How to Eat a Graph: Computing Selection Sequences for the Continuous Generalization of Road Networks

Markus Chimani
University of Osnabrück
markus.chimani@uni-osnabrueck.de

Thomas C. van Dijk
University of Würzburg
thomas.van.dijk@uni-wuerzburg.de

Jan-Henrik Haunert
University of Osnabrück
janhhaunert@uni-osnabrueck.de

ABSTRACT

In a connected weighted graph, consider deleting the edges one at a time, in some order, such that after every deletion the remaining edges are still connected. We study the problem of finding such a deletion sequence that maximizes the sum of the weights of the edges in all the distinct graphs generated: the weight of an edge is counted in every graph that it is in. This effectively asks for the high-weight edges to remain in the graph as long as possible, subject to connectivity. We apply this to road network generalization in order to generate a sequence of successively more generalized maps of a road network so that these maps go well together, instead of considering each level of generalization independently. In particular, we look at the problem of making a road segment selection that is consistent across zoom levels.

We show that the problem is $\mathcal{NP}$-hard and give an integer linear program (ILP) that solves it optimally. Solving this ILP is only feasible for small instances. Next we develop constant-factor approximation algorithms and heuristics. We experimentally demonstrate that these heuristics perform well on real-world instances.

Categories and Subject Descriptors

F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Sequencing and scheduling*; H.2.8 [**Database Management**]: Database Applications—*Spatial databases and GIS*

Keywords

map generalization, optimization, scheduling

1. INTRODUCTION

Map generalization is the process of deriving a map of a smaller scale from a given map. During this process, some map objects are eliminated; others are aggregated or simplified. A high degree of automation has been achieved for the generalization of static topographic maps, which national mapping agencies typically produce at a small number of different scales [5]. For interactive maps that allow for continuous zooming, however, many problems of automatic generalization are still open. Generally, the aim is to avoid sudden changes caused by generalization during zooming [22].

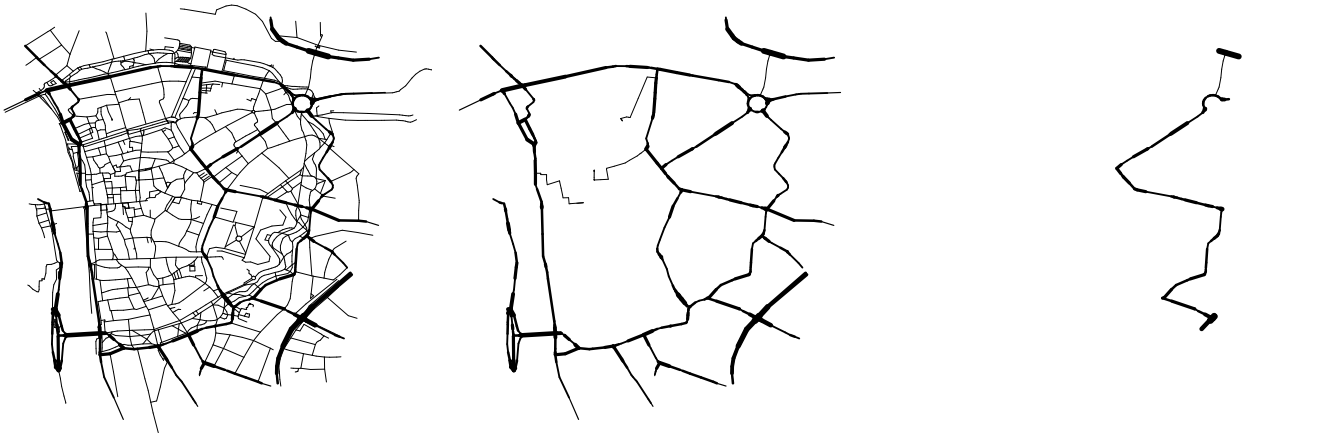
In this paper, we address interactive maps of road networks in which the road segments successively disappear when a user zooms out. To this end, we represent a road network as a graph whose edges (that is, the straight-line segments of the polylines representing the roads) are weighted by an importance measure, which can reflect road category or relevance for navigation. We present algorithms that, given a weighted graph (a road network at the largest scale) decide in which order the edges of the graph are removed from the map when scale decreases. The results of our algorithms could be used to set up data structures that provide users with maps of arbitrary scales. That is, similar to Van Oosterom and Meijers [23], who deal with the generalization of polygonal maps, we suggest computing the sequence of edge removals once and reusing it each time a user queries a map. We assume that, once we have computed the sequence of edge removals, we can map every edge removal to a scale by following existing guidelines about the relationship between the scale and the number of map objects selected for that scale [19]. Furthermore, to achieve the impression of a continuous transformation process, we suggest combining our selection method with existing algorithms for line simplification and morphing [4, 15].

An important constraint when generalizing geographic networks is that a connected network must be kept connected [14]. Therefore, simply removing the edges in increasing order of importance is not an acceptable solution to our problem. Nevertheless, we generally should try to avoid the removal of important edges for as long as possible. We take this concern into account by measuring the quality of the map (at any fixed scale) with the total weight of all displayed edges. Then, we ask for a sequence of edge removals that maximizes the sum of weights over all distinct maps that are shown between the original map and the empty map (which finally will be shown at a very small scale), while requiring that at each scale the displayed graph is connected. See Figs. 1 and 10 for some example snapshots computed using our algorithms.

Related Work.

Map generalization can be modeled with a set of constraints, each of which is concerned either with the legibility of the output map or with the preservation of character-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
SIGSPATIAL'14, November 04 - 07 2014, Dallas/Fort Worth, TX, USA
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
978-1-4503-3131-9/14/11 ...$15.00
http://dx.doi.org/10.1145/2666310.2666414.



(a) Input road network (G_0). (b) After 2480 edge deletions (G_{2480}). (c) After 2880 edge deletions (G_{2880}).

Figure 1: Three steps of GreedySelection's solution on a map of Würzburg, where the edges have been weighted using PageRank and road class as described in Sec. 8; edge weight is indicated using line width. Notice that some low-weight edges remain in the north-east almost until the end since they form an important connection.

istic features of the input map [3]. Usually, it is impossible to fully satisfy all constraints, thus some constraints are relaxed. Optimization is often chosen to satisfy those constraints best [9, 17, 24]. For maps supporting continuous zooming, however, most of the existing algorithms iteratively perform small steps of generalization without optimizing a global objective [23]. An edge-based selection method for continuously zoomable network maps has been presented by Aliakbarian [1]. We discuss this method in more detail in Section 3 and show that, when measuring quality according to our objective function, the method is arbitrarily bad.

Most similar to our general approach by optimization is the approach of Schwartges et al. [16], which is to compute *active ranges* for points of interest in a map that admits continuous zooming. In this context, an active range is an interval of scales in which a point is visible. By allowing only one active range for each point it is avoided that points flicker on and off during zooming. Schwartges et al. optimize the total length of all active ranges while requiring that the map never contains two points whose distance is smaller than a user-set value. This is similar to our approach in the sense that the objective function aggregates a criterion of map quality (that is, preservation of information) over all scales. Originally, the concept of active ranges has been introduced for the labeling of zoomable maps [2]. Similar approaches have been used for labeling maps that can be rotated [8] and maps that follow the trajectory of a user [7].

The generalization of geographic networks with a single target scale has been examined extensively; multiple researchers have pointed out the importance of network connectivity [1,3,12,14,27]. Given a connected graph that represents a geographic network and whose edges are weighted by travel time, Mackaness and Beard [14] have suggested generalizing the network by computing a minimum spanning tree T of G (which is the smallest subgraph of G that keeps all vertices of G connected) and adding edges from G to T until an acceptable degree of completeness is reached. We argue, however, that for a generalized road network we need not select all vertices of the input graph. Therefore, for every scale, we do *not* require a connected *spanning* subgraph of G to be selected but a connected subset of the edges of G.

Weighting the edges of a road network by importance is often considered a crucial step preceding the selection. For this purpose, graph-theoretical measures such as betweenness centrality [12] and PageRank [21] have been used. We make no assumption about how the edges of the graph are weighted but consider betweenness centrality and PageRank as good choices if the map is intended for navigation.

Instead of selecting on the basis of edges, some generalization methods for road networks select on the basis of strokes [18, 26, 27], which are paths in which every two consecutive edges have similar orientations. The reason is that humans perceive a stroke as one unit which should not be broken apart [18]. Our edge-based approach, on the other hand, has the advantage that the generalization from the largest to the smallest scale is performed in very small steps, which can support the impression of a continuous transformation process. Therefore, we think that the knowledge of strokes would be best integrated in our method by requiring connectivity for each stroke, that is, allowing edges to be removed only from the two ends of a stroke. This connectivity constraint has been used before [20] but not for continuously zoomable maps. However, since the problem of removing the edges in an optimal order is new, we focus on the more common definition of connectivity.

One of our algorithms is based on results from scheduling theory. We observe that our objective function has been studied under the names *total weighted completion time* and *linear delay penalties*. Subject to arbitrary precedence constraints between jobs, minimizing the total weighted completion time is $\mathcal{NP}$-hard [13], but there exist polynomial-time algorithms for several restricted classes of precedence constraints, for example for rooted forests [11]. Our connectivity constraint has not been studied in this setting.

Our Contribution.

When a sequence of maps is shown during a zoom operation, what criteria are essential for the quality of that sequence? According to those criteria, how can we efficiently compute a map sequence that is (close to) optimal? Those

two questions are almost completely unexplored in the map generalization literature. Computing a well-generalized map by optimization is rather common, of course, but a sequence of well-generalized maps is not necessarily the same as a good *sequence* of maps. New approaches are needed that ensure consistency and aggregate quality across zoom levels, while satisfying constraints at every scale. We consider our work a small but crucial step in that direction.

In particular, we present a new approach to the problem of finding a good sequence of edge removals in order to define how a map of a road network becomes successively more generalized when the map's scale decreases. We optimize a global objective function that reflects the completeness of the map, aggregated over all scales, and ensures that, at every single scale, the displayed road network is connected.

We define the problem of finding an optimal removal sequence formally (Sect. 2), give an efficient greedy heuristic (Sect. 3), and prove that the problem is $\mathcal{NP}$-hard (Sect. 4). Then, we present an exact method based on integer programming (Sect. 5) and two different approaches that both have an approximation guarantee of 1/3: one based on balanced tree partitions (Sect. 6) and one that uses an existing scheduling algorithm (Sect. 7). Finally, we present some experimental results (Sect. 8) and our conclusion (Sect. 9).

Though we focus on the application to map generalization here, our algorithm TreeSchedule (Sec. 7) can also be seen in a more general scheduling context.

2. PROBLEM DEFINITION

We define the problem of computing an optimal sequence of edge removals as follows.

PROBLEM 1 (EDGESCHEDULING). *Given a connected graph $G = (V, E)$ with m edges and edge weights $w\colon E \to \mathbb{R}^+_{\geq 0}$, find a permutation $S = \langle e_1, \ldots, e_m \rangle$ of E such that*

- *for $i = 0, \ldots, m-1$, the subgraph $G_i \subseteq G$ induced by the edges with index larger than i is connected, and*
- *$Q(S) := \sum_{i=1}^{m-1} W(G_i)$ is maximized, where $W(G_i) := \sum_{j=i+1}^{m} w(e_j)$ is the total weight of G_i.*

In other words, G_i needs to be connected and arises from G by deleting the edges $e_1, \ldots, e_i$ together with all resulting degree-0 vertices. Hence, from now on we use the term *"deleting an edge e"* to mean removing e and updating the vertex set V such that it only keeps the vertices with at least one incident edge.

Note that $G_0 = G$ and that G_{m-1} contains only one edge, thus for $i = 0$ and $i = m-1$ the graph G_i is trivially connected. When measuring the quality $Q(S)$ of a solution S, we do not include the weight of G_0 since $W(G_0)$ is identical for all possible solutions. Obviously, excluding $W(G_0)$ from the objective function does not have an influence on whether or not a solution is optimal. It can have an impact on approximation factors, however.

3. A GREEDY HEURISTIC

To better understand the problem, we introduce a simple greedy algorithm (Algorithm 1), which we will refer to as *the greedy algorithm*. It iteratively deletes a minimum-weight admissible edge from the current graph $G = (V, E)$, where an edge e is *admissible* if G remains connected after deleting e or if e is the only edge in G.

Algorithm 1: GreedySelection

```
1 Let S be a new array of m = |E| edges.
2 for i = 1 to m do
3 |   S[i] = a minimum-weight admissible edge in G
4 |   Delete edge S[i]
5 return S
```

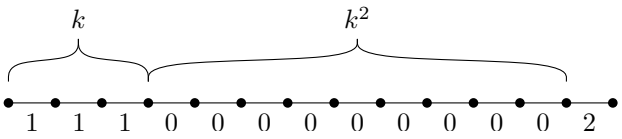


Figure 2: A bad case for the greedy algorithm.

The greedy algorithm may seem reasonable as it attempts to postpone the deletion of heavy edges: they will then contribute to the objective value for longer. It is easy to see, however, that the algorithm's result can be arbitrarily bad compared to an optimal solution. Consider the instance in Fig. 2, where the input graph G is a path of $k + k^2 + 1$ edges. In this path, the first k edges have weight one, the next k^2 edges have weight zero, and the last edge has weight two.

The greedy algorithm first deletes all edges of weight one, then the edges of weight zero, and finally the edge of weight two. The solution S that we obtain has quality

$$Q(S) = [(k-1) + \ldots + 1] + (k + k^2) \cdot 2 = \frac{5}{2}k^2 + \frac{3}{2}k\,.$$

Now, consider the solution S^* obtained by deleting the edges in the reverse order, that is, by always picking the rightmost edge in Fig. 2. This solution has quality

$$Q(S^*) = (k^2+1) \cdot k + [(k-1) + \ldots + 1] = k^3 + \frac{1}{2}k^2 + \frac{1}{2}k$$

which is strictly better than S if $k > 2$.

As $Q(S^*)$ is a polynomial of higher degree than $Q(S)$, the ratio $Q(S^*)/Q(S)$ exceeds any constant-factor bound if we choose k large enough. Therefore, the solution S of the greedy algorithm can be arbitrarily bad compared to S^*.

This worst-case quality notwithstanding, the greedy algorithm is of practical interest. We have seen experimentally that it tends to give good results on realistic instances (Sec. 8). Furthermore, with proper data structures it can be implemented to run efficiently.

THEOREM 1. *Algorithm GreedySelection computes a feasible solution. Its objective value may be arbitrarily far from the optimum. The algorithm can be implemented to run in $\mathcal{O}(\, m \log^2 m\,)$ time.*

PROOF. The correctness of the algorithm is evident, and we discussed the non-approximation above. It remains to show that the claimed runtime can be achieved. Line 3 of the pseudocode is crucial; we handle it in $\mathcal{O}(\log^2 m)$ time. Then the claimed bound follows.

Consider the remaining graph during an iteration of the loop. An edge $e = uv$ is admissible if, after deleting e, the graph still has a path between u and v. The edge is also admissible if at least one of u and v has degree 1: then e is a dangling edge and can be deleted without harming connectivity. The latter can be tested in constant time. For the former we use the dynamic connectivity data structure

of Holm et al. [10]. It handles edge deletions, insertions, and connectivity queries between pairs of vertices in amortized $\mathcal{O}(\log^2 m)$ time each, so we can test the admissibility of an arbitrary edge in that time.

After sorting the edges by increasing weight, which takes $\mathcal{O}(m \log m)$ time, we test each edge in this order. If the edge being considered is admissible, we pick it next in the schedule and delete it from the graph. If the edge is not admissible, then we are not allowed to delete it now. Instead, we mark it as being *on hold* and continue with the next edge.

Whenever we do delete an edge e, we check whether either of its endpoints has its degree reduced to one, where the neighboring edge e' is on hold. If this happens, e' was not admissible before the deletion of e, but is admissible now. This cannot happen on both side of e, so e' is well-defined. Then e' is the minimum-weight admissible edge in the new graph: it is the only newly-admissible edge and its weight must be minimum since we tried to delete it before. This process can cascade along a path, but cannot branch.

The runtime follows from the fact that each edge is considered for deletion at most twice—once for "normal" deletion, and possibly once more when it becomes a dangling edge on hold. In either case, the deletion requires amortized $\mathcal{O}(\log^2 m)$ time. □

See Fig. 1 for a solution computed using this algorithm; the data set is described in Sec. 8. Note in Fig. 1(b) that the edge deletions mostly follow the road classes, but that in the north-east a few low-weight edges remain. This is due to the connectivity constraint: these edges form a bridge with a high amount of weight on both sides. They remain even in Fig. 1(c).

We note that Aliakbarian [1] proposes an iterative algorithm that is somewhat similar to GreedySelection. This algorithm first stores the edges in a queue, sorted by weight. In every iteration, the algorithm takes an edge from the queue, starting with the least important edge. If the selected edge is admissible, the algorithm deletes it from the graph. Otherwise, the algorithm reinserts the edge at the back of the queue. In Fig. 2, for example, it would attempt to delete the zero-weight edges first, but without success. Depending on how the algorithm resolves ties, it may then delete all unit-weight edges, then the edge of weight two, and finally the zero-weight edges. This means that on this instance, the algorithm performs even worse than GreedySelection, as it keeps the zero-weight edges until the very end. We also note that a straightforward implementation of Aliakbarian's method would take $\Omega(m^3)$ time in the worst case. Using the dynamic-connectivity data structure, this can be improved to $\mathcal{O}(m^2 \log^2 m)$, but it can occur that the algorithm actually performs quadratically-many iterations.

4. NP-HARDNESS

We now analyze the computational complexity of EdgeScheduling. We show that it is $\mathcal{NP}$-hard.

Theorem 2. *Given an instance $\mathcal{I} = (G, w)$ to the* EdgeScheduling *problem and an integer bound $Q_{\min}$, the decision problem of whether $\mathcal{I}$ has a solution S with $Q(S) \geq Q_{\min}$ is $\mathcal{NP}$-complete, even if G is planar.*

Our proof is by reduction from SteinerTreeInPlanarGraphs, which is known to be $\mathcal{NP}$-complete [6]. We abbreviate it as P-SteinerTree.

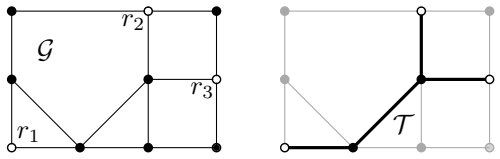


Figure 3: A graph G with three terminals r_1, r_2, r_3 (left) and a Steiner tree $\mathcal{T}$ spanning them (right).

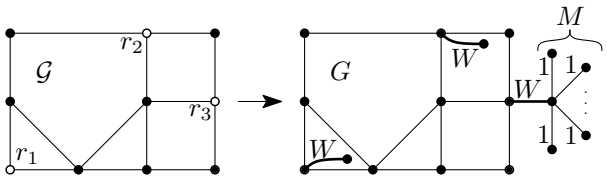


Figure 4: A P-SteinerTree instance (left) and the corresponding EdgeScheduling instance (right). The edges present in $\mathcal{G}$ receive weight zero.

Problem 2 (P-SteinerTree). *Given a planar graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, a set $\mathcal{R} \subseteq \mathcal{V}$ of terminals, and a positive integer bound β, is there a subtree $\mathcal{T}$ of $\mathcal{G}$ that spans $\mathcal{R}$ and has at most β edges?*

Before proving Theorem 2, we give a coarse idea of the relation between the Steiner tree problem and the generalization of road networks. Suppose the planar graph $\mathcal{G}$ in Fig. 3 (left) represents part of a road network and the vertices r_1, r_2, r_3 are of ultimate importance. For example, they might be the only access to important roads that are not part of $\mathcal{G}$. Suppose now that for a map of a certain scale we have an edge budget: we are allowed to select a certain number of edges from $\mathcal{G}$. Deciding if we can select a connected subgraph of $\mathcal{G}$ that spans all terminals is essentially P-SteinerTree. The following proof uses a similar idea, but we have to consider that EdgeScheduling asks for a sequence of edge deletions rather than for a single subgraph of the input.

Proof of Theorem 2 (sketch). The construction is as follows; exact calculations have been omitted due to space restrictions. Clearly EdgeScheduling is in $\mathcal{NP}$. We prove $\mathcal{NP}$-hardness by reduction from P-SteinerTree: for a given instance $(\mathcal{G} = (\mathcal{V}, \mathcal{E}), \mathcal{R}, \beta)$, we construct the instance (G, w) of EdgeScheduling as shown in Fig. 4. The graph G is obtained from $\mathcal{G}$ by adding, for each terminal $r \in \mathcal{R}$, a new vertex v_r and a new edge $e_r = rv_r$ of weight W. Call these the *terminal* edges. Let u be an arbitrary vertex among the new vertices. We create M vertices and connect each of them to u with a unit-weight edge: call these *anchor* edges. The *original* edges of $\mathcal{G}$ receive weight zero.

Suppose that $\mathcal{G}$ contains a tree $\mathcal{T}$ of β edges spanning $\mathcal{R}$. Then, in the EdgeScheduling instance, the following is allowed: first delete all original edges that do not belong to $\mathcal{T}$, then all anchor edges, then all terminal edges, and, finally, all edges of $\mathcal{T}$. In this way, the existence of a Steiner tree with β edges gives a lowerbound on the optimal quality achievable. In the opposite direction, picking W sufficiently large guarantees that any optimal solution of G has a similar structure: some original edges, then the anchor edges, and then a combination of terminal edges and original edges spanning $\mathcal{R}$. Call this second group of original edges the Steiner edges. Picking M sufficiently large guarantees

$$\textbf{Maximize} \sum_{e\in E}\sum_{d=1}^{m}(d-1)\cdot w(e)\cdot x_{ed} \quad (1)$$

$$\sum_{d=1}^{m} x_{ed} = 1 \qquad \forall e \in E \quad (2)$$

$$\sum_{e\in E} x_{ed} = 1 \qquad \forall 1 \le d \le m \quad (3)$$

$$\sum_{e'\in N(e)}\sum_{d'=d+1}^{m} x_{e'd'} \ge x_{ed} \qquad \forall e \in E, 1 \le d < m \quad (4)$$

$$x_{ed} \text{ binary} \qquad \forall e \in E,\ 1 \le d \le m \quad (5)$$

Figure 5: An integer linear program for EdgeScheduling**.**

that any optimal solution with at most β Steiner edges has higher quality than any optimal solution with more than β Steiner edges. This gives a value $Q_{\min}$ that allows us to decide whether $\mathcal{G}$ has a Steiner tree of a certain size. Picking $W := m + |\mathcal{R}| + 1$ and $M := (|\mathcal{R}| + m) \cdot |\mathcal{R}| \cdot W + 1$ suffices. Note that the constructed instance has polynomial size. □

5. INTEGER LINEAR PROGRAMMING

Since the EdgeScheduling problem is $\mathcal{NP}$-hard, it is natural to consider integer linear programming. This will allow us to compute optimal solutions, though this will only be feasible for small instances. This serves a double purpose. First of all, it allows us to verify that our problem statement makes sense: is an optimal solution to our optimization problem actually a good sequence of maps? Secondly, having optimal solutions to (small) instances enables us to experimentally investigate the quality of our heuristics and approximation algorithms.

A solution to the EdgeScheduling problem is a permutation of the graph's edges: we seek an indexing $\langle e_1, ..., e_m\rangle$ of the edges as described in the problem definition (Sec. 2). We call these indices *"deletion indices."*

The integer linear program is given in Fig. 5. We have a binary decision variable x_{ed} for every edge $e \in E$ and every possible deletion index $1 \le d \le m$: if $x_{ed} = 1$, then e is the d-th edge to be deleted. The objective function can be cleanly expressed using these variables (Eq. 1), where we note that the weight of G_0 is correctly ignored. We require that every edge is deleted exactly once (Eq. 2) and that every deletion index occurs exactly once (Eq. 3). Finally, we ensure connectivity at every arising subgraph in the deletion sequence using the following constraint: for every edge e (except the one that is deleted last) it must hold that one of its adjacent edges is deleted later—otherwise edge e, directly before its deletion, would be disconnected from the final edge. This constraint is implemented in Ineq. (4), where we use $N(e)$ to denote the set of edges adjacent to e.

6. CONSTANT-FACTOR APPROXIMATION

We will now discuss a polynomial-time algorithm that, in contrast to the aforementioned greedy approach, guarantees a constant approximation ratio. Our ultimate algorithm will achieve a ratio of 1/3 in the worst case, but in order to describe it, we will start with somewhat weaker algorithms. The following observation is essential.

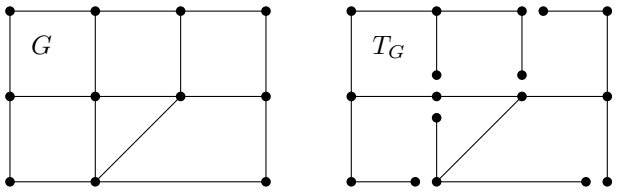


Figure 6: Transforming G into a tree T_G.

Theorem 3 (Wu et al. [25]). *Let T be a tree. We can edge partition T into k subtrees, for $k = 2, 3, 4$, such that the ratio between the number of edges in the smallest and the largest subtree is at most 2. This partition can be found in linear time.*

We note that, at the cost of raising the edge ratio to 3, Wu et al. also present polynomial solutions for larger k. Those are, however, not readily applicable to our approaches. Furthermore observe that there are indeed instances where the ratio 2 is unavoidable.

In our problem, we are given a general weighted graph G, rather than a tree. Therefore, we will first construct a tree T_G from G with the same number of *edges*: cf. Fig. 6. Start with T_G as any spanning tree of G. For every edge $uv \in E(G)$ not initially in T_G, we add a new vertex v' together with the edge uv' into T_G. Observe that this gives a 1-to-1 correspondence between edges of T_G and G.

6.1 Approximation via 2-partition

We start very naïvly, by using Theorem 3 to obtain an edge partition of T_G into two connected components such that the larger component holds at most twice as many edges as the smaller. This induces an edge partition $(F_1 : F_2)$ in G with the same property. For $j = 1, 2$, let H_j denote the subgraph of G induced by F_j. Since each H_j is connected, we can generate a solution graph sequence by first deleting all edges of one of those subgraphs, and only then picking edges of the other. We say we *eat* one component before the other. Thereby, we do not care much about the precise deletion sequence within the component H_j, $j = 1, 2$, as long as the remaining graph (not only the component) remains connected after each deletion step.

Let $w_j := W(H_j)$ and $c_j := |F_j|$ be the weight and the number of edges of the two components, respectively. If we start with eating H_1, we will have at least c_1 subgraphs in our sequence, namely $G_1, \ldots, G_{c_1}$, that contain the whole component H_2. Hence, each G_i, $1 \le i \le c_i$, has weight at least w_2, and the overall solution will have an objective value of at least $c_1 w_2$. By simply choosing the better option between eating H_1 or H_2 first, we can guarantee a solution value of at least $APX := \max\{c_1 w_2, c_2 w_1\}$.

Theorem 4. *The above algorithm achieves an approximation ratio of at least 2/9 and requires $\mathcal{O}(m)$ time.*

Proof. Let v be any vertex contained in both H_1 and H_2. Observe that we do not care how the individual components H_j are eaten, as long as, when eating the first of the two component, the vertex v is retained until the end. This is trivially achieved, for example by a post-order traversal, resulting in the given running time.

In the optimal solution, it may be the case that only a (small) constant number of edges holds all the weight of the graph and can be kept in the graph until the very end. Such a solution would yield an optimum objective value approaching $OPT := m \cdot W$, where $W := W(G)$ is the total sum of all edge weights.

It hence remains to show that $APX/OPT \geq 2/9$. Assume w.l.o.g. that $c_1 \leq c_2$. Recall that $w_1 + w_2 = W$, $c_1 + c_2 = m$, and $c_2 \leq 2c_1$. We hence ask where the function $\max\{c_1 w_2, c_2 w_1\}$ attains its minimum. Straight-forward algebra shows that this happens for $c_1 = m/3, c_2 = 2m/3$ and $w_1 = W/3, w_2 = 2W/3$, and we hence have $c_1 w_2 = c_2 w_1 = 2mW/9 = \frac{2}{9} OPT$. □

6.2 Approximations via 3- and 4-partitions

We can reuse the above ideas to obtain algorithms based on 3- and 4-partitions. However, we have to be more careful in our bounding arguments.

Similarly to before, consider a k-partition ($k = 3, 4$) of T_G subject to Theorem 3, and let $F_1, \ldots, F_k$ be the induced partition edge sets of G. For $j = 1, \ldots, k$, let H_j, $w_j := W(H_j)$, and $c_j := |F_j|$ be the connected edge-induced subgraph, its weight, and its edge cardinality, defined analogously to above.

Consider the auxiliary graph $\mathcal{H}$ on vertices $n_1, \ldots, n_k$. Each vertex n_a ($a = 1, \ldots, k$) in $\mathcal{H}$ represents a component H_a. Two vertices n_a, n_b are adjacent if and only if there exists some vertex in G that is incident to an edge of F_a and to an edge of F_b. Since G is connected, $\mathcal{H}$ is connected as well. Our algorithms will, again, eat our graph componentwise. We may only eat a component H_a if $\mathcal{H}$ remains connected after removing n_a.

First, let us concentrate on $k = 3$. Then $\mathcal{H}$ is either a cycle (triangle) or a path. In the former case, we can start with eating any component. In the latter case, assume w.l.o.g. that n_2 is the inner vertex. We may start with eating either H_1 or H_3. Afterwards, we may choose freely among the two remaining components. Only looking at the values of w_j, c_j for $j = 1, 2, 3$, our algorithm will decide on the best eating order of the three components. There are at most six different orders to consider.

THEOREM 5. *The algorithm that eats the given graph G componentwise in the optimal order, based on a 3-partition subject to Theorem 3, achieves an approximation ratio of at least* 3/10 *and requires* $\mathcal{O}(\,m\,)$ *time.*

PROOF. Since $k = 3$ is a constant, there are only a constant number of eating orders (6, in fact) to check. Eating a single component can again be done arbitrarily, and hence runs in the claimed time for the disjoint components. We may still assume that the optimum solution OPT is (arbitrarily close to) mW.

It suffices to consider the four available eating orders when $\mathcal{H}$ is a path (w.l.o.g. the inner vertex is labeled n_2). If any of these attain $\frac{3}{10} OPT$, the solution for cyclic $\mathcal{H}$ is also at least as good. Assume we eat the components in order $\langle H_1, H_2, H_3 \rangle$. Then, the weight w_2 is retained for the first c_1 many graphs in the solution sequence. The weight w_3 is even retained for at least $c_1 + c_2$ subgraphs. Our algorithm will hence find a solution of objective value at least

$$\zeta := \max \left\{ \begin{array}{ll} c_1 w_2 + (c_1 + c_2) w_3, & c_1 w_3 + (c_1 + c_3) w_2, \\ c_3 w_2 + (c_3 + c_2) w_1, & c_3 w_1 + (c_3 + c_1) w_2 \end{array} \right\}$$

subject to

$$\begin{array}{ll} c_1 + c_2 + c_3 = m, & w_1 + w_2 + w_3 = W, \\ c_a \leq 2c_b, \quad c_a, w_a \geq 0 & \forall a, b \in \{1, 2, 3\}. \end{array}$$

We find the worst possible approximation ratio by asking where ζ attains its minimum. Solving the above system readily gives $\zeta \geq \frac{3}{10} OPT$. This is achieved by setting $c_1 = c_3 = \frac{2}{5} m,\ c_2 = \frac{1}{5} m,\ w_1 = w_3 = \frac{1}{2} W,\ w_2 = 0$. □

Now, consider the case $k = 4$.

THEOREM 6. *The algorithm that eats the given graph G componentwise in the optimal order, based on a 4-partition subject to Theorem 3, achieves an approximation ratio of at least* 1/3 *and requires* $\mathcal{O}(\,m\,)$ *time.*

PROOF. The runtime argument is identical to the one for $k = 3$; we can concentrate on the approximation guarantee.

The auxiliary graph $\mathcal{H}$ can only be one of the following types: a path, a star (with three arms), a cycle, a cycle through three vertices plus one edge, or a complete graph. The latter three types all contain a path through all vertices as a subgraph, and hence allow all eating orders the path allows (plus some additional ones). Hence it suffices to show that if $\mathcal{H}$ is a path or a star, we can always find an eating order achieving ratio 1/3.

If $\mathcal{H}$ is a path, assume w.l.o.g. that the edges are (n_1, n_2), (n_2, n_3), (n_3, n_4). We may start with eating either of the path' two ends (component H_1 or H_4). The second component may then be either the other of those two components, or the component corresponding to the former inner vertex that was incident to the vertex selected first. For the third component, we may freely choose between the two remaining ones. Enumerating all feasible eating orders, we obtain the following function for a lower bound on the solution value.

$$\zeta_{\text{path}} = \max \left\{ \begin{array}{l} c_1 w_2 + (c_1 + c_2) w_3 + (c_1 + c_2 + c_3) w_4, \\ c_1 w_2 + (c_1 + c_2) w_4 + (c_1 + c_2 + c_4) w_3, \\ c_1 w_4 + (c_1 + c_4) w_2 + (c_1 + c_4 + c_2) w_3, \\ c_1 w_4 + (c_1 + c_4) w_3 + (c_1 + c_4 + c_3) w_2, \\ c_4 w_3 + (c_4 + c_3) w_2 + (c_4 + c_3 + c_2) w_1, \\ c_4 w_3 + (c_4 + c_3) w_1 + (c_4 + c_3 + c_1) w_2, \\ c_4 w_1 + (c_4 + c_1) w_3 + (c_4 + c_1 + c_3) w_2, \\ c_4 w_1 + (c_4 + c_1) w_2 + (c_4 + c_1 + c_2) w_3, \end{array} \right\}.$$

Similarly, if $\mathcal{H}$ is a star we obtain the following function; assume w.l.o.g. that n_4 is the central vertex:

$$\zeta_{\text{star}} = \max \left\{ \begin{array}{l} c_1 w_2 + (c_1 + c_2) w_3 + (c_1 + c_2 + c_3) w_4, \\ c_1 w_2 + (c_1 + c_2) w_4 + (c_1 + c_2 + c_4) w_3, \\ c_1 w_3 + (c_1 + c_3) w_2 + (c_1 + c_3 + c_2) w_4, \\ c_1 w_3 + (c_1 + c_3) w_4 + (c_1 + c_3 + c_4) w_2, \\ c_2 w_1 + (c_2 + c_1) w_3 + (c_2 + c_1 + c_3) w_4, \\ c_2 w_1 + (c_2 + c_1) w_4 + (c_2 + c_1 + c_4) w_3, \\ c_2 w_3 + (c_2 + c_3) w_1 + (c_2 + c_3 + c_1) w_4, \\ c_2 w_3 + (c_2 + c_3) w_4 + (c_2 + c_3 + c_4) w_1, \\ c_3 w_1 + (c_3 + c_1) w_2 + (c_3 + c_1 + c_2) w_4, \\ c_3 w_1 + (c_3 + c_1) w_4 + (c_3 + c_1 + c_4) w_2, \\ c_3 w_2 + (c_3 + c_2) w_1 + (c_3 + c_2 + c_1) w_4, \\ c_3 w_2 + (c_3 + c_2) w_4 + (c_3 + c_2 + c_4) w_1 \end{array} \right\}.$$

In either case, we require the following side constraints:

$$\begin{array}{ll} c_1 + c_2 + c_3 + c_4 = m, & w_1 + w_2 + w_3 + w_4 = W, \\ c_a \leq 2c_b, \quad c_a, w_a \geq 0 & \forall a, b \in \{1, 2, 3, 4\}. \end{array} \tag{6}$$

By solving the above systems algebraically, we obtain that

$$\min_{\substack{c_1, \ldots, c_4, \\ w_1, \ldots, w_4, \\ \text{subj. to } (6)}} \{\zeta_{\text{path}}, \zeta_{star}\} \geq \frac{1}{3} OPT.$$

This value is attained, e.g., by ζ_{path} for $c_1 = c_4 = \frac{1}{3}m,\ c_2 = c_3 = \frac{1}{6}m,\ w_1 = w_4 = \frac{1}{2}W,\ w_2 = w_3 = 0$. □

One may wonder if the above strategies can be extended to k-partitions with $k \geq 5$. We only know that an edge ratio of 3 (instead of 2) is achievable for such partitions. This drop seems to always result in worse approximation ratios than the one for the 4-partition.

7. TREE SCHEDULING

We now develop another approximation algorithm, which we call *tree scheduling* since it is based on a more careful consideration of the tree T_G. A version of this algorithm matches our best approximation factor, 1/3, and is exact on trees; recall that the greedy algorithm is arbitrarily bad even on paths (Fig. 2). We also present a more efficient version that works well in practice, but for which we only prove approximation factor 2/9.

A motivation for this more complicated algorithm is that the algorithms from the preceding section may give results that are unreasonable from a practical perspective: they delete some part of the graph completely before moving on to another part of the graph. When applied to road segment selection, this could mean (depending on T_G) that we completely delete one half of the map before deleting even the least important segment in the other half. The tree scheduling algorithm is more sophisticated in that respect, and we have determined experimentally that on realistic instances it approximates very well (see Sec. 8).

The starting point for this algorithm is to look at the EdgeScheduling problem in terms of scheduling theory: consider the edges of the graph as *jobs* to be processed on a single machine: processing the job corresponds to deleting the edge. Then our objective function is known as *total weighted completion time* or *linear delay penalties* (see the related work section). To the best of our knowledge, our connectivity constraint has not been studied in the scheduling literature. There is, however, much work on various other classes of *precedence constraints*, i.e., the requirement that some job i must be processed before job j. A job can only be processed once *all* jobs that must precede it have been processed. For a set of such constraints, the *precedence graph* has an arc (i, j) if job i is constrained to go before job j. If this graph is a rooted forest, then a schedule that optimizes our objective function can be determined in polynomial time using Horn's algorithm [11]. An implementation that runs in $\mathcal{O}(\, n \log n\,)$ time on n jobs is given by Lawler [13].

On trees, we can model the connectivity constraint of EdgeScheduling using a set of precedence constraints. Let T be a tree and r an edge in T. Perform a graph search of the edges of T, starting at r. Since G is a tree, this gives a well-defined parent relation on the edges. This relation is a tree, directed toward r; call it $P(T, r)$, cf. Fig. 7. Interpreting $P(T, r)$ as a set of precedence constraints on the edges, it is equivalent to the connectivity constraint under the assumption that r is deleted last: if indeed we delete r last, then for any edge e, all its descendants in $P(T, r)$ must be deleted before we can delete e.

Theorem 7. EdgeScheduling *on a tree $T = (V, E)$ can be solved optimally in $\mathcal{O}(\, |V|^2 \log |V|\,)$ time.*

Proof. Let r be any edge in T, and assume it is deleted last in an optimum solution. Then $P(T, r)$, computable in linear time, exactly models the connectivity constraint. An optimal scheduling of $P(T, r)$, obtained by Lawler's algorithm, gives an optimal solution to EdgeScheduling. Running this subalgorithm for each possible choice of r achieves the theorem. □

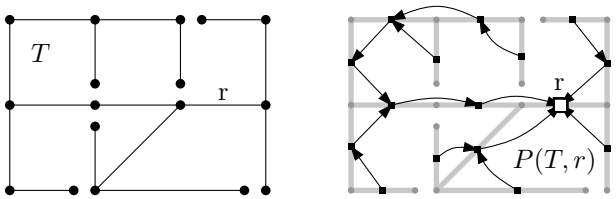


Figure 7: Transforming a tree into a set of job precedences $P(T, r)$, starting from some root r.

Now we consider general graphs. First we construct a tree T_G as in the preceding section. Then we compute schedules based on $P(T_G, r)$, for one or more root edges r. This clearly guarantees connectivity of the remaining graph at every step, but forcing connectivity *along* T_G introduces unintended restrictions on the schedule: by taking this step we lose optimality (but recall that the full problem is $\mathcal{NP}$-hard). We could test every possible root, like in Theorem 7.

Theorem 8. *Testing all possible roots, the above algorithm runs in $\mathcal{O}(\, m^2 \log m\,)$ time and has an approximation ratio of at least 1/3 on general graphs.*

Proof. Construct T_G as before; this can be done in linear time. Construct $P(T_G, r)$ for every edge $r \in E$. Solve each of these scheduling problems and return the best result. Using Lawler's algorithm gives the claimed running time.

Now consider the solution found by the approximation algorithm in Theorem 6, which has approximation factor at least 1/3. Let r be the edge that it deletes last. Since this solution keeps T_G connected and ends at r, it satisfies $P(T_G, r)$; it is otherwise arbitrary. The tree scheduling algorithm finds the optimal scheduling under these constraints. Hence, when the correct edge r is used, the algorithm's approximation factor is at least as good. The algorithm tries all r. □

We now consider a faster variant of the algorithm that checks only a single root; we can efficiently find a root that results in approximation ratio 2/9.

Theorem 9. *Using a single root found in linear time, the above algorithm runs in $\mathcal{O}(\, m \log m\,)$ time and has an approximation ratio of at least 2/9 on general graphs.*

Proof. Consider the factor-2/9 approximation algorithm from Theorem 4. It uses a balanced edge partition $(F_1 : F_2)$ of T_G and decides, based on the number of edges and the total weight in each, to *eat* one of F_1 or F_2 completely before the other. This is what gives the approximation guarantee; within F_1 and F_2, the order of deletion is arbitrary. W.l.o.g. let F_1 be the set of edges it eats first and let r be any edge in F_2 that has a neighboring edge in F_1. Then $P(T_G, r)$ allows a solution that eats F_1 before F_2; the algorithm finds, in fact, the optimal solution subject to $P(T_G, r)$, hence its approximation factor is at least as good. The runtime follows immediately, since we test only one root and this root can be found in linear time (Theorem 3). □

8. COMPUTATIONAL EXPERIMENTS

We have implemented GreedySelection, the tree scheduling algorithm and the integer linear program (ILP) in C++, using CPLEX 12.6 for solving integer linear programs. Runtimes were measured on an Intel® Core™ i5-2520M CPU at 2.5 GHz; memory usage was not an issue.

For the tree scheduling algorithm, we get T_G by taking a maximum-weight spanning tree and extending it to contain all edges (as in Fig. 6). Unless otherwise noted, our implementation tests only a single root: the heaviest edge in the input is used for r when constructing $P(T_G, r)$. Then the approximation ratio 2/9 from Theorem 9 is not guaranteed, but the experiments show that this choice of r works well. We call this implementation TreeSchedule.

The implementation contains two departures from the earlier description of the algorithms. Both changes result in a simpler implementation with a worse asymptotic runtime, but do not affect the output. Since we observe a low runtime in practice, we would recommend this implementation unless the runtime is explicitly found to be problematic. We have implemented GreedySelection without the dynamic connectivity data structure: we simply check connectivity via depth-first search. We have implemented TreeSchedule using Horn's algorithm [11] instead of Lawler's algorithm [13].

In this paper we present experiments using two real-world data sets. The `Wuerzburg` data set is a crop of the *OpenStreetMap* road network of Würzburg, Germany,[1] and consists of 2995 road segments. The `Dallas` data set is the largest connected component in the *City of Dallas GIS Services*' road network of Dallas, Texas.[2] This data set contains 3×10^5 road segments.

In order to determine weights for the road segments, we have used either betweenness centrality or PageRank, as indicated per experiment; these are standard choices for scoring the importance of road segments [12, 21]. It is also common to generalize based on *road class* information, which is available in our data sets. We have multiplied the weight of some road segments by a factor of 3 or 5 based on the road class.[3] In this way, the optimization problem asks for the lower-class roads to be deleted first, unless they are structurally important (either for the connectivity constraint or because of high centrality). Though somewhat arbitrary, this choice of weights does not seem particularly objectionable and gives reasonable results.

`Dallas` and `Wuerzburg` are much too large to solve optimally using the integer linear program. We have used the following procedure to generate realistic small data sets from a large data set. We start at a random vertex in the road network and perform a breadth-first search until a desired number of vertices is reached. We take the road network induced by these vertices (which will have some more road segments than vertices). We generated 150 such random subgraphs of `Wuerzburg` with 40 vertices, a further 50 subgraphs with 50 vertices, and 15 subgraphs with 60 vertices.

First we consider the integer linear program. Fig. 8(a) shows a histogram of runtimes on the random 40-vertex subgraphs of `Wuerzburg`. The average runtime is 4.5 s, with a few outliers up to 90 s. This increases to an average of 16.8 s for the 50-vertex instances and further to 118 s for the 60-vertex instances. By then, runtimes of multiple minutes are common. Results on `Dallas` are similar, though slightly faster because the network is sparser, resulting in graphs with fewer edges.

Next, we consider GreedySelection and TreeSchedule. On the small instances from the preceding paragraph, both algorithms require virtually no time. Figs. 8(b) and 8(c) show a histogram of their observed approximation ratio, which we can determine since the ILP gave us an optimum solution. Both algorithms give very good solutions on these instances, always staying within 99% of the optimum. We also see that, of the two algorithms, TreeSchedule gives better solutions, even finding the optimum in 85% of the instances.

The tree scheduling algorithm involves the choice of a root. Depending on the chosen root, it performs better or worse than the greedy algorithm, but there is almost always a root that performs at least as well. A typical situation is displayed in Fig. 9(a), where GreedySelection achieves an approximation quality of 99.5%: the tree scheduling algorithm gives better solutions only for about 1/3 of the possible roots. However, simply picking the heaviest edge in the graph as the root turns out to dominate the greedy approach in all the 40-vertex graphs tested (150 graphs) and in 98% of the 50-vertex graphs (50 graphs).

Finally, we evaluate the algorithms on the full data sets, see Figures 1 and 10 for graphical snapshots. On these graphs we cannot feasibly determine the optimum, but can compare GreedySelection and TreeSchedule. In Fig. 9(b) we see that on `Wuerzburg`, the tree scheduling algorithm is consistently better than the greedy algorithm, but not by much. This is the case almost regardless of the choice of the root: the quality still depends on the chosen root, yet on this graph the tree scheduling algorithm dominates the greedy algorithm in all but 7 out of 2995 possible roots (the heaviest edge was, again, one of the dominating choices). The results on `Dallas` are similar. This is in stark contrast to the smaller instances discussed before (Fig. 9(a)). It appears that on larger instances, it becomes more important to do "smart" scheduling than it is to be (un)lucky with selecting the root. This shows that the full enumeration over all possible roots is not essential on large graphs. This is convenient since trying all m roots may easily become too costly in practice. On smaller graphs, however, where the algorithms are fast anyhow, testing all roots is worthwhile and dominates the greedy approach.

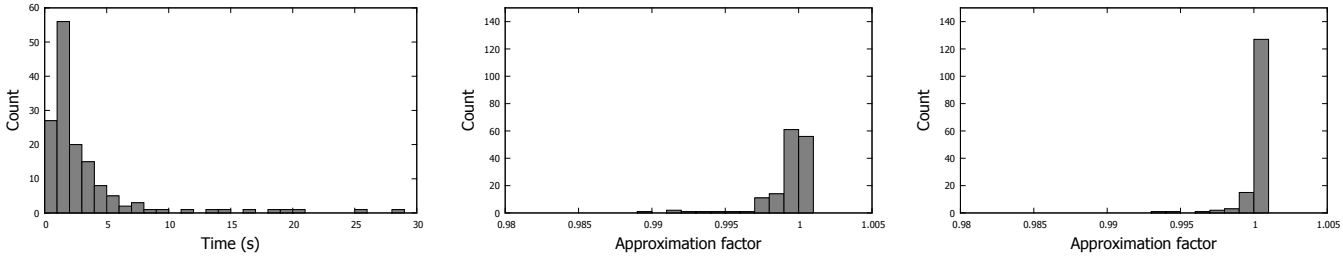
9. CONCLUSION & DISCUSSION

The solutions given by some of our approximation algorithms (Sec. 6) can be, in a sense, unreasonable: completely delete a particular area of the graph, before deleting anything else. We have two remarks about this. On the one hand, we give a heuristic (Sec. 3) and a different approximation algorithm (Sec. 7) and have found that they do give reasonable results in practice. On the other hand, it does suggest that our current problem statement does not model all relevant criteria of map generalization: if it is possible for a solution to be evaluated as good, even though it is actually bad, then we may need to adjust the objective function and/or constraints. We are not deterred by this state of affairs, since it is only natural that the correct model is not immediately clear when first introducing a rigorous op-

[1] `http://download.geofabrik.de/`

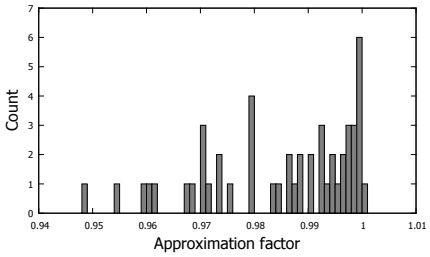
[2] `http://gis.dallascityhall.com/`

[3] We have multiplied the weight of road segments marked `primary`, `EXPY`, `FWY` or `TPKE` by a factor 5, and road segments marked `secondary`, `HWY` or `BLVD` by a factor 3.

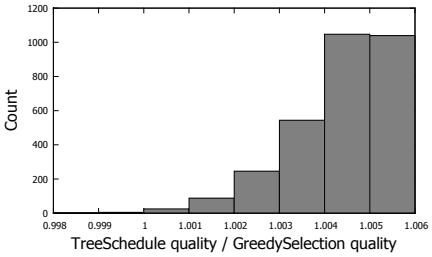


(a) Integer linear program (b) GreedySelection (c) TreeSchedule

Figure 8: Histograms of runtime and observed approximation factor on 150 subgraphs of `Wuerzburg`, each with 40 vertices and on average about 60 edges. Edge weights are set using betweenness centrality and road class. Note that the rightmost bar in Figs. 8(b) and 8(c) represents factor 1, that is, optimal solutions.



(a) Observed approximation ratio on a 40-vertex subgraph of `Wuerzburg` (49 edges). Observe that an optimum solution is found using one of the roots.



(b) Ratio of solution quality of TreeSchedule w.r.t. GreedySelection on the full road network `Wuerzburg`. Observe that the former dominates the latter for almost every possible choice of root.

Figure 9: Quality of TreeSchedule for each possible root; weights set using PageRank and road class.

timization approach to a new topic. Future work will have to address these issues.

For example, in addition to global connectivity, we would like to ensure that, at every scale, the edges belonging to the same stroke form a connected path. This would help readability of the network. Furthermore, we would like to ensure that, at every scale, the graph nicely spans the map plane, that is, it must not leave large parts of the map empty. Like in the literature on static map generalization, it seems beneficial to formalize these (and other) considerations, so that proper optimization techniques may be applied.

10. REFERENCES

[1] M. Aliakbarian. Derivation of continuous zoomable road network maps through utilization of space-scale-cube. Master's thesis, Technische Universität Dresden, Germany, 2014.

[2] K. Been, M. Nöllenburg, S.-H. Poon, and A. Wolff. Optimizing active ranges for consistent dynamic map labeling. *CGTA*, 43(3):312–328, 2010.

[3] D. Burghardt, S. Schmid, and J. Stoter. Investigations on cartographic constraint formalisation. In *Proc. 11th ICA Workshop on Generalisation and Multiple Representation*, 2007.

[4] A. Cecconi and M. Galanda. Adaptive zooming in web cartography. *Computer Graphics Forum*, 21(4):787–799, 2002.

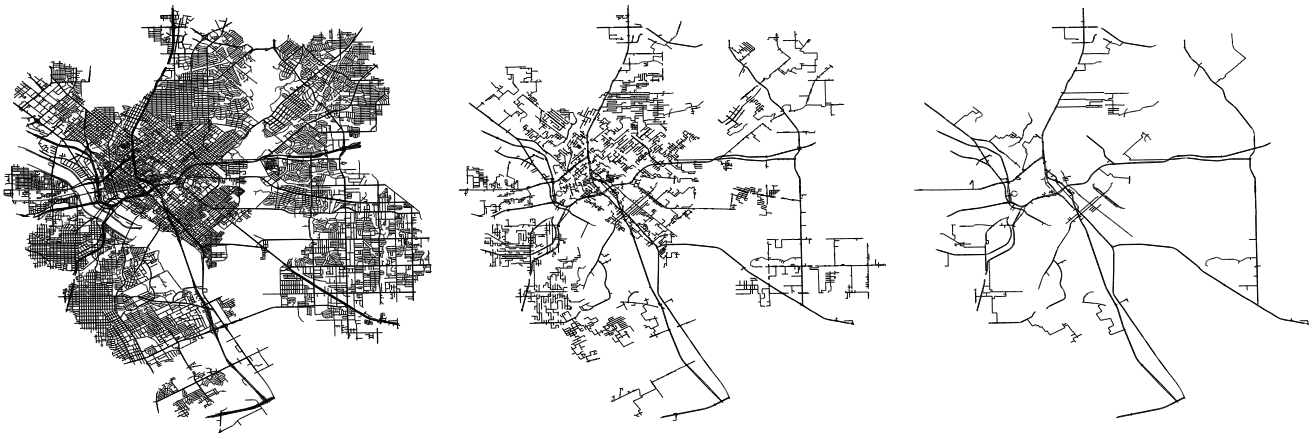
[5] C. Duchêne, B. Baella, C. A. Brewer, D. Burghardt, B. P. Buttenfield, J. Gaffuri, D. Käuferle, F. Lecordix, E. Maugeais, R. Nijhuis, M. Pla, M. Post, N. Regnauld, L. V. Stanislawski, J. Stoter, K. Tóth, S. Urbanke, V. van Altena, and A. Wiedemann. Generalisation in practice within national mapping agencies. In D. Burghardt, C. Duchêne, and W. Mackaness, editors, *Abstracting Geographic Information in a Data Rich World*, Lecture Notes in Geoinformation and Cartography, pages 329–391. Springer-Verlag, Berlin, Germany, 2014.

[6] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Co., New York, USA, 1979.

[7] A. Gemsa, B. Niedermann, and M. Nöllenburg. Trajectory-based dynamic map labeling. In *Proc. 24th Internat. Sympos. Algorithms and Computation (ISAAC '13)*, volume 8283 of *Lecture Notes in Computer Science*, pages 413–423. Springer-Verlag, Berlin, Germany, 2013.

[8] A. Gemsa, M. Nöllenburg, and I. Rutter. Consistent labeling of rotating maps. In *Proc. 12th Internat. Sympos. Algorithms and Data Structures (WADS '11)*, volume 6844 of *Lecture Notes in Computer Science*, pages 451–462. Springer-Verlag, Berlin, Germany, 2011.

[9] J.-H. Haunert and A. Wolff. Area aggregation in map generalisation by mixed-integer programming. *International Journal of Geographical Information Science*, 24(12):1871–1897, 2010.



(a) Input road network (G_0). (b) After 90000 edge deletions (G_{90000}). (c) After 101000 edge deletions (G_{101000}).

Figure 10: Three steps of TreeSchedule's solution on a 10^5-vertex subgraph of `Dallas`. Edge-weights are set using PageRank and road class as described in Sec. 8; due to the scale of the map, edge weight is not indicated.

[10] J. Holm, K. de Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48(4):723–760, jul 2001.

[11] W. Horn. Single-machine job sequencing with treelike precedence ordering and linear delay penalties. *SIAM Journal on Applied Mathematics*, 23(2):189–202, 1972.

[12] B. Jiang and C. Claramunt. A structural approach to the model generalization of an urban street network. *Geoinformatica*, 8(2):157–171, 2004.

[13] E. L. Lawler. Sequencing jobs to minimize total weighted completion time subject to precedence constraints. *Annals of Discrete Mathematics*, 2:75–90, 1978.

[14] W. A. Mackaness and M. K. Beard. Use of graph theory to support map generalization. *Cartography and Geographic Information Systems*, 20:210–221, 1993.

[15] M. Nöllenburg, D. Merrick, A. Wolff, and M. Benkert. Morphing polylines: A step towards continuous generalization. *Computers, Environment and Urban Systems*, 32(4):248–260, 2008.

[16] N. Schwartges, D. Allerkamp, J.-H. Haunert, and A. Wolff. Optimizing active ranges for point selection in dynamic maps. In *Proc. 16th ICA Generalisation Workshop*, 2013.

[17] M. Sester. Optimization approaches for generalization and data abstraction. *International Journal of Geographical Information Science*, 19(8–9):871–897, 2005.

[18] R. C. Thomson and R. Brooks. Exploiting perceptual grouping for map analysis, understanding and generalization: The case of road and river networks. In *Selected Papers from the Fourth Internat. Workshop Graphics Recognition Algorithms and Applications (GREC '01)*, volume 2390 of *Lecture Notes in Computer Science*, pages 148–157. Springer-Verlag, Berlin, Germany, 2002.

[19] F. Töpfer and W. Pillewizer. The principles of selection. *The Cartographic Journal*, 3(1):10–16, 1966.

[20] T. C. van Dijk, K. Fleszar, J.-H. Haunert, and J. Spoerhase. Road segment selection with strokes and stability. In *Proc. 1st ACM SIGSPATIAL Workshop on MapInteraction*, pages 72–77, 2013.

[21] T. C. van Dijk and J.-H. Haunert. A probabilistic model for road selection in mobile maps. In *Proc. 12th Internat. Sympos. Web and Wireless Geographical Information Systems (W2GIS '13)*, volume 7820 of *Lecture Notes in Computer Science*, pages 214–222. Springer-Verlag, Berlin, Germany, 2013.

[22] M. van Kreveld. Smooth generalization for continuous zooming. In *Proc. 20th Internat. Cartographic Conference (ICC '01)*, 2001.

[23] P. van Oosterom and M. Meijers. Vario-scale data structures supporting smooth zoom and progressive transfer of 2d and 3d data. *International Journal of Geographical Information Science*, 28(3):455–478, 2014.

[24] J. M. Ware, C. B. Jones, and N. Thomas. Automated map generalization with multiple operators: a simulated annealing approach. *International Journal of Geographical Information Science*, 17(8):743–769, 2003.

[25] B. Y. Wu, H.-L. Wang, S. T. Kuan, and K.-M. Chao. On the uniform edge-partition of a tree. *Discrete Applied Mathematics*, 155(10):1213–1223, 2007.

[26] B. Yang, X. Luan, and Q. Li. Generating hierarchical strokes from urban street networks based on spatial pattern recognition. *International Journal of Geographical Information Science*, 25(12):2025–2050, 2011.

[27] Q. Zhang. Road network generalization based on connection analysis. In *Developments in Spatial Data Handling – Proc. 11th Internat. Sympos. Spatial Data Handling (SDH '04)*, pages 343–353. Springer-Verlag, Berlin, Germany, 2005.