounding of a German city in high detail, since only edges have been pruned which are used less than two times. A requests of larger scale is shown in Figure 6.8b: The artificial Europe500 path sets was queried and only edge strokes used 500 times or more were kept. In the last setting, the Saarland1M Dataset was requested filtering with ζ being 512.



(a)



(b)



(c)

Figure 6.8: Typical EBPF outputs on different graph and path data: (a) depicts a response on the Germany Dataset, (b) a high level request on the Europe500K Dataset, and (c) the full Saarland1M Dataset filtered for most used edges.

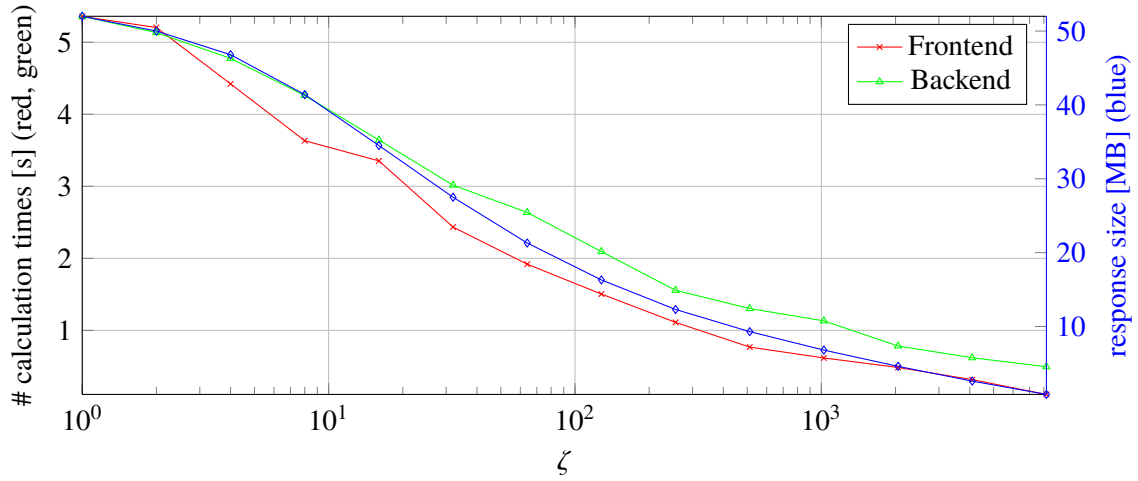


Figure 6.9: Transmission size and calculation time changes using the Saarland1M Dataset under variation of ζ. Note, that the abscissa is scaled logarithmically.

6.5.2 Measurements

For evaluating the edge based PF variant, the following setup was used: Frontend timings were measured on the ThinkPad, which also hosted the backend for the Saarland Dataset retrievals. Germany and Europe graphs ran on the more powerful Threadripper.

The Tables 6.3, 6.4 and 6.5 list the measurements taken from Saarland Datasets, namely the filtered original Dataset and the artificially generated set containing 100 000 and 1 000 000 paths respectively. Comparing the Saarland100K Dataset and its ten times larger variant, based on the dominating edge sweeping part, times for small request boxes do not differ to much. Measurements were also taken for the Germany and Europe500k dataset, but sweeping through all edges exceeds the time limit of two seconds by far. For the Germany Dataset, for example, no response has been answered in less than 13.9 seconds (see Table C.6 for more measurements). Hence, sensitive unpacking can not be applied to graphs of that size.

Finally, measurements were taken on the Saarland10M Dataset containing ten Million trajectories. The results are appended in C.7. Calculation took place on the Threadripper. Interestingly, the times up to a request factor of $1/4$ are sill acceptable. This is because the algorithm runs into saturation, because at some point all relevant edges have been used once and reusing does not increase the response's size, but only the count.

To further investigate the influence of ζ, Figure 6.9 shows the results for the Saarland1M Dataset under variation of the minimum intersection threshold. The graph bounding box served as a static request box and 30 bins were used. Each point of the green and red curve resulted from averaging ten time measurements. There are a few conclusions which can be drawn from the graphic. First, all three curves behave similarly, meaning that backend times correlate with transmission sizes and frontend work, because varying ζ only influences the edges collected in the last sweep. The larger this threshold is chosen, the less edges are collected, and the fewer have to be unpacked. Less unpacked edges result in smaller response packages and less frontend work. Secondly, the overall times needed intuitively reduce the larger the threshold is chosen. For this special graph and request

	Zoom								
	2	4	6	8	10	12	14	16	18
Factor 1/32									
Frontend [ms]	1	2	1	2	2	1	1	2	2
Backend [ms]	125	134	119	119	133	91	120	140	134
Size [KB]	5	5	5	7	6	5	5	13	13
Factor 1/16									
Frontend [ms]	1	3	2	2	4	3	3	2	4
Backend [ms]	130	139	142	133	136	134	126	112	143
Size [KB]	20	14	11	11	31	24	26	19	33
Factor 1/8									
Frontend [ms]	2	6	4	8	6	9	6	14	9
Backend [ms]	154	143	154	152	155	164	145	164	166
Size [KB]	33	44	46	77	61	89	79	105	118
Factor 1/4									
Frontend [ms]	7	16	16	15	19	15	20	19	29
Backend [ms]	160	167	177	169	186	177	190	191	196
Size [KB]	127	169	166	171	229	204	243	304	357
Factor 1/2									
Frontend [ms]	20	46	45	50	56	53	61	62	91
Backend [ms]	203	219	222	230	231	235	255	254	275
Size [KB]	408	469	527	600	694	687	878	834	1153

Table 6.3: Transmission sizes and calculation times using EBPF with binning activated ($b = 25$) on the Saarland Dataset.

size choice, the frontend typically needs less time to process the response than the backend needs to generate but the other graphs show a very similar behavior. This can be explained by the fact that the naive sweeps iterating through all edges are expensive operations, but at most all plain edges are retrieved. Most importantly, the curve drops quickly for small values of ζ: Both the times for the back- and frontend, as well as the size reduced by more than 20 percent for changing ζ to being ten. At around $\zeta = 80$, times and size even halved. Note, that for the last value $\zeta = 8\,192$, still 19 712 edges are retrieved.

On the one hand, unpacking edges sensitively results in a simplification of edge strokes. Profound unpacking at intersections or where multiple trajectories start or end to overlap is needed on the other hand. This helpful behavior is stressed out using Figure 6.10, which compares the fully unpacked EBPF output with its SUA counterpart: Especially dead ends and streets having only very few junctions are replaced by a few straight lines only. While the overall topology still applies, most of the path-specific details were removed. Transmitting the simplified scene consumes 20.8 KB which is, compared to its original unpacked version using 56.4 KB, a reduction by over 63 %.

	Zoom								
	2	4	6	8	10	12	14	16	18
Factor 1/32									
Frontend [ms]	9	8	10	6	9	9	9	20	10
Backend [ms]	160	163	172	168	181	180	175	182	178
Size [MB]	0.10	0.12	0.14	0.13	0.19	0.17	0.20	0.25	0.26
Factor 1/16									
Frontend [ms]	42	25	30	20	28	20	18	17	18
Backend [ms]	195	186	194	186	207	236	208	198	216
Size [MB]	0.36	0.29	0.31	0.28	0.43	0.49	0.46	0.38	0.51
Factor 1/8									
Frontend [ms]	73	66	69	52	49	74	68	76	77
Backend [ms]	245	250	264	288	266	315	300	315	309
Size [MB]	0.79	0.74	0.90	1.15	1.01	1.45	1.25	1.53	1.48
Factor 1/4									
Frontend [ms]	222	257	242	142	183	163	176	167	237
Backend [ms]	409	474	454	443	522	491	529	517	663
Size [MB]	2.52	3.04	2.88	2.86	3.61	3.38	3.78	3.58	5.08
Factor 1/2									
Frontend [ms]	764	783	750	528	446	466	535	564	536
Backend [ms]	985	1043	1051	1096	1065	1118	1228	1285	1291
Size [MB]	8.30	8.99	8.88	9.52	9.28	9.72	10.98	11.51	11.68

Table 6.4: Transmission sizes and calculation times using EBPF with binning activated ($b = 25$) on the Saarland100K Dataset.

(a) **(b)**

Figure 6.10: Showing the same EBPF response (a) fully unpacked and (b) postprocessed with SUA on a large zoom scale.

	Zoom								
	2	4	6	8	10	12	14	16	18
Factor 1/32									
Frontend [ms]	9	22	10	12	11	9	11	13	13
Backend [ms]	188	199	184	187	210	186	187	201	184
Size [MB]	0.22	0.24	0.23	0.22	0.30	0.25	0.30	0.38	0.37
Factor 1/16									
Frontend [ms]	35	21	21	19	44	45	47	41	48
Backend [ms]	255	232	228	222	261	255	240	235	225
Size [MB]	0.80	0.57	0.58	0.51	0.73	0.83	0.75	0.61	0.77
Factor 1/8									
Frontend [ms]	94	74	106	114	81	113	92	122	139
Backend [ms]	384	357	351	412	350	424	379	421	429
Size [MB]	1.80	1.58	1.86	2.23	1.84	2.61	2.16	2.60	2.41
Factor 1/4									
Frontend [ms]	315	349	331	293	328	317	347	360	540
Backend [ms]	793	898	763	757	821	768	798	758	1066
Size [MB]	5.97	6.87	6.21	5.82	7.05	6.41	6.79	6.40	8.50
Factor 1/2									
Frontend [ms]	1257	1199	1212	1120	921	1000	1177	1642	1655
Backend [ms]	2216	2309	2200	2175	1898	1929	2040	2194	2146
Size [MB]	19.93	20.97	19.78	20.25	18.70	19.04	20.32	21.32	19.97

Table 6.5: Transmission sizes and calculation times using EBPF with binning activated ($b = 25$) on the Saarland1M Dataset.

While SUA yields good results for compressing edge strokes, it is a limitation that EBPF's advantage of returning only local edges will get lost due to the required full edge sweeps of SUA. This is a drawback, especially for graphs with many edges, where sweeps are expensive in relation to the edgeset being returned. Further research is required to develop a sweep-less sensitive unpacking version to maintain locality. An initial approach is presented in the Future Work chapter (Section 8.4). For now, it might be worth to utilize duplicate free unpacking for very small edge sets instead.

Another limitation which needs to be fixed in future implementations is the double-linking of edges. For two adjacent nodes u and v, there is one edge connecting u with v and another edge in opposite direction. This was implemented to easily encode trajectories by edges, since directions are implicitly stored. Figure 6.11 explains the difference between trajectories linked to bidirectional and unidirectional edges in the context of sensitive unpacking: In the figures 6.11a to 6.11d, a typical input scene on bidirectional edges is shown. Initially, two fully unpacked trajectories colored in blue and orange are drawn with their respective plain edges. In the scene a path split is given consisting of three plain edges a, b, and c, and two shortcut edges d and e connecting $a - b$ and

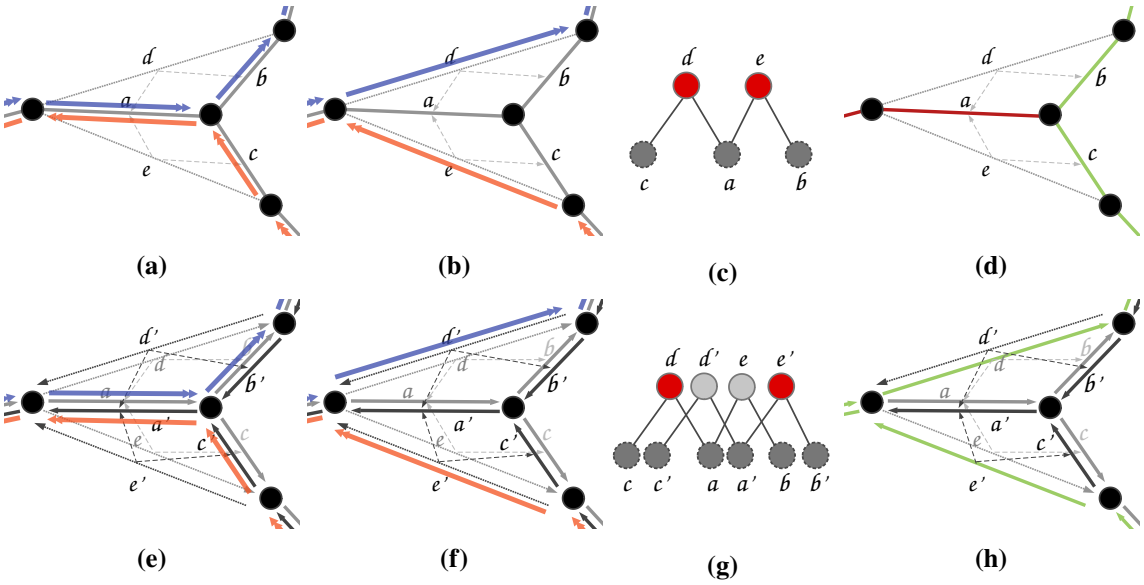


Figure 6.11: Comparison between unidirectional and bidirectional edge orientation.

$a - c$ respectively. The compressed version, as they would be stored in the PF data structure, are depicted in Figure 6.11b. On requesting EBPF, it will return the two edges d and e, which will then be spitted into a, b, and c by SUA, since they both have a in common. The unpacking resulted from the common parent in the CH-hierarchy, as sketched in Figure 6.11c. Figure 6.11d depicts the final result, where green indicates weight 1 and red weight 2, since it was used by two trajectories. In the current implementation, however, the following inconsistency occurs if edges are traversed in different directions: Figure 6.11e shows the same scene, but using unidirectional edges. Each edge $u - v$ also has a back edge $v - u$ and depending on which direction the trajectory is traversed through, the one or the other edges is used. In this case, there are also back edges for the shortcuts. If now the same trajectories are in use, both back and front edges will be picked, as visualized in Figure 6.11f, but there will be no shared edges any longer (Figure 6.11g). The final result is summarized in Figure 6.11h: Since there is no intersection, no unpacking will take place. The latter case is not an intended situation and should be fixed in the further work. An initial sketch solving this issue is described in Section 8.4.

As a recommendation regarding possible use cases, EBPF does not support microscopic analysis tasks, since single trajectories have been aggregated on the server side. For small graphs, the method works fine for both inter-city and and macroscopic analysis, since the results highlight paths of high usage without losing to much route details.

6.5.3 Summary

The modified PF version EBPF retrieves request-box related edges only, allowing for high resolution outputs, e.g., by unpacking the matches edges. SUA, an alternative exporting mechanism, calculates overlapping segments and unpacks edges having different usage counts while edge strokes of same usage are simplified. The parameter ζ steers the edge return selection process by filtering edges

out which were used less that the given threshold. For larger graphs having multiple millions of edges, however, the method is not applicable in combination with SUA, hence fully unpacking is suggested. Closing this gap will be a central issue for future research.

7 Tiling

All previously presented methods have in common that the requested trajectories are encoded by lists of coordinates. The server first collects all matching edges, converts them into a list of latitude-longitude pairs and transmits the data string. The client interprets the lists as vector lines defined on a 2D surface.

During the evaluation of the implemented methods, a clear bottleneck on client's side has been detected: While a server just needs longer to process a larger request, the client's response time to the user increases significantly. For larger inputs, it even stops working or the browser tab crashes. A reasonable explanation for this behavior is based on the fact that each tap only uses a single CPU core for all underlying processing tasks [37]. If a request is received, the json-string has to be decoded, parsed into point lists and plotted. Additional callback functions like scrolling or rolling the map require further plotting updates.

This section tackles bottleneck problems on client's side by shifting most of the plotting to server's side. Instead of sending a vectorized description of the plot to the client, only a bitmap is transferred to keep the client as lightweight as possible. This approach also allows for parallel processing on multiple cores, since plotting is performed on a powerful multicore architecture. In our concrete example, the server has 16 times as many cores available as the single client thread has.

7.1 Basics

To reduce the client's processing load, retrieval and plotting will be performed on the server. The final result will be embedded in a *tile layer*. This small section briefly introduces the concept of tiles in the context of map rendering.

From an abstract perspective, an interactive map (like leaflet in this work's case) works as follows: The whole map is rendered in advance on the server side, discretized on a grid of squared images. In the context of tiling, these images are called *tiles* [42]. For a given view on a zoom level l, tiles are loaded such that the whole requested viewport is covered. The tile sizes are usually powers of two, in the case of OSM it is 256×256 pixles [40, 42]. When performing a map-pan operation, meaning the map slips around while dragging the mouse, some tiles disappear from the viewport and new tiles have to be requested [40].

Most maps already support own tiling embeddings [2, 35, 36, 39]. A common naming style follows the pattern `http://<host>/<domain>/z/x/y.png` for requesting a specific tile index by zoom level z and position (x,y). The mapping between a location on the map and the respective indices is given by the following equations, taken from [40]:

$$x = \left\lfloor \frac{lon + 180}{360} \cdot 2^z \right\rfloor, \quad y = \left\lfloor \left(1 - \frac{\ln\left(\tan(lat \cdot \frac{\pi}{180}) + \frac{1}{\cos(lat \cdot \frac{\pi}{180})}\right)}{\pi} \right) \cdot 2^{z-1} \right\rfloor \tag{7.1}$$

and

$$lon = \frac{x}{2^z} \cdot 360 - 180, \quad lat = \arctan\left(\sinh\left(\pi - \frac{y}{2^z} \cdot 2\pi\right)\right) \cdot \frac{180}{\pi} \tag{7.2}$$

where the last equations relate to the upper-left coordinate. The grid is chosen in such a way, that zooming in by one level, breaks up a single tile in exactly four smaller images. Therefore, the number of tiles required quadruples each level and both x and y range from 0 to 2^z.

7.2 Offline Tiling

This section mainly focuses on the use case where no time and space restrictions are given. Since the result set will always be the same, plotting can be prepossessed very efficiently and with high degree of detail. This allows exploring the full dataset on any level without having any processing bottlenecks.

7.2.1 Architecture

To extend the frontend by a static tiling, a tile layer was added to the map. It already handles the mapping of the current position on the map to the corresponding requesting longitude-latitude pairs and the (x, y)-positions indexing the grid cells. An additional *pistache* server has been set up for the purpose of answering tile requests, which are sent automatically by the map tile layer. On the backend side, handling the request becomes trivial: The stored tiles are checked for a match and if so, the correct tile-image is returned. Following a standard convention [41], tiles are grouped together by z and x coordinates by stacking them into folders.

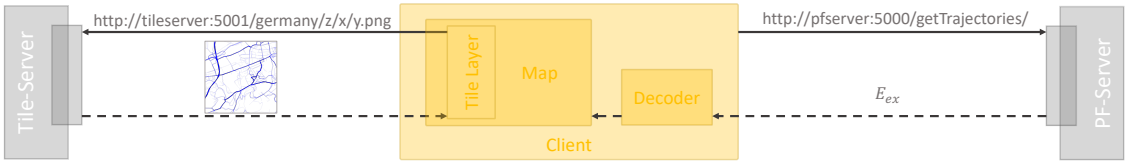


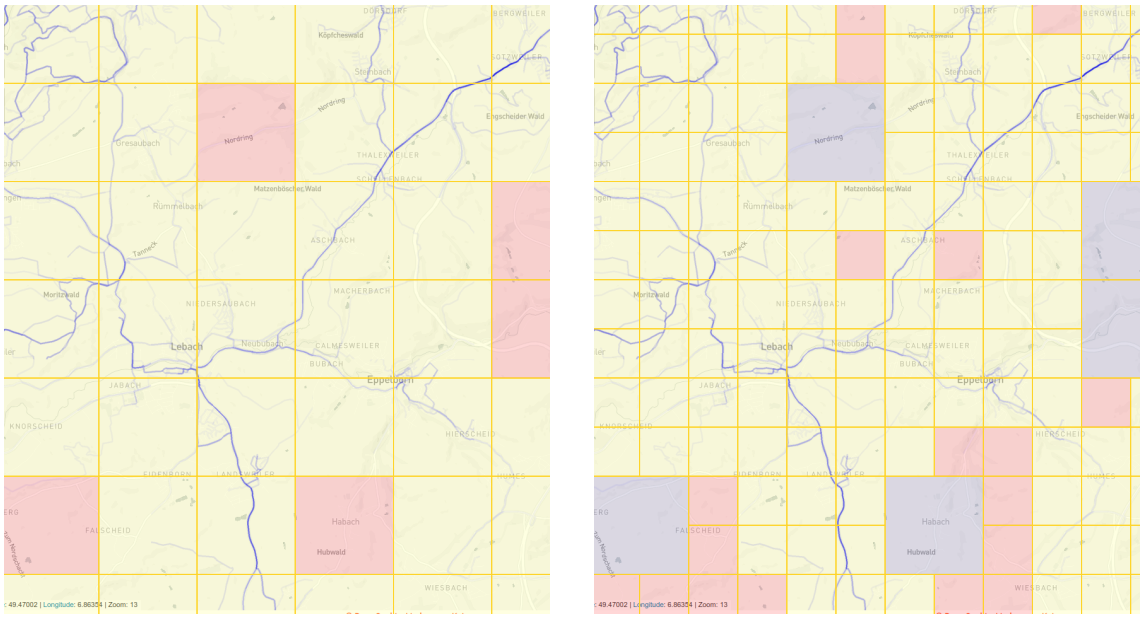
Figure 7.1: Overall setup explained graphically: An additional tiling server provides data requested by the tiling layer. PF requests are still possible.

For generating the files, the image library OpenCV [57] was integrated, which supports simple drawing operations and multiple image encoding methods. As a first step, the bounding box for the graph was calculated on the coarsest level to determine the cells on the top-left and bottom-right of

the grid. Now, for each cell within this range, a tile is plotted and stored. This can be repeated for all finer layers. One could plot a single tile naively by iterating through all paths, creating a image layer for each path, extracting the edges based on the current level using unpacking, and plotting this path edge-wise. Finally, all layers are merged. If only one layer was used, paths would have overwritten their plottings. Hence, using multiple layers is important, this allows for adding layer values up which in turn highlights more frequently used paths in darker tones, while more rarely used paths stay less saturated. To further improve speed, only one color channel is used while aggregating. Right before the images is stored to disk, this usage-channel gets transformed into a color-image.

7.2.2 Adaptive Search

For larger zoom scales, each tile includes many paths. The finer the scale gets, the more details are plotted, but the smaller the bounding boxes become. Iterating over all cells, however, is not required for the following reason: If the bounding box $B_{l,x,y}$ used to create tile on level l does not contain any path to plot, clearly its children $B_{l+1,2x,2y}$, $B_{l+1,2x+1,2y}$, $B_{l+1,2x,2y+1}$ and $B_{l+1,2x+1,2y+1}$ on the next finer level will also be empty. The idea is visualized in Figure 7.2.

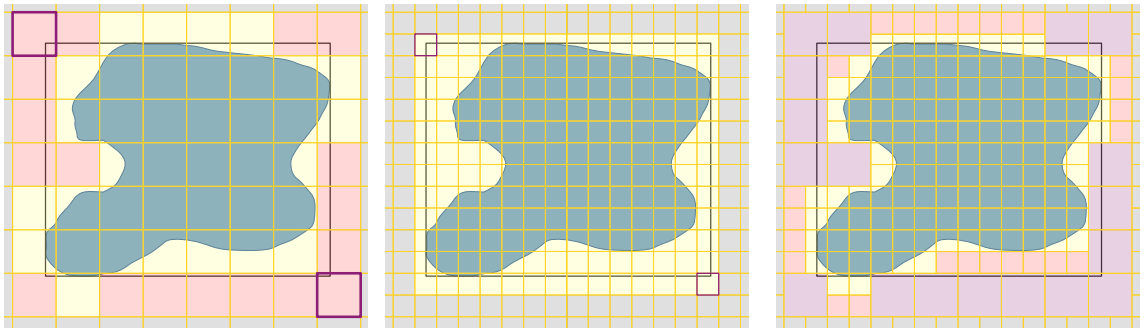


(a) level l **(b)** level $l + 1$

Figure 7.2: Showing plotted paths data with (artificial) grid on the Saarland Dateset. All red colored cells are excluded from further refinements in subsequent steps. Areas colored in blue indicate excluded cells from previous rounds.

For the same reason, all further children will be empty, too. To avoid traversing cells whose parents were already empty, the implementation keeps track of which cells have a non-empty intersection on level l and runs the new round on level $l + 1$ with this cells only. The adaptive structure pays out especially for input graphs which are not well represented by their bounding box, meaning they

cover only a small share, or if there are many spacial holes in the graph, i.e., areas where no path passes through. This is illustrated in Figure 7.3. An example for this could be a large mountain range or sea, where no trajectory have been recorded yet.



(a) Naive and adaptive on level l **(b)** Naive on level $l + 1$ **(c)** Adaptive on level $l + 1$

Figure 7.3: Comparison of visited cells on two consecutive levels: Initially, all 56 cells are iterated through on starting level l. The anchor-cells at top-left and bottom-right were highlighted. The adaptive methods, however, marks some cells as empty (red) and skips them for the next round. It therefore only sweeps through $(56 - 18) \cdot 4 = 151$ cells, while the naive approach visits $12 \cdot 14 = 168$. This difference keeps increasing on finer levels the more the shape gets adapted.

To illustrate the practical advantage compared to the naive implementation, some measurements were taken when constructing the tiles for both methods: The number of visited cells nearly quadruple each level for the naive method visiting the cells independently from the previous round while the adaptive method keeps pruning future cells search paths. Note that the hierarchical adaptive method visits slightly more tiles at the beginning. This is because some tiles are naturally pruned by the naive method because of the refinement of the grid. It is illustrated by Figure 7.3, when transitioning from sub-figure (a) to (b).

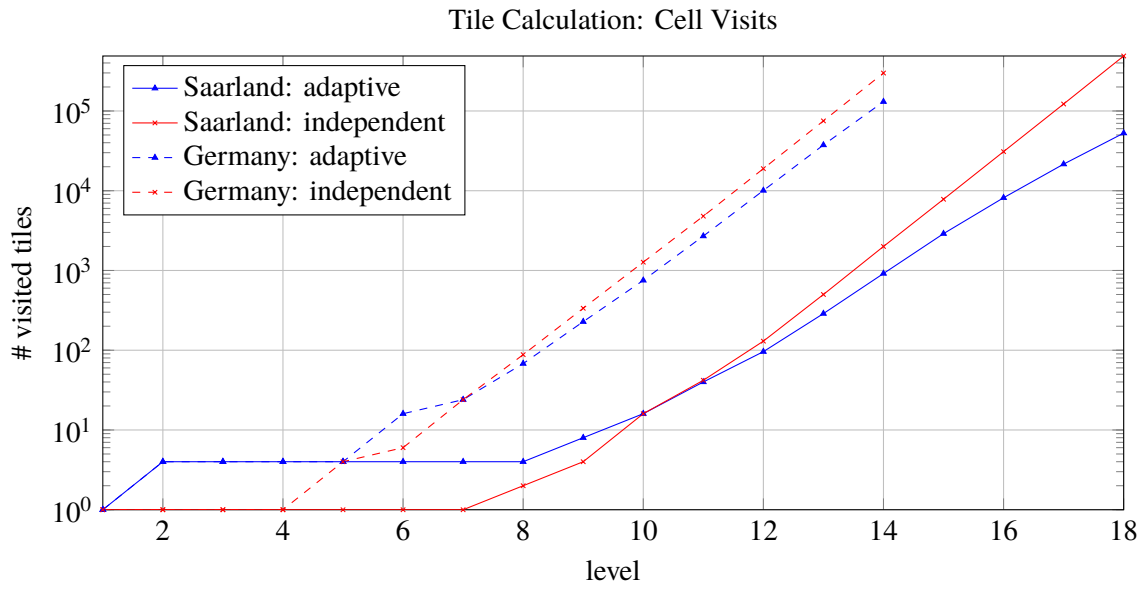


Figure 7.4: Comparison of visited cells using naive traversal (red) its adaptive counterpart (blue). Note that the ordinate is log-scaled due to the exponential growth in tiles visited.

Figure 7.4 shows the number of tiles which were visited on each layer for the two different methods. While the difference seems to be small, the total number of accumulated visits underlines the significant speedup: For generating the full Saarland Dataset tiling, 660 791 cells are visited in total for the naive method. Using an adaptive grid reduces the number of visits down to 86 769, which is a reduction by 86.9%.

7.2.3 Path Skipping

So far, an optimization only in the number of traversed cells has been implemented. Another inefficient implementation is the creation of layers for each path while plotting: Especially for larger zoom scales, only very few paths are part of a single tile. Unpacking and copying all path data is not required if the path will not be visible anyway. To skip this expensive operation, a bounding box for each path is created on startup. This bounding box is then used on tile-creation time to check for empty intersection with the tile-bounding box. If so, the path can be skipped. The path boxes can be calculated efficiently in a recursive manner, similar to Section 5.1.2: Each shortcut's bounding box results form its childrens' boxes using min and max operation on the bounding coordinates. Plain edge boxes are trivial to create. In case of the Saarland Dataset, the creation time for the layers $6 - 12$ reduced by a factor of four, for the creation range $9 - 14$ by a factor of 12.

To further speed up the creation time, the fixed input list can be utilized: because the cells on one level can be visited in any order, parallelization was implemented. Again, a solid reduction can be achieved, ranging from 38 to 46% (using test cases Saarland level 17 and levels $9 - 14$ respectively).

7.2.4 PATHFINDER based

Even faster than testing paths sequentially are hierarchy-organized search structure retrievals, like the one PF makes use of. Therefore, the Path-Skipping algorithm, which served as an appropriate benchmark, is replaced by PF, which now takes care of the retrieval of matching paths for each tile. On our two core CPU, compared to path skipping, this reduced the response time by approximately 32% calculating the tiling on level 17.

The refinement considerations about the tile generator are concluded by an interesting marginal node, regarding adaptive refinement: It turned out to be faster to calculate level 17 by first calculating its smaller hierarchical predecessors 14, 15 and 16, instead of starting at this layer directly. The obvious reason is that layer 14 is $4^3 = 64$ times smaller than layer 17 and prunes the cells search for the next levels significantly. This again shows the outstanding advantage of adaptive refinement.

To sum this section up, a significant increase in the creation of tilings was achieved by searching adaptively based on the last layer's results and integrating PF. The section is closed by showing a timing overview for the methods presented in their chronological order. All measurements were takes on the Thinkpad, as stated in Section 3.4.2.

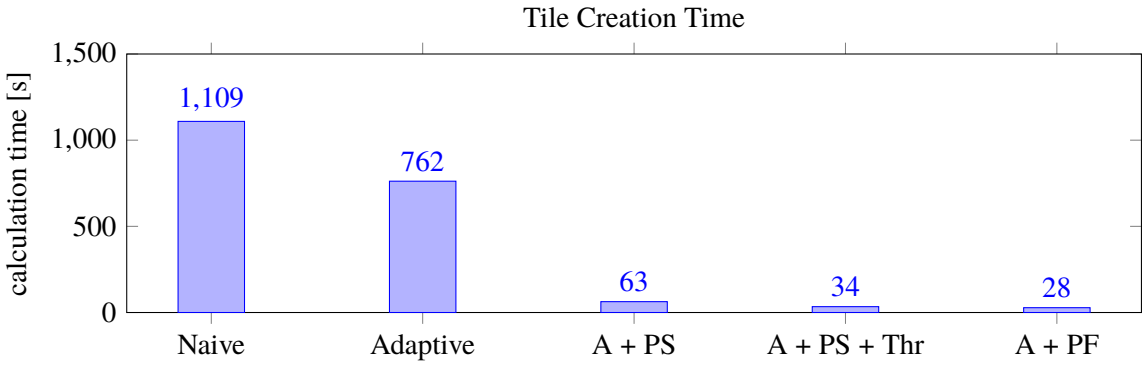


Figure 7.5: Time comparisons: Calculating tiles for levels 9 – 14 on the Saarland Dataset using adaptive grids (A), path skipping (PS), threading (Thr), and PATHFINDER (PF).

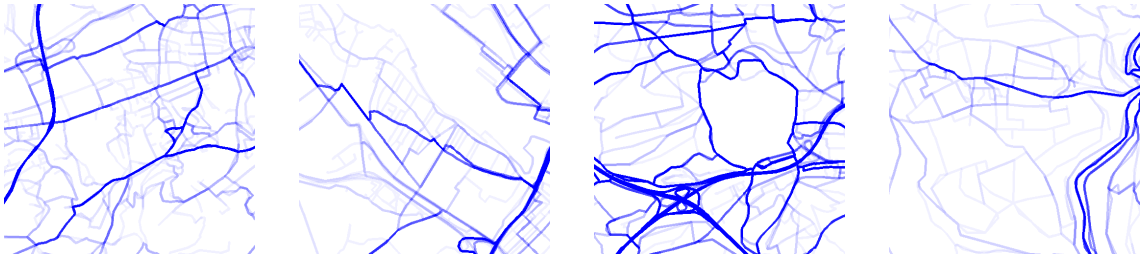
7.3 Online Tiling

In the last section, a tiling server was used to avoid expensive calculations on requesting time: All tiles are calculated in advance, they only have to be transmitted when requested. This section tries to motivate the use of tiles more abstractly by enforcing a lightweight, plotting-free client.

7.3.1 Potential

The underlying idea is rather simple: While plotting on client side is slow and vectorized data has to be transmitted, plotting on the backend side is fast and the result can be sent as a raster graphic. This introduces some degree of result-independent returns, because the image size does not increase linearly in the number of paths it shows. Examples are shown in Figure 7.6. Also, overlays of paths

are transmitted as the saturation of pixels rather than as different result set entries that have the same coordinate list. Transmitting images is also advantageous if complex paths has to be drawn: It is common knowledge that a lot of points are necessary to describe a jittering curve. This also applies to trajectories with *zigzag*-shaped characteristics. Straight line trajectories, however, outperform its rasterized variant. Another advantage of using tiles over transmitting raster files is a batching-like transmitting behavior. While all requests are sent automatically, the smaller responses ones arrive first, providing a quick feedback to the user. Further, details are refined on demand, because no tile of higher resolution is loaded, until the user requests them by zooming in at the corresponding position.



(a) 43.4kB **(b)** 40.3kB **(c)** 39,2kB **(d)** 35,5kB

Figure 7.6: Showing Saarland Dataset tiles of different complexities, all ranging between 35-45 kB in size.

7.3.2 Implementation

To illustrate the concept, a small demonstration has been developed. This proof of concept implements all required components showed in Figure 7.7.

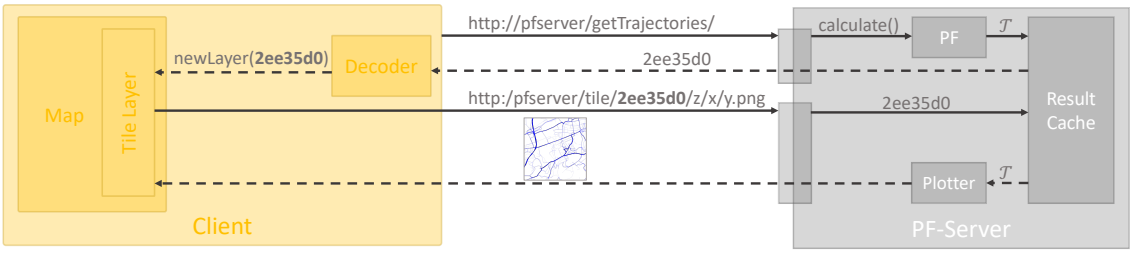


Figure 7.7: Architecture for online tile generation.

When the user triggers a calculation query, the PF instance calculates all matching results. Instead of exporting the result set, it is stored in a global cache and linked with an identifier. Latter gets returned to the client, which includes a new tiling layer. In contrast to the offline case, where the base tiling URL was static, this time the received reference becomes part of its domain path. The tile layer automatically checks for tiles to be included and requests them. Finally, the PF server receives the requests, creates the plots according to the sets obtained from the cache, and sends back the final images which are embedded into the tile layer.

Dataset	Levels	Calculation Time	Storage Requirement	# Tiles
Saarland	1-18	1:05h	1.40GB	159 337
Germany	1-13	1:13h	0.65GB	45 970
	1-18	>10h*	1.50GB*	>250 000*
Europe 500k	1-12	5:09h	1.40GB	207,915
	1-18	>30h*	>20GB*	>20 000 000*

Table 7.1: Tile creation statistics. The marked values represent estimates only.

To make sure that the server cache does not run out of memory, only a fixed number of result sets are stored. They are managed in a cyclic buffer, such that the oldest cache entries are overwritten first in case of to many requests.

7.4 Discussion

This final section discusses the method's characteristics and gives a final recommendation based on that. To preface the argumentation, Figure 7.8 shows screenshots of typical results.

For the implementation, a tile size of 512×512 pixels were chosen. Table 7.1 shows recorded tile statistics for preprocessing. Assuming a desktop application of size 1920×1080 pixels, which was by far the most frequently used desktop resolution in Europe last year [53], and subtracting the left and right navigation menu, nine tiles are usually needed to fully cover the map's tile layer.

Figure 7.9 visualizes the data sizes for transmitting tiles on different zoom levels: Regarding the Saarland data, in the beginning, tiles are small, since the shape of Saarland is comparably small. Increasing the level size leads to an increase in the average tile size which finally peaks at level nine. From that point on, the size decreases again. This is because only a few paths per tile are visible due to large zoom scales. The tiling for the Germany Dataset shows a qualitatively very similar behavior, except that it peaks earlier at level seven.

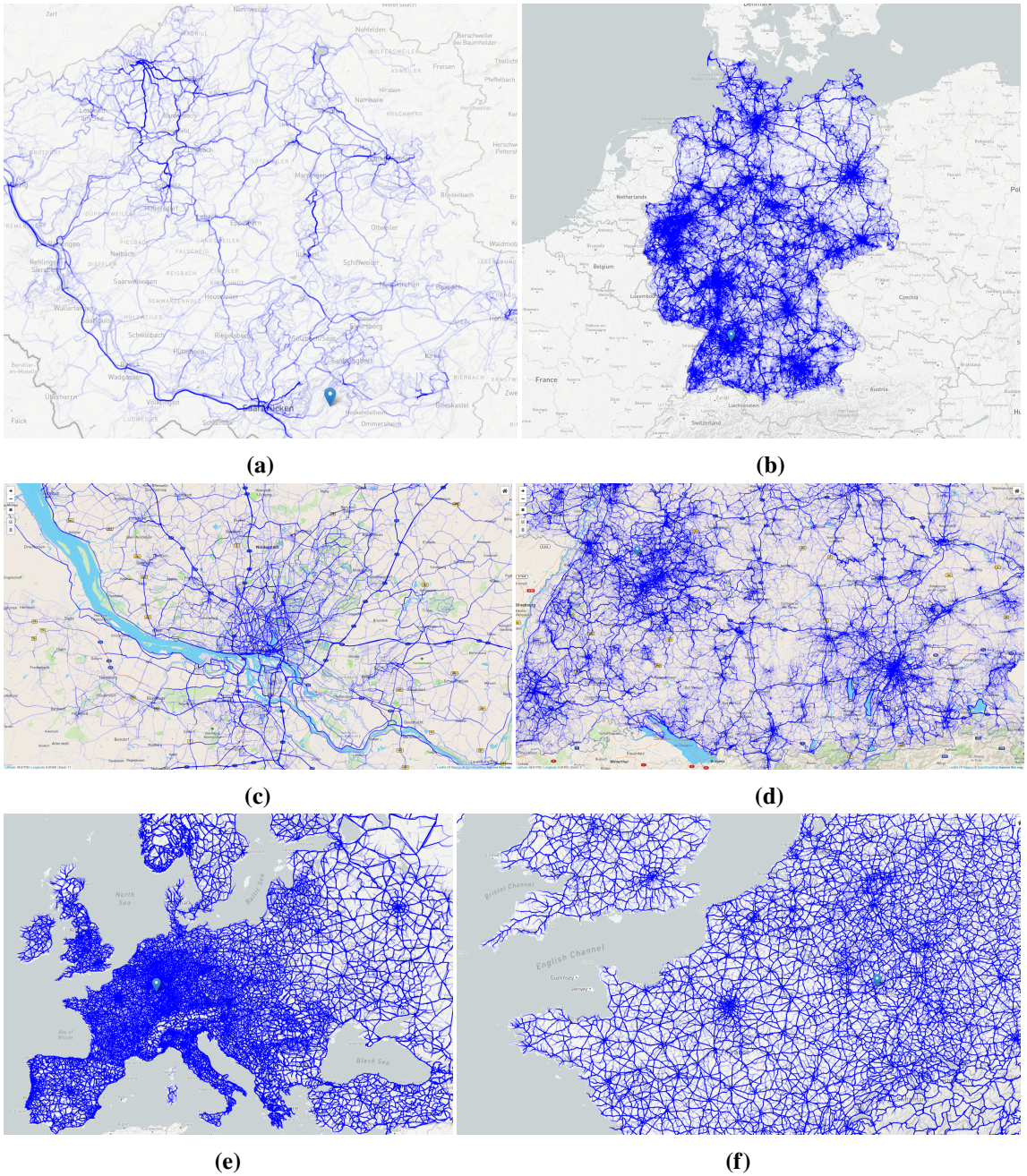


Figure 7.8: Showing typical result sets using precomputed tiles. (a) using Saarland tiles, (b), (c), and (d) using Germany tiles and Europe dataset tiling used in (e) and (f).

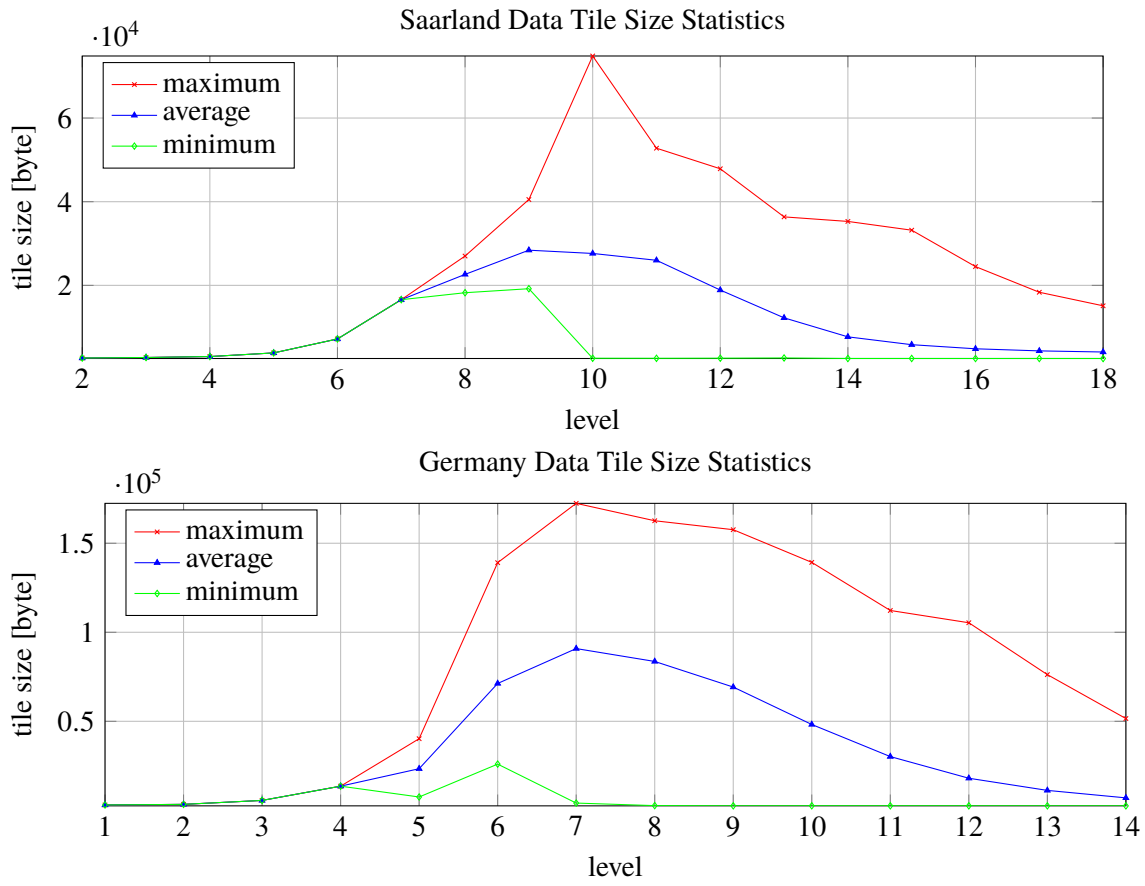


Figure 7.9: An overview of final tiles sizes on the Saarland and Germany datasets.

The plots clearly show, that tiles are typically small: Even in the worst case, when requesting the nine largest Germany tiles on level seven, at most 1.48 MB have to be transmitted. For the average case on the critical level seven, 0.78 MB of tile data is received in total. Although the data sizes are rather small, the tiles are rich of details, as shown in Figure 7.8.

Another advantage of using tile layers for displaying data resulting from heavy calculations is that it comes with caching by default. Panning the map slightly only reloads areas which have not been transmitted yet. Moreover, when a zoom-in followed by a zoom-out operation is performed, the data does not need to be transmitted again in the latter case, in opposition to the earlier presented methods.

Furthermore, the implemented tiling shows a high degree of interactivity. The user sees an upscaled version of the tiles of level t when zooming in to level $t+1$, but different to the vector-versions, single tiles update one after another, independent from each other. This again decreases the user-response time and enhances usability.

According to Definition 1, tiling supports macroscopic analysis very well, but has the drawback that it does not, at least in the current implementation, support single-trajectory selection which is required for low-level use cases. A possible solution might be the inclusion of another raster layer on top for reloading data lazily, see Section 8.5.

To sum up, using tiling for displaying render results yields highly detailed views on any scale at low transmission cost. Details are loaded on demand only for requested areas. Once the tiles have been calculated, tile servers are fast to setup and start. Moreover, using tiling allows very different end users to access the map, the lightweight implementation works fast and independent from the user's frontend device. In the case of static tiling, however, no time and space restrictions can be set. A possible solution might be online tiling, where requests are calculated on the server side and only raster images are transmitted. Also, building a high resolution tile set takes a fair amount of time and consumes a lot of disk storage. Additionally, many tiles have to be recalculated if the trajectory dataset changes. More research has to be done, to investigate whether online tiling could be combined with methods like pruning or EBPF to combine fast calculation with small transmission sizes and lightweight clients.

8 Future Work

This chapter touches on the presented methods to show possible extensions and open questions for further research.

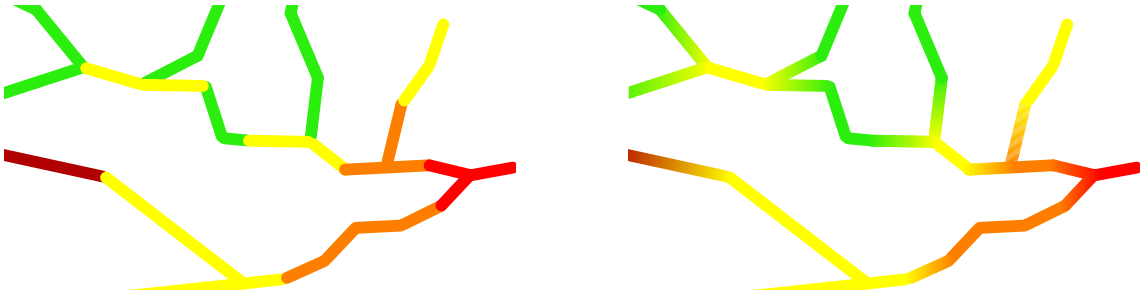
Decoder-Encode Wrapping

To increase transmission speed method-independently for larger packages, a message decoder can compress the json-response on the server side before sending and a client-side counterpart decompresses the received message. Further investigations must be carried out to decide at what size (de-)compression is worthwhile.

8.1 Segment Graph

Smooth Transitions

The segment graph algorithm merges edge to edge strokes according to their usage. Each stroke is plotted on its own, connected to other strokes, according to the connection graph's edge set. This, however, produces sharp transitions between neighboring strokes. to enhance visibility, edge strokes should be colored smoothly with blended colors determined by their neighboring strokes. A structure, similar to a 1D heatmap would be the result. An example is shown below in Figure 8.1.



(a) Current **(b)** Smoothed

Figure 8.1: Comparing the current sharp transitions with a smoothed version to enhance visibility.

Sorting With Respect to Edge Colors

Section 6.3 demonstrated a massive speedup if edges can be plotted in groups. The same technique can be applied for the segment graph if edge strokes are grouped together with respect to their color. It should be noted, however, that this change would prevent the plotter from adding mouse-hover and on-click events.

Graph Strokes Simplification

By definition, each edge stroke only represents a fixed composition of trajectories. Analogously to the SUA, strokes can be simplified before transmitting the graph. Arguing the same way as with sensitive unpacking, this does not change the overall topology, but reduces complexity.

8.2 Batched Transmission

Optimal Transmission Order

In Section 4.4, node orderings were presented and evaluated. It turned out that PLOU performed best on the input data. A very interesting research question would be, however, how well an optimal strategy would perform. This would give a better understanding of how effective the presented methods are. Subsequently, it is also unclear how an optimal strategy can be found in general. Naively, all permutations could be checked, but is there an overall method polynomial run time? Are there (greedy) polynomial-time approximation schemes which produce a solution that grantees to be very close to the optimum?

Aggregated Transmission

The chapter on batching dealt with transmitting points on a single trajectory. A possible extension transmits a group of trajectories. After unpacking the input trajectories to a certain level and calculating the union, edges are refined with respect to their usage count. Edges used by many trajectories are preferred over single-used ones. This strategy would nicely extend the EBPF methods, which already perform the required preprocessing.

Prioritized Updating

If a request is answered using batching, a queue of open paths is stored and one trajectory is refined after another and again enqueued if not all points have been transmitted yet. To improve this idea, a different approach could be chosen: replacing the queue by a priority queue allows more advanced updating rules.

The overall communication sketch is depicted in Figure 8.2: To begin with, the clients starts requesting a spatio-temporal query, which results in a list of matched trajectories. The server caches the result and answers with an initial raw batch for all paths. The data is transmitted and loaded

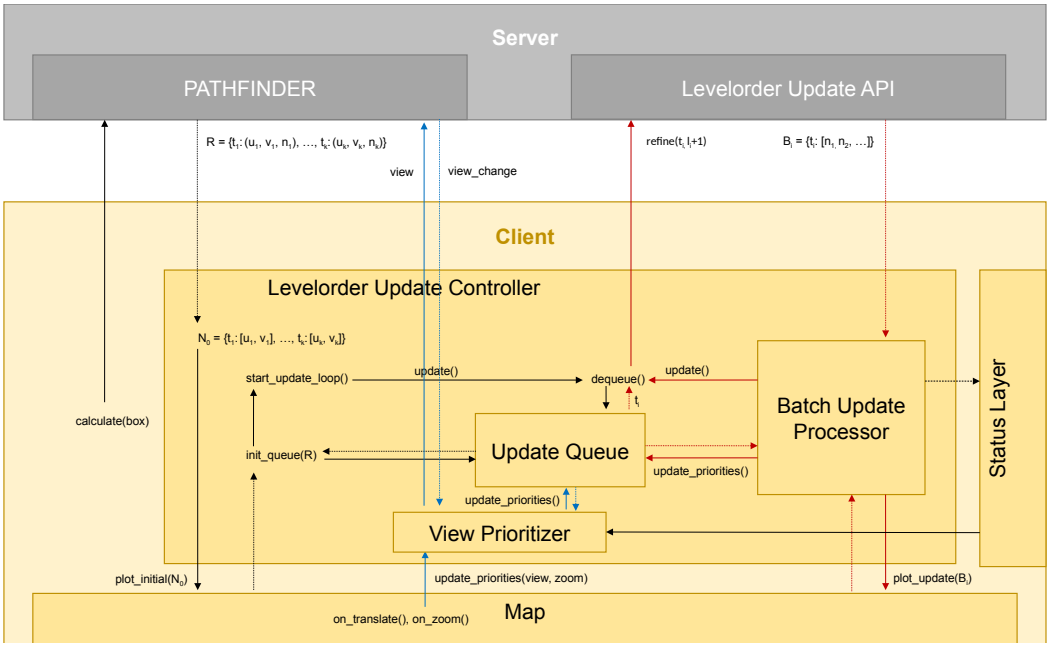


Figure 8.2: A novel architecture showing a more advanced updating procedure.

into the map. Similar to the current implementation, the elements are pushed to an update queue storing all unfinished trajectory objects, but this time certain priorities are assigned. The trajectory's length measured by the number of points, or its intersection-ratio between the path-bounding box and the current view box are possible weights. This allows longer and typically more complex paths to update first, while updates for paths which are only partially visible are postponed until the very end. Whenever an update is received, the priority of the respective trajectory object is change accordingly. This architecture can be extended to integrate user interaction: When a user translated the map or zooming was performed, the priorities of the remaining paths are updated, e.g., trajectories whose bounding boxes do not overlap with the view's box anymore are postponed to the very end, while paths which have not been updated yet, since they were not visible, are brought forward. To inform the user about the transmission process, i.e., how many open trajectories are still enqueued, an interactive status layer is integrated into the frontend. This would allow to set certain download thresholds, for example, the user could specify to download only 50% of the points for each trajectory. If the user is not satisfied with the result, the threshold can be adjusted, which triggers the prioritizer to enqueue the trajectories again.

8.3 Graph Layer Pruning

Fixed and Sorted Cutoff Levels

The process of edge filtering can be accelerated if the number of allowed cutoffs is restricted: if there are only k possible cutoff levels, where k is much smaller than l_{max}, e.g., $k = 4$, edges could be grouped together in a preprocessing step.

To extend this idea, edges should be sorted for each trajectory by increasing edge-level, so only k pointers must be stored, referencing the cutoff positions.

Alternatively, a binary search on the sorted edges can be implemented to find the correct split index in logarithmic time. Once this index has been determined, no further filtering-check has to be performed due to the sorting. To speedup the merging further, m partial heatmap results $H_1, H_2, \ldots, H_m$ can be pre-calculated for cutoffs $c_1 < c_2 < \cdots < c_m$. On request time, the smallest pre-calculated sub result H_c must be found and the remaining edge-updates are applied. An example is given in Figure 8.3: cutoff c is requested, therefore, $H_c = H_1$ is loaded because $c_1 < c < c_2$ and only heatmap cell counts for $[14]$ and $[12, 4]$ have to be updated.

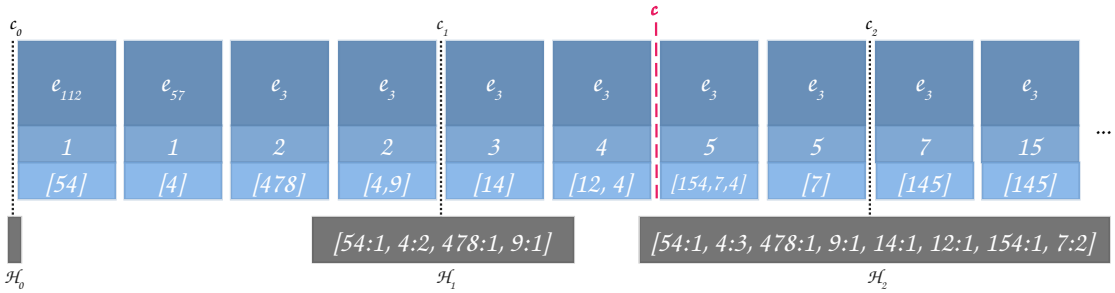


Figure 8.3: Edge sorting enables preprocessing for faster heatmap merging. Edge-IDs are colored in dark blue. Their respective edge levels and grid cell lists are drawn using lighter blue tones.

Include Batching

In the case where $|E^+|$ is very large, transmitting the edge set in a batched fashion, similar to plain batching, can be integrated to reduce waiting times on client side. Similarly, rough heatmaps could be calculated and returned first, since they are faster to build, and smaller in size. Once the heatmap-preview is transmitted, calculating the next finer level is triggered.

Heatmap Pruning

In the implementation of the pruned graph, only edges are returned which are visible within the user's device screen and the result edge set is cached on server side. When the map is panned, an updated view is loaded to avoid transmitting edges outside the view. A very similar concept could be introduced for the heatmap as well, which would noticeably decrease the response time for the first update.

8.4 Edgebased PATHFINDER

Manual Bin Selection

When binning is activated, the server groups the result edges with respect to theirs usage into k bins. Afterwards, all bins with edges included are transmitted. To increase responsiveness and to lighten the client processing load, a future implementation only returns the first layer. All other layers are not visible by default and are reloaded from the server when the user activates the receptive check box. Besides the speedup for transmitting the initial request, this also allows the user to switch single layers on- and off. A suitable use case would be to only display edges with the most or least visits within the selected bounding box.

Dynamic Bin Cutting

Reducing the number of colors allowed comes down to finding a suitable function $binned : C \rightarrow B$ with $C \subset \mathbb{N}$ being the unique counter values and $|B| \leq b$ the target values allowed for binning. In the implementation explained in Section 6.3, splits are introduces at the values $\min(C) + \left\lfloor i \cdot \left(\frac{\max(C)-\min(C)}{b-1} \right) \right\rfloor$ for $i = 0, \ldots, b-1$ and each edge is assigned to the group having the last smaller split value. Finally, each group gets the value of count-wise smallest element assigned, which is at least as big as the split value. This ensures to include $\min(C)$ and $\max(C)$ in a separate bin, if $b > 1$. However, due to the static range, some bins might by empty. Although the method requires low running time, more efforts for choosing the correct bins will boost the quality. A reasonable approach will also take the input distribution into account, where small count values occur typically often and high usage values rarely. The distribution for the unbinned request on Dataset 5 is shown in Figure 8.4.

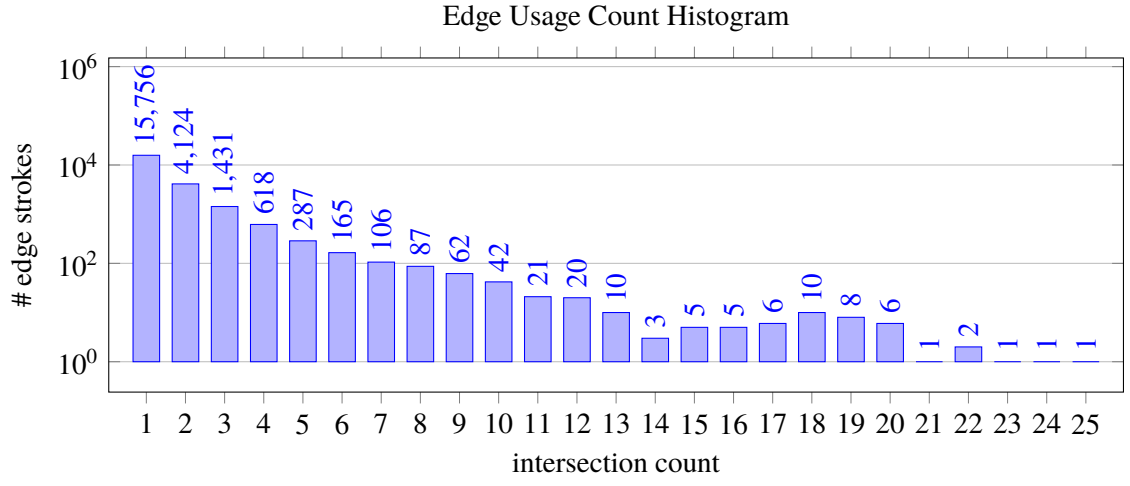


Figure 8.4: Edge usage distribution for the response of the Dataset 5 from Section 6.3. Counts for intersection values 26 to 41, whose values were smaller than two, have been truncated for better visibility. Note that the ordinate is of logarithmic scale.

Since most of the edges occur less then ten times, to increase contrast, more bins should be spent on lower counts and only a few bins on the very high-frequent counts. An input-distribution dependent binning function will address this issue, for example by applying histogram equalization, e.g., as in [30, 46].

Parameterized Rendering Function

If weighted edge plotting is enabled, the EBPF returned strokes are outlined according their absolute usage numbers, i.e., edges which are part of many trajectories are plotted less transparent or thicker, than edges only used a few times. When a static function is used, not all datasets are supported equally well: While on small datasets, the maximum intersection count for a request typically does not exceed a couple of dozen, e.g., 41 for the filtered Saarland Dataset, dense graphs can have many thousands. The current implementation uses a log function for scaling values accordingly, however it can't be guaranteed that the full range $[l, 1] \subset \mathbb{R}$ is covered, where l is a minimum opacity value for paths only used once. A future implementation can return the maximum intersection count for the whole graph as an additional parameter. This allows the output weights to be scaled accurately, regardless of the underlying path collection. All edges with usage count one will have opacity l, all edge strokes with maximum intersection count will be fully visible, i.e., having an opacity value of one.

Range-Based Customizable Bin Plotting

It was already mentioned in Section 6.3 that small binning values allow for a clear separation between frequently and rarely used edge strokes. An illustration follows below in Figure 8.5. To even extent this method, edges could be grouped into sections defined by the user with individually defined colors ranges. As a concrete example, the user could request for the top ten percent most frequent edges and the top ten least used edges and plot these edge strokes using two bins in a red and green tone receptively. All edges with counts in-between are colored in a neutral color. This yields a user friendly and customizable exploration method.

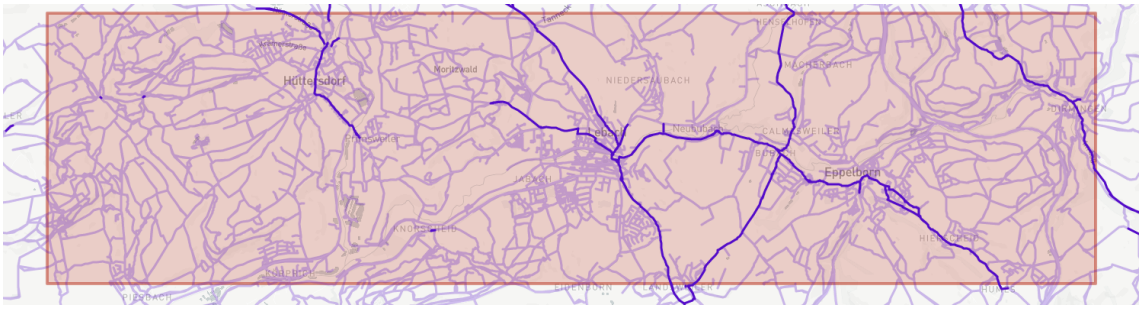
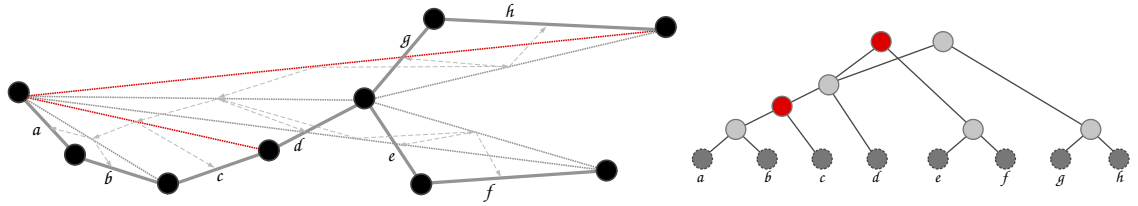


Figure 8.5: Using a few bins only yields a good separation between frequently used edge strokes and edges rarely in use. In the Saarland graph example above, three bins are used.

Prevent Subedge Unpacking For SUA

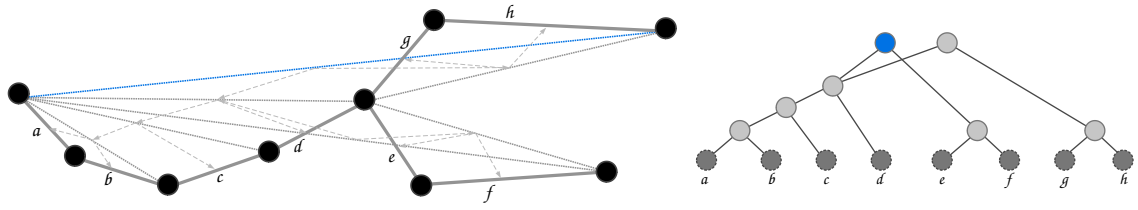
Sensitive Unpacking yield good results when several input edges have non-empty intersections. In the special case, however, where two edges e_s and e_p are part of the input set, unnecessarily small edges are returned. To illustrate this concept, Figure 8.6 shows an input instance where a subedge is present.



(a) **(b)**

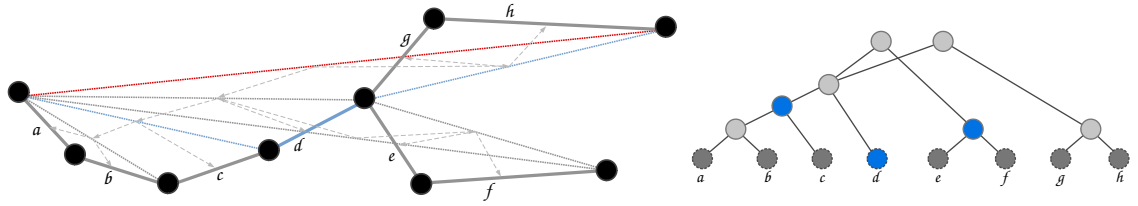
Figure 8.6: A input scene where EBPF returned a subedge.

Although the parent edge would be enough to be returned (see Figures 8.7a and 8.7b), unpacking is performed and edges of smaller level are added to the output set (Figures 8.7c and 8.7d). To solve that problem, in a first run sub edges should be removed. Note, though, that this only applies to the unweighted case. For the weighted case, splits have to be made to allow for different coloring. In this concrete example, edge d would be colored slightly darker to account for the different usages.



(a) Desired output scene

(b) Desired output hierarchy



(c) Actual output scene

(d) Actual output hierarchy

Figure 8.7: Comparison of the optimal desired solution (a) and (b) and the current implementation (c) and (d).

Handle Independent Connection Components in Parallel

For unpacking edges sensitively, first a marking sweeps downwards is performed to find overlapping subtrees and a second sweep is initiated to eliminate unnecessary unpacking. Given a graph with multiple connection components, e.g., islands or continents, both sweeps can be done independently on each sub-component. Since edges are not shared between independent connection components by definition, no overlapping subtrees will be found. If edges are grouped together and sorted by level only within their components, sweeps can be efficiently accelerated. Different from the current implementation, requests on small islands, for example, would be very cheap to answer.

Avoiding Full Sweeps

Due to the sweeps in SUA, output sensitivity gets lost: Even if only very few edges are returned by EBPF, still all edges are traversed during sweeping up and down. Instead sweeping downwards, a priority queue, initialized with edges from the EBPF and their corresponding levels, can be used to maintain the correct traversal ordering, while only relevant edges are visited. If all traversed edge are pushed to a stack in the order of visit, sweeping upwards can be replaced by visiting the nodes in the exact opposite order. Collecting edges can be integrated in the second upwards-sweep: If an edge e differs from the values of its children, both are exported if not empty and e gets 0. To make sure that no edge is reported twice, the exported children are also marked 0. By replacing the sweeps using a priority queue and a stack, much faster responses for small input sets are expected.

Resolving Unidirectional Edges

The problem of having two edges between two adjacent vertices was already explained in the discussion section of Chapter 6. To avoid that problem, a possible solutions would be to re-parse the trajectories to build a compressed path data structure, where only edges of increasing vertex ID are used. Whenever trajectories are re-assembled in a postprocessing step, non-matching (i.e., reversed) edges have to be flipped. This way, only one well defined edge for each neighboring vertex pair is chosen, regardless of the trajectories' orientations.

8.5 Tiling

Plotting Speedup Techniques

In Chapter 6, a concept was introduced that solely base on edge-operations instead of trajectories: EBPF. Since each tile usually only shows a very small extract of the whole trajectory's shape, loading a full trajectory is a waste of resources. Think about a level-18-tile showing only a few edges from ten trajectories: Instead of iterating through all root edges of the ten trajectories and perform unpacking, plotting the edges directly will yield a noticeable speedup. In a future work, these two methods can be combined nicely to replace PF- by EBPF requests. This can be extended to the online-scenario, where fast processing times are even more critical.

There is also potential for further speedup while creating the tiling-images using adaptive search: It makes sense to skip areas where no trajectories are present as early as possible, but generating tiles level by level slows down the plotting, because paths are unpacked from scratch each time a tile is created. Instead, consider the case where a tile, which was rendered on level l, is used to also render the levels $l + 1$ by analyzing which of the four sub-tiles use which trajectories. Then, the already partially unpacked paths can be passed as a reference to the sub-tiling unpacking steps for further unpacking. In the best case, each path is unpacked only once.

Alternatively, tiles are generated path-by-path: For each path, the intersecting tile cells on each level are determined. Then, the path is plotted on each level. If no path has used tile $c_{i,j,z}$ yet, it is created. Otherwise, the tile gets updated. This method ensures, that each path is only fully unpacked once.

While the two previously presented methods require rather extensive implementation work, caching of partially unpacked trajectories is worth integrating at low cost. Introducing a fixed-sized cache for unpacked trajectory coordinates already formatted as point list for the current level and tile, is also very useful for its neighboring tiles, since there is a good chance that they have a firm amount of trajectories in common.

Include Information on Demand

Using vector-based methods to plot results requires to create polyline objects to be displayed on the map. These objects can be extended by *onclick events* to allow for interactive single-trajectory analysis (see screenshot Figure 2.7). In contrast to vector methods, this is not possible for tile layers by default. To allow for exploration methods like this, onclick events on the map could be caught and forwarded to the server, where the relative click positions have been converted to absolute longitude-latitude coordinates. On the server side, matching trajectories could be traversed and returned, but this time as vector objects, since the returning set is typically very small. Finally, the retrieved trajectories are added to a new overlay-layer. This would allow to display additional meta-information about the trajectories, similar to the original work, with the only difference that loading would take place lazily on-demand.

Bibliography

[1] V. Agafonkin. *Leaflet - an open-source JavaScript library for mobile-friendly interactive maps*. https://leafletjs.com/. Accessed: 2021-10-26. 2021 (cit. on p. 18).

[2] V. Agafonkin. *TileLayer*. https://leafletjs.com/reference.html#tilelayer. Accessed: 2021-11-05. 2021 (cit. on p. 100).

[3] *AIS Data as Trajectories and Heat Maps*. New York, NY, USA: Association for Computing Machinery, 2021, pp. 431–434. ISBN: 9781450386647 (cit. on p. 15).

[4] V. Alexiadis, K. Jeannotte, A. Chandra, A. Skabardonis, R. Dowling. *Traffic analysis toolbox volume i, traffic analysis tools primer*. Tech. rep. United States. Federal Highway Administration. Office of Research, 2004 (cit. on p. 17).

[5] G. Andrienko, N. Andrienko, W. Chen, R. Maciejewski, Y. Zhao. "Visual analytics of mobility and transportation: State of the art and further research directions". English (US). In: *IEEE Transactions on Intelligent Transportation Systems* 18.8 (Aug. 2017). Publisher Copyright: © 2000-2011 IEEE. Copyright: Copyright 2017 Elsevier B.V., All rights reserved., pp. 2232–2249. ISSN: 1524-9050. DOI: 10.1109/TITS.2017.2683539 (cit. on p. 19).

[6] N. Andrienko, G. Andrienko. "Visual Analytics of Movement: An Overview of Methods, Tools and Procedures". In: *Information Visualization* 12 (Jan. 2013), pp. 3–24. DOI: 10.1177/1473871612457601 (cit. on p. 19).

[7] L. Baur. *Videos*. https://bitbucket.org/baurls/ma_baur_pathfinder_visualization/src/main/docs/videos/batched_transmission/. Accessed: 2021-11-17. 2021 (cit. on p. 64).

[8] S. Bekas. "Visualisierung großer Straßengraphen mittels Vulkan". MA thesis. Universitätsstraße 38D, 70569 Stuttgart: University of Stuttgart, Nov. 2019. URL: https://d-nb.info/1205737057/34 (cit. on pp. 18, 29).

[9] V. Buchhold, D. Wagner. "Nearest-Neighbor Queries in Customizable Contraction Hierarchies and Applications". In: *arXiv preprint arXiv:2103.10359* (2021) (cit. on p. 18).

[10] C. Calenge, S. Dray, M. Royer-Carenzi. "The concept of animals' trajectories from a data analysis perspective". In: *Ecological Informatics* 4.1 (2009), pp. 34–41. ISSN: 1574-9541. DOI: https://doi.org/10.1016/j.ecoinf.2008.10.002. URL: https://www.sciencedirect.com/science/article/pii/S1574954108000654 (cit. on p. 15).

[11] M. Chimani, T. C. van Dijk, J.-H. Haunert. "How to Eat a Graph: Computing Selection Sequences for the Continuous Generalization of Road Networks". In: *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. SIGSPATIAL '14. Dallas, Texas: Association for Computing Machinery, 2014, pp. 243–252. ISBN: 9781450331319. DOI: 10.1145/2666310.2666414. URL: https://doi.org/10.1145/2666310.2666414 (cit. on p. 19).

[12] B. team with the help contributors. *Build fast, responsive sites with Bootstrap*. https://getbootstrap.com/. Accessed: 2021-10-26. 2021 (cit. on p. 18).

[13] M. F. individual mozilla.org contributors. *Firefox The browser is just at the beginning*. https://www.mozilla.org/en-US/firefox/. Accessed: 2021-11-15. 2021 (cit. on p. 42).

[14] N. Dianati. "Unwinding the hairball graph: Pruning algorithms for weighted complex networks". In: *Physical Review E* 93.1 (2016), p. 012304 (cit. on p. 19).

[15] J. Dibbelt, B. Strasser, D. Wagner. "Customizable contraction hierarchies". In: *International Symposium on Experimental Algorithms*. Springer. 2014, pp. 271–282 (cit. on p. 18).

[16] T. C. van Dijk, K. Fleszar, J.-H. Haunert, J. Spoerhase. "Road Segment Selection with Strokes and Stability". In: *Proceedings of the 1st ACM SIGSPATIAL International Workshop on MapInteraction*. MapInteract '13. Orlando, Florida: Association for Computing Machinery, 2013, pp. 72–77. ISBN: 9781450325363. DOI: 10.1145/2534931.2534936. URL: https://doi.org/10.1145/2534931.2534936 (cit. on p. 19).

[17] S. AL-Dohuki, F. Kamw, Y. Zhao, X. Ye, J. Yang, S. Jamonnak. "An Open Source TrajAnalytics Software for Modeling, Transformation and Visualization of Urban Trajectory Data". In: *2019 IEEE Intelligent Transportation Systems Conference (ITSC)*. Oct. 2019, pp. 150–155. DOI: 10.1109/ITSC.2019.8917058 (cit. on p. 19).

[18] S. Al-Dohuki, Y. Wu, F. Kamw, J. Yang, X. Li, Y. Zhao, X. Ye, W. Chen, C. Ma, F. Wang. "SemanticTraj: A New Approach to Interacting with Massive Taxi Trajectories". In: *IEEE Transactions on Visualization and Computer Graphics* 23.1 (Jan. 2017), pp. 11–20. ISSN: 1941-0506. DOI: 10.1109/TVCG.2016.2598416 (cit. on p. 19).

[19] D. Douglas, T. Peucker. "Algorithms for the reduction of the number of points required to represent a digitized line or its caricature". In: *Cartographica: The International Journal for Geographic Information and Geovisualization* 10 (1973), pp. 112–122 (cit. on pp. 19, 49).

[20] S. Fenwick, H. Khatri. "THE STATE OF MOBILE NETWORK EXPERIENCE 2020: ONE YEAR INTO THE 5G ERA". In: (2020) (cit. on p. 42).

[21] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee. *Hypertext transfer protocol–HTTP/1.1*. 1999 (cit. on p. 52).

[22] R. T. Fielding. *Architectural styles and the design of network-based software architectures*. University of California, Irvine, 2000 (cit. on p. 49).

[23] M. L. Fredman, R. E. Tarjan. "Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms". In: 34.3 (July 1987), pp. 596–615. ISSN: 0004-5411. DOI: 10.1145/28869.28874. URL: https://doi.org/10.1145/28869.28874 (cit. on p. 83).

[24] S. Funke, A. Nusser, T. Rupp, S. Storandt. *Pathfinder*. https://gitlab.com/anusser/pathfinder. Accessed: 2021-11-12. 2019 (cit. on p. 18).

[25] S. Funke, T. Rupp, A. Nusser, S. Storandt. "PATHFINDER: Storage and Indexing of Massive Trajectory Sets". In: *Proceedings of the 16th International Symposium on Spatial and Temporal Databases*. SSTD '19. Vienna, Austria: Association for Computing Machinery, 2019, pp. 90–99. ISBN: 9781450362801. DOI: 10.1145/3340964.3340978. URL: https://doi.org/10.1145/3340964.3340978 (cit. on pp. 3, 15, 18, 21, 22, 25–27).

[26] S. Funke, N. Schnelle, S. Storandt. "URAN: A Unified Data Structure for Rendering and Navigation". In: *Web and Wireless Geographical Information Systems*. Ed. by D. Brosset, C. Claramunt, X. Li, T. Wang. Cham: Springer International Publishing, 2017, pp. 66–82. ISBN: 978-3-319-55998-8 (cit. on p. 18).

[27] R. Geisberger, P. Sanders, D. Schultes, D. Delling. “Contraction hierarchies: Faster and simpler hierarchical routing in road networks”. In: *International Workshop on Experimental and Efficient Algorithms*. Springer. 2008, pp. 319–333 (cit. on pp. 18, 21).

[28] F. Giannotti, M. Nanni, F. Pinelli, D. Pedreschi. “Trajectory pattern mining”. In: *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2007, pp. 330–339 (cit. on p. 19).

[29] D. A. Huffman. “A Method for the Construction of Minimum-Redundancy Codes”. In: *Proceedings of the IRE* 40.9 (1952), pp. 1098–1101. DOI: 10.1109/JRPROC.1952.273898 (cit. on p. 82).

[30] R. Hummel. “Image enhancement by histogram transformation”. In: *Computer Graphics and Image Processing* 6.2 (1977), pp. 184–195. ISSN: 0146-664X. DOI: https://doi.org/10.1016/S0146-664X(77)80011-7. URL: https://www.sciencedirect.com/science/article/pii/S0146664X77800117 (cit. on p. 116).

[31] A. M. D. Inc. *AMD Vulkan Graphics API*. https://www.amd.com/en/technologies/vulkan. Accessed: 2021-11-12. 2021 (cit. on p. 18).

[32] J. Kim, H. S. Mahmassani. “Spatial and Temporal Characterization of Travel Patterns in a Traffic Network Using Vehicle Trajectories”. In: *Transportation Research Procedia* 9 (2015). Papers selected for Poster Sessions at The 21st International Symposium on Transportation and Traffic Theory Kobe, Japan, 5-7 August, 2015, pp. 164–184. ISSN: 2352-1465. DOI: https://doi.org/10.1016/j.trpro.2015.07.010. URL: https://www.sciencedirect.com/science/article/pii/S2352146515001702 (cit. on p. 15).

[33] R. Krüger, D. Thom, M. Wörner, H. Bosch, T. Ertl. “TrajectoryLenses – A Set-based Filtering and Exploration Technique for Long-term Trajectory Data”. In: *Computer Graphics Forum* 32.3pt4 (2013), pp. 451–460. DOI: https://doi.org/10.1111/cgf.12132. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.12132. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.12132 (cit. on p. 19).

[34] M. Lu, Z. Wang, X. Yuan. “Trajrank: Exploring travel behaviour on a route by trajectory ranking”. In: *2015 IEEE Pacific Visualization Symposium (PacificVis)*. IEEE. 2015, pp. 311–318 (cit. on p. 19).

[35] Mapbox. *Vector Tiles*. https://docs.mapbox.com/api/maps/vector-tiles/. Accessed: 2021-11-05. 2021 (cit. on p. 100).

[36] Mapzen, Contributors. *Tangram: WebGL Maps for Vector Data*. https://github.com/tangrams/tangram. Accessed: 2021-11-05. 2021 (cit. on p. 100).

[37] Mozilla, individual contributors. *Intensive JavaScript*. https://developer.mozilla.org/en-US/docs/Tools/Performance/Scenarios/Intensive_JavaScript. Accessed: 2021-11-04. 2021 (cit. on p. 99).

[38] F. F.-H. Nah. “A study on tolerable waiting time: how long are web users willing to wait?” In: *Behaviour & Information Technology* 23.3 (2004), pp. 153–163 (cit. on pp. 42, 47).

[39] OSM, individual contributors. *Rendering*. https://wiki.openstreetmap.org/wiki/Rendering. Accessed: 2021-11-05. 2021 (cit. on p. 100).

[40] OSM, individual contributors. *Slippy Map*. https://wiki.openstreetmap.org/wiki/Slippy_Map. Accessed: 2021-11-05. 2021 (cit. on pp. 99, 100).

[41] OSM, individual contributors. *Slippy map tilenames*. https://wiki.openstreetmap.org/wiki/Slippy_map_tilenames. Accessed: 2021-11-05. 2021 (cit. on pp. 27, 100).

[42] OSM, individual contributors. *Tiles*. https://wiki.openstreetmap.org/wiki/Tiles. Accessed: 2021-11-05. 2021 (cit. on p. 99).

[43] OSM, individual contributors. *Zoom levels*. https://wiki.openstreetmap.org/wiki/Zoom_levels. Accessed: 2021-11-10. 2021 (cit. on p. 27).

[44] N. Owens. "Verwaltung und Indizierung von Bewegungstrajektorien". B.S. thesis. 2018. DOI: 10.18419/opus-10202. URL: http://elib.uni-stuttgart.de/handle/11682/10219 (cit. on p. 18).

[45] Pistache. *Pistache - An elegant C++ REST framework*. http://pistache.io/. Accessed: 2021-10-26. 2021 (cit. on p. 18).

[46] S. M. Pizer, E. P. Amburn, J. D. Austin, R. Cromartie, A. Geselowitz, T. Greer, B. ter Haar Romeny, J. B. Zimmerman, K. Zuiderveld. "Adaptive histogram equalization and its variations". In: *Computer Vision, Graphics, and Image Processing* 39.3 (1987), pp. 355–368. ISSN: 0734-189X. DOI: https://doi.org/10.1016/S0734-189X(87)80186-X. URL: https://www.sciencedirect.com/science/article/pii/S0734189X8780186X (cit. on p. 116).

[47] P. Poorten, S. Zhou, C. Jones. "Topologically-Consistent Map Generalisation Procedures and Multi-scale Spatial Databases". In: Sept. 2002, pp. 209–227. DOI: 10.1007/3-540-45799-2_15 (cit. on p. 20).

[48] C. Proissl, T. Rupp. "On the Difference between Search Space Size and Query Complexity in Contraction Hierarchies". In: *SIAM Conference on Applied and Computational Discrete Algorithms (ACDA21)*. SIAM. 2021, pp. 77–87 (cit. on p. 18).

[49] T. Rupp, S. Funke. "A Lower Bound for the Query Phase of Contraction Hierarchies and Hub Labels and a Provably Optimal Instance-Based Schema". In: *Algorithms* 14.6 (2021), p. 164 (cit. on p. 18).

[50] P. Selvidge. "How long is too long to wait for a website to load". In: *Usability news* 1.2 (1999), pp. 1–3 (cit. on p. 47).

[51] M. Seybold. "Robust Map Matching for Heterogeneous Data via Dominance Decompositions". In: June 2017, pp. 813–821. ISBN: 978-1-61197-497-3. DOI: 10.1137/1.9781611974973.91 (cit. on p. 28).

[52] E. Spahiu, P. Lindemann, S. Kaiser, T. Rupp. "Bachelor-Forschungsprojekt: Webbasierte Visualisierung von Trajektorien". In: (Dec. 2019) (cit. on pp. 18, 30).

[53] StatCounter. *Desktop Screen Resolution Stats Europe*. https://gs.statcounter.com/screen-resolution-stats/desktop/europe. Accessed: 2021-11-17. 2021 (cit. on p. 106).

[54] Statista. *Forecast number of mobile devices worldwide from 2020 to 2025 (in billions)*. https://www.statista.com/statistics/263437/global-smartphone-sales-to-end-users-since-2007/. Accessed: 2021-11-28. 2021 (cit. on p. 15).

[55] Statista. *Number of smartphones sold to end users worldwide from 2007 to 2021*. https://www.statista.com/statistics/263437/global-smartphone-sales-to-end-users-since-2007/. Accessed: 2021-11-28. 2021 (cit. on p. 15).

[56] O. team. *Image Pyramids*. https://docs.opencv.org/3.4.15/dc/dff/tutorial_py_pyramids.html. Accessed: 2021-10-26. 2021 (cit. on p. 68).

[57] O. team. *OpenCV: Open Source Computer Vision Library*. https://github.com/opencv/opencv. Accessed: 2021-11-05. 2021 (cit. on p. 100).

[58] Z. Wang, M. Lu, X. Yuan, J. Zhang, H. Van De Wetering. "Visual traffic jam analysis based on trajectory data". In: *IEEE transactions on visualization and computer graphics* 19.12 (2013), pp. 2159–2168 (cit. on p. 19).

[59] Y. Yue, Y. Zhuang, Q. Li, Q. Mao. "Mining time-dependent attractive areas and movement patterns from taxi trajectory data". In: *2009 17th international conference on geoinformatics*. IEEE. 2009, pp. 1–6 (cit. on p. 19).

[60] Y. Zheng. "Trajectory data mining: an overview". In: *ACM Transactions on Intelligent Systems and Technology (TIST)* 6.3 (2015), pp. 1–41 (cit. on p. 19).

All links were last followed on November 28, 2021.

A Proofs

A.1 Balanced and Saturated Search Tree

The following theorem is common knowledge. Since its extensively used in the augmentation, however, it is included here for the sake of completeness.

Theorem 6 (Saturated balanced tree) *Given are $\hat{n} = 2^k - 1$ nodes. Then, the nodes can be arranged in a full balanced and saturated tree T_k with full last layer having 2^{k-1} nodes.* □

Proof The proof is given by induction over k. To begin with, the case $k = 1$ is trivial since there is only one vertex. Let now $k > 1$ and a saturated, balances tree T_{k-1} with full layers exists for $k - 1$. Since this tree is fully saturated with $k - 1$ levels, there are exactly $2^{(k-1)-1}$ leave nodes. The tree T_k emerges from T_{k-1} by appending two new children to each leaf. Clearly, this tree is still saturated. It is also balances because each branch grew the same size. T_k now has $(2^{k-1} - 1) + 2 \cdot (2^{k-2}) = 2 \cdot 2^{k-1} - 1 = 2^k - 1$ nodes. ■

A.2 No Gaps in Left-Aligned Tree Ordering

Definition 11 (Gap) A tree has a gap on layer l if and only if layer l has less than 2^l elements and there is at least an empty node between two nodes. □

Theorem 7 (No gap in layer) *Let T be a binary search tree obtained by using left-aligned level-order sorting on the input $V = [1, \ldots, n]$. Then, there's no layer in T having a gap.* □

Proof The case where $n = 2^l - 1$ for some $l \in N$ is trivial. Let $n = 2^l - 1 + t$ with $1 \leq t < 2^{l-1}$. Appending virtual nodes $V' = [n+1, n+2, \ldots, n+a]$ with $a = 2^{l-1} - t$ yields the new input. Since virtual nodes will be skipped later, we call them empty. Now create and traverse T from top to bottom, left to right (i.e. level-order). Let's assume $b \in V$ to be the first node creating a gap by having at least one level-order predecessor $a \in V'$. Because of the search tree property, $a < b$ holds. This, however, contradicts the property $b \leq n < n + 1 \leq a$. ■

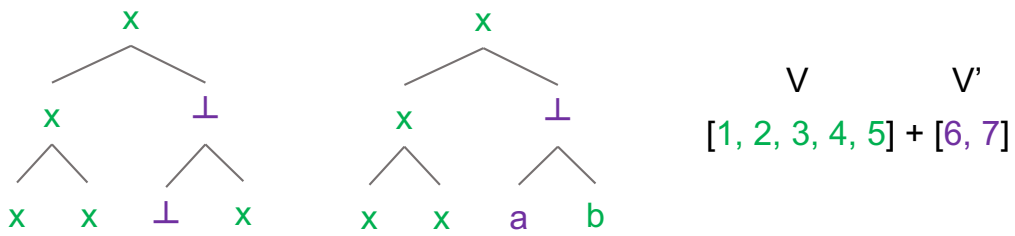


Figure A.1: Proof sketch: $a \in V$, $b \in V'$. Left part shows T. The tree right to it has the first gap-producing element marked as b.

B Used Datasets

The listing below shortly describes the used datasets and provides references where to find the data. This allows for transparent reproduction of the discussed findings.

Cuba Trajectories

The *Cuba-Dataset*[1] consists of a path file containing 302 trajectories, combined with a CH of the open street maps graph of Cuba. The shortest paths uses one edge, the longest 283 compressed edges (3 847 uncompressed). On average, there are 34.69 edges per path in the compressed setting and 358.87 edges in the uncompressed case.

Size: 21.3MB paths, 216.0 MB graph.

Saarland Trajectories

The *Saarland-Dataset*[2] is a small CH-graph with a dense paths-overlay. It contains 2 572 trajectories with an average number of 42.74 compressed edges.

Size: 101.8MB paths, 120.0MB graph.

Since the some paths have invalid or missing time stamps, there is also a filtered dataset, called *Filtered Saarland-Dataset*, containing 472 paths.

Based on the original Saarland graph, various random path files have been generated. Due to theirs considerable sizes, they are not included in the repository, but can be generated by the PF. Most importantly, the following artificial path files are listed, which were part of evaluations:

Saarland100k

The *Saarland100K-Dataset* contains 100 000 generated shortest paths between two randomly sampled points within a 500km distance each.

Size: 13.6MB paths

[1] Available at https://bitbucket.org/baurls/ma_baur_pathfinder_visualization/src/main/pathfinder_server/data/cuba/

[2] Available at https://bitbucket.org/baurls/ma_baur_pathfinder_visualization/src/main/pathfinder_server/data/saarland/

Saarland1M

The *Saarland1M-Dataset* contains one million generated shortest paths between two randomly sampled points within a 50km distance each.

Size: 134.3MB paths

Saarland10M

The *Saarland10M-Dataset* consists of ten million generated shortest paths analogously to the 1M dateset. The paths have an average number of 15.29 compressed edges.

Size: 1.3GB paths

Germany Trajectories

In this work, the *Germany-Dataset*[3] is the largest real world dataset with 5 748 344 nodes and 24 849 999 edge. It is a fine-grained version of OSM routes in Germany containing 372 534 trajectories with an average number of 35.33 compressed edges.

Size: 111.33MB paths, 8.41GB graph.

Europe Trajectories

The Europe-graph contains the main high speed routes in Europe includes 134 405 349 edges and 38 064 931 nodes with a maximum CH-level of 391.

Size: 6.56GB graph.

Since there are no real paths available, the following generated path set is in use:

Europe500k

For the *Europe500K Dataset*, 500 000 trajectories were generated, each connecting two randomly chosen points within a 500km radius using a shortest path.

Size: 70.78MB paths

[3] Available at https://fmi.uni-stuttgart.de/files/alg/data/SPP/trajectory_data.tar.gz

C Additional Plots and Measurements

The most important plots were embedded in the main parts of the thesis, but for the sake of completeness the remaining referenced plots are appended in the following.

Batching

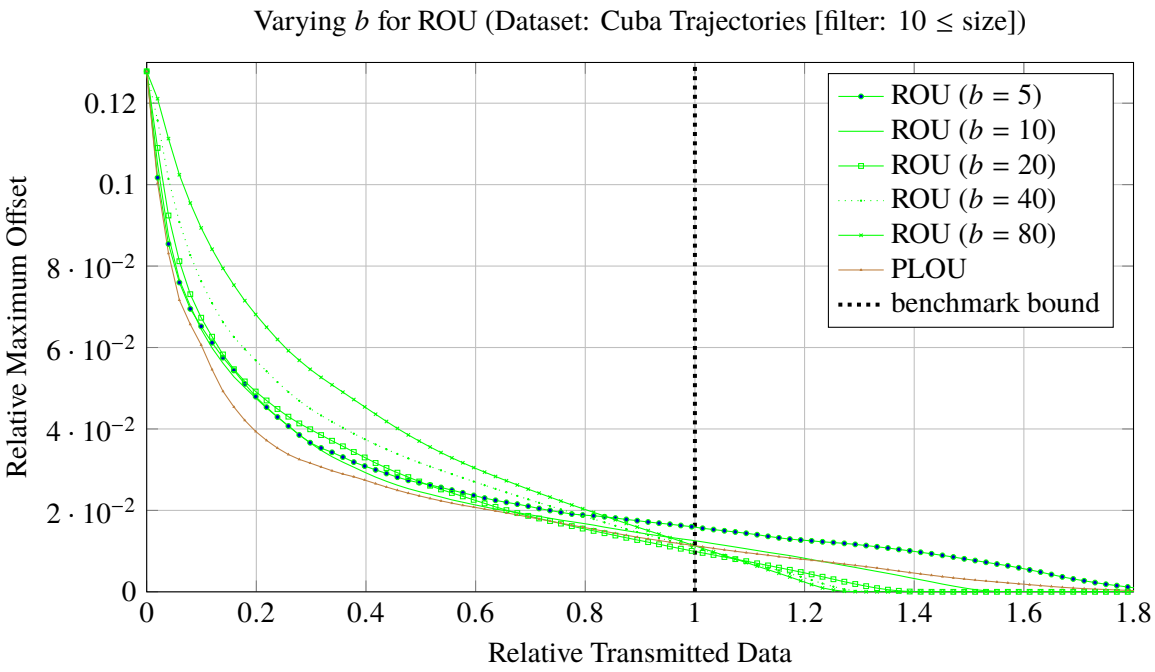


Figure C.1: Plot showing the average errors over the (normalized) number of transmitted data batches for different values of b and PLOU for reference. The colors were chosen the same as in the original version.

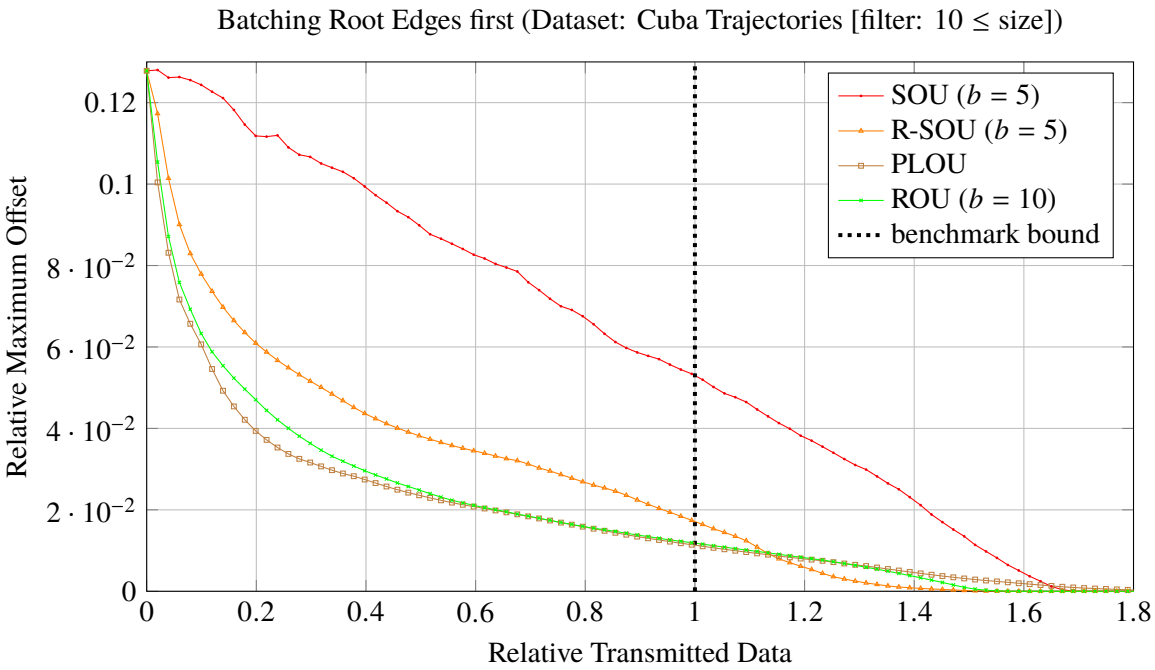
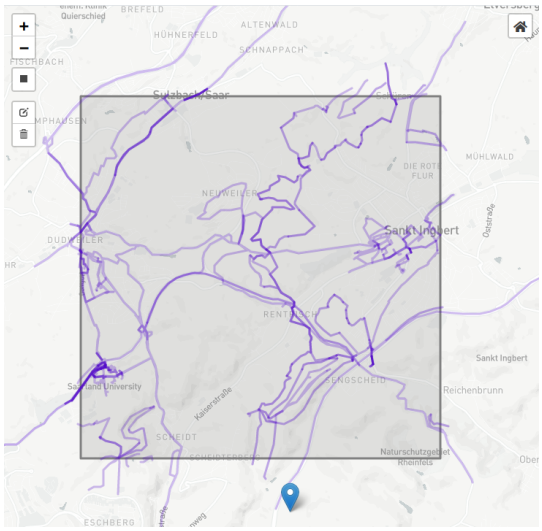
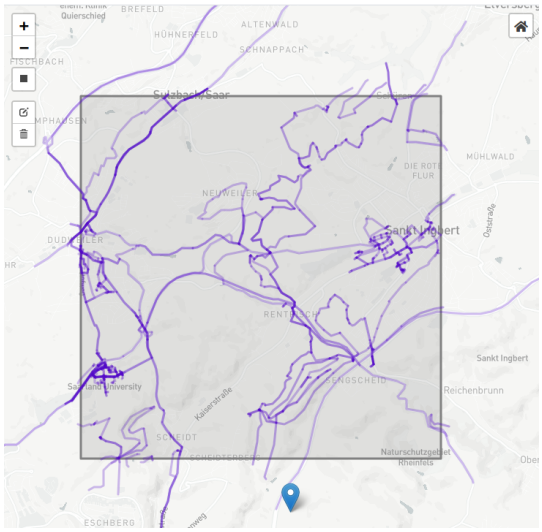


Figure C.2: Plot showing the performance for R-SOU transmission compared to benchmark methods.

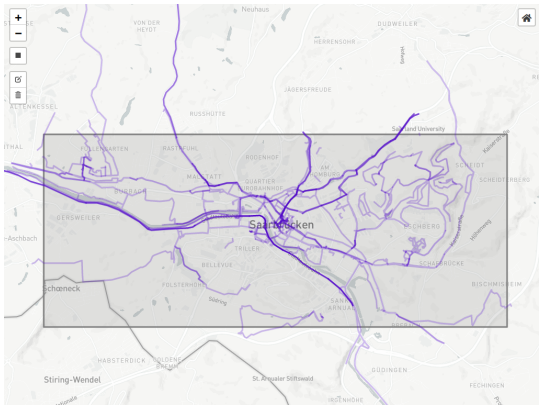
C.1 Binning comparisons



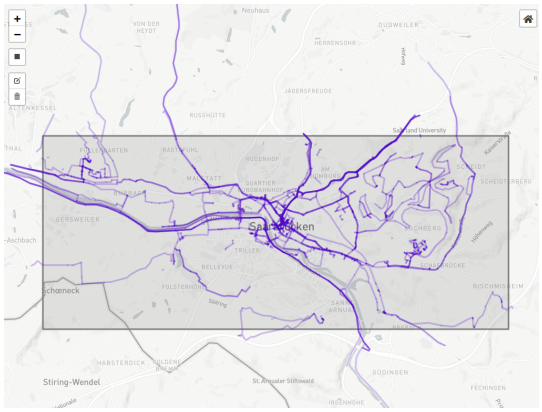
(a) Dateset 1: 25 bins



(b) Dateset 1: no binning

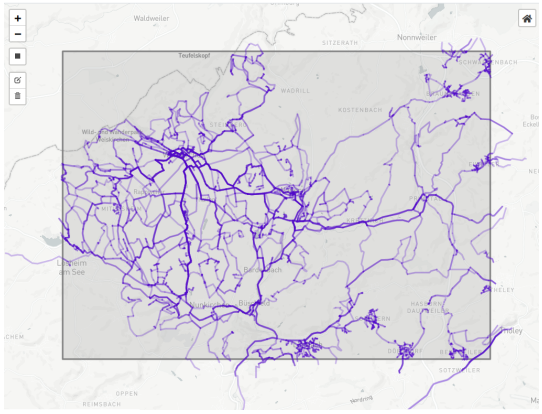


(c) Dateset 2: 25 bins



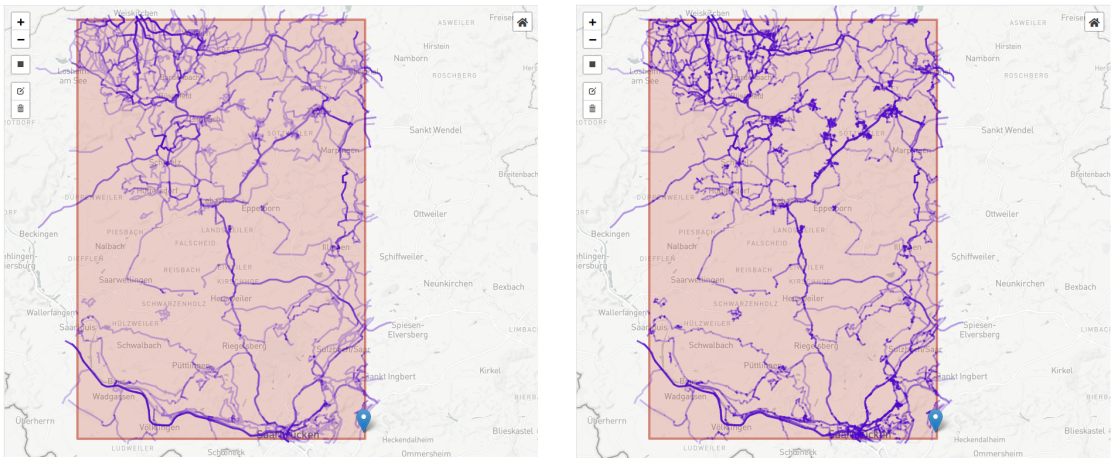
(d) Dateset 2: no binning

(e) Dateset 3: 25 bins



(f) Dateset 3: no binning

Figure C.3: Comparison of the original datasets plots (right) and their equivalent representation which is limited to 25 color scales.



(a) Dateset 4: 25 bins

(b) Dateset 4: no binning

Figure C.4: Comparison of the original datasets plots (right) and their equivalent representation which is limited to 25 color scales (continued).

C.2 Additional Measurements

Binning measurements

Binning Types	Plotting	Exporting	Total
1	0.0183	0.0113	0.0295
2	0.0204	0.0122	0.0326
3	0.0254	0.0111	0.0365
4	0.0174	0.0115	0.0289
5	0.0234	0.0117	0.0351
10	0.0204	0.0118	0.0321
25	0.022	0.0119	0.0338
-	0.1293	0.0119	0.1412

(a) Dateset 1, processing times (in seconds)

Binning Types	Plotting	Exporting	Total
1	0.0283	0.0221	0.0503
2	0.0271	0.0237	0.0508
3	0.0256	0.022	0.0476
4	0.0259	0.0219	0.0478
5	0.0245	0.0223	0.0468
10	0.0243	0.0222	0.0465
25	0.0305	0.0241	0.0546
-	0.1951	0.026	0.2211

(a) Dateset 2, processing times (in seconds)

Binning Types	Plotting	Exporting	Total
1	0.0518	0.0492	0.101
2	0.0524	0.0493	0.1017
3	0.0518	0.049	0.1008
4	0.0497	0.0488	0.0984
5	0.0571	0.0491	0.1061
10	0.0584	0.0497	0.1081
25	0.057	0.0488	0.1058
-	0.3376	0.0521	0.3897

(a) Dateset 3, processing times (in seconds)

Binning Types	Plotting	Exporting	Total
1	0.126	0.1053	0.2313
2	0.1214	0.1043	0.2257
3	0.1186	0.1058	0.2243
4	0.1219	0.1043	0.2262
5	0.1189	0.1047	0.2236
10	0.1218	0.1058	0.2276
25	0.1314	0.1067	0.2381
-	0.6642	0.1112	0.7754

(a) Dateset 4, processing times (in seconds)

Binning Types	Plotting	Exporting	Total
1	0.1993	0.1851	0.3845
2	0.1889	0.1867	0.3755
3	0.1971	0.1841	0.3812
4	0.2023	0.1841	0.3864
5	0.1834	0.1838	0.3672
10	0.204	0.1837	0.3877
25	0.1892	0.1844	0.3737
-	1.2159	0.1996	1.4155

(a) Dateset 5, processing times (in seconds)

EBPF Queries

	Zoom								
	2	4	6	8	10	12	14	16	18
Factor $1/32$									
Frontend [ms]	**8**	**6**	**16**	**97**	**9**	**14**	**24**	**22**	**24**
Backend [s]	**14.0**	**13.9**	**13.9**	**14.0**	**13.9**	**14.0**	**14.0**	**14.0**	**14.0**
Size [MB]	**0.07**	**0.13**	**0.25**	**0.87**	**0.15**	**0.26**	**0.75**	**0.55**	**0.71**
Factor $1/16$									
Frontend [ms]	**90**	**215**	**34**	**123**	**223**	**82**	**31**	**84**	**129**
Backend [s]	**14.2**	**14.2**	**14.0**	**14.0**	**14.2**	**14.0**	**13.9**	**14.2**	**14.2**
Size [MB]	**1.87**	**3.75**	**0.69**	**1.29**	**4.53**	**1.67**	**1.02**	**2.58**	**3.81**
Factor $1/8$									
Frontend [ms]	**265**	**64**	**248**	**441**	**653**	**734**	**538**	**325**	**388**
Backend [s]	**14.5**	**14.1**	**14.5**	**14.5**	**14.4**	**14.5**	**15.3**	**14.7**	**14.3**
Size [MB]	**6.43**	**1.85**	**6.41**	**7.40**	**6.17**	**8.31**	**18.59**	**9.85**	**4.62**
Factor $1/4$									
Frontend [ms]	**865**	**1033**	**1278**	**2509**	**469**	**1952**	**1610**	**2324**	**2445**
Backend [s]	**15.4**	**16.1**	**16.5**	**16.9**	**15.0**	**16.2**	**16.0**	**17.6**	**18.1**
Size [MB]	**16.17**	**23.76**	**27.48**	**33.21**	**12.98**	**27.19**	**24.74**	**42.43**	**51.57**
Factor $1/2$									
Frontend [ms]	**2027**	**2425**	**1320**	**4571**	**6272**	**6578**	**7148**	**4094**	**4151**
Backend [s]	**18.3**	**18.6**	**16.4**	**19.4**	**21.7**	**22.1**	**24.7**	**22.0**	**20.9**
Size [MB]	**45.38**	**49.28**	**27.23**	**59.93**	**89.17**	**95.72**	**128.09**	**95.50**	**89.21**

Table C.6: Transmission sizes and calculation times using EBPF with binning activated ($b = 25$) on the Germany Dataset. Bold values indicate early stopping due to exceeding the calculation time limit.

	Zoom								
	2	4	6	8	10	12	14	16	18
Factor $1/32$									
Frontend [ms]	12	11	10	9	12	10	11	17	14
Backend [ms]	117	118	118	117	122	119	122	129	127
Size [MB]	0.24	0.27	0.25	0.25	0.33	0.28	0.33	0.43	0.41
Factor $1/16$									
Frontend [ms]	51	44	42	42	50	52	48	43	49
Backend [ms]	163	145	145	139	155	163	157	146	156
Size [MB]	0.90	0.63	0.64	0.57	0.81	0.92	0.84	0.68	0.85
Factor $1/8$									
Frontend [ms]	105	100	100	148	127	137	131	146	142
Backend [ms]	240	224	244	274	241	302	266	300	281
Size [MB]	2.02	1.77	2.08	2.48	2.03	2.88	2.39	2.87	2.65
Factor $1/4$									
Frontend [ms]	398	437	365	404	488	363	416	417	549
Backend [ms]	565	630	584	555	647	596	616	590	741
Size [MB]	6.68	7.64	6.91	6.46	7.80	7.08	7.46	7.02	9.24
Factor $1/2$									
Frontend [ms]	2041	1907	1463	2069	1871	1287	1702	2025	1659
Backend [ms]	1695	1784	1677	1724	1592	1614	1706	1779	1648
Size [MB]	22.27	23.41	22.02	22.46	20.66	20.99	22.27	23.37	21.67

Table C.7: Transmission sizes and calculation times using EBPF with binning activated ($b = 25$) on the Saarland10M Dataset.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature