Efficient Generation of Geographically Accurate Transit Maps

HANNAH BAST and PATRICK BROSI, University of Freiburg, Germany
SABINE STORANDT, University of Konstanz, Germany

We present LOOM (Line-Ordering Optimized Maps), an automatic generator of geographically accurate transit maps. The input to LOOM is data about the lines of a transit network: for each line, its station sequence and geographical course. LOOM proceeds in three stages: (1) construct a line graph, where edges correspond to network segments with the same set of lines following the same course; (2) apply a set of local transformation rules that compute an optimal partial ordering of the lines and speed up the next stage; (3) construct an Integer Linear Program (ILP) that yields a line ordering for each edge and minimizes the total number of line crossings and line separations; and (4) based on the line graph and the computed line ordering, draw the map. As our maps respect the geography of the transit network, they can be used as overlays in typical map services. Previous research either did not take the network geography into account or was only concerned with schematic metro map layouting. We evaluate LOOM on six real-world transit networks, with line-ordering search-space sizes up to 2×10^{267}. Using our transformation rules and an improved ILP formulation, we compute optimal line orderings in a fraction of a second for all networks. This enables interactive use of our method in map editors.

CCS Concepts: • **Human-centered computing** → **Graph drawings**; • **Theory of computation** → **Integer programming**;

Additional Key Words and Phrases: Public transit network, graph drawing, map generation, graphical optimization

ACM Reference format:
Hannah Bast, Patrick Brosi, and Sabine Storandt. 2019. Efficient Generation of Geographically Accurate Transit Maps. *ACM Trans. Spatial Algorithms Syst.* 5, 4, Article 25 (September 2019), 36 pages.
https://doi.org/10.1145/3337790

1 INTRODUCTION

Cities with a public transit network usually have an iconic map that illustrates the network. There is usually a picture of it at every larger station. Similarly, most map services nowadays feature a transit layer that displays all lines and stations in the currently selected area. Both kinds of maps have in common that they satisfy the following criteria:

(1) The map should depict the topology of the network: which transit lines are offered, which stations do they serve in which order, and which transfers are possible.

Authors' addresses: H. Bast and P. Brosi, University of Freiburg, Georges-Köhler-Allee 51, 79117 Freiburg, Germany; emails: {bast, brosi}@cs.uni-freiburg.de; S. Storandt, University of Konstanz, Universitätsstraße 10, Postbox 67, 78457 Konstanz, Germany; email: sabine.storandt@uni-konstanz.de.

2374-0353/2019/09-ART25 $15.00
https://doi.org/10.1145/3337790

(2) The map should be neatly arranged and esthetically pleasing.
(3) The map should reflect the geographical course of the lines, at least to some extent.

So far, such maps have been designed and drawn by hand. Concerning (3), the designers usually take some liberty to make the map fit into a certain format, to simplify the layout, or both.

The goal of this article is to produce transit maps fully automatically, adhering to (3) rather strictly: within a given tolerance, the lines on the map should be drawn according to their geographical course. This gives rise to several algorithmic challenges; in particular, because the geographical course of some lines may overlap partially. These lines should then of course not be rendered on top of each other, as this would obfuscate the visibility. Instead, they should be drawn next to each other. This requires us to first identify overlapping parts and then to choose the line ordering in the rendered map. A bad ordering can lead to many unnecessary line crossings. Hence our goal is to find orderings that minimize these undesired crossings. As the number of possible orderings exceeds an octillion even for the transit network of medium sized cities, we need to develop efficient methods to find the best ordering in reasonable time.

1.1 Overview and Definitions

LOOM (Line-Ordering Optimized Maps) proceeds in four stages, which we briefly describe in the following along with some notation and terminology that will be used throughout the article: (1) construct a so-called line graph, where edges correspond to segments of the network with the same set of lines following the same course; (2) apply a set of transformation rules to the line graph that simplify the graph and compute an optimal partial ordering of the lines; (3) construct an ILP that yields a line ordering for each edge that minimizes the total number of line crossings and line separations; and (4) based on the line graph and the computed line ordering, draw the map. Each stage is described in more detail in one of the following sections.

Input: The input to LOOM is a set $\mathcal{S}$ of stations and a set $\mathcal{L}$ of lines. Each station has a geographical location. Each line has a unique ID (in our examples: numbers) and information about the sequence of stations it serves and the geographical course between them. These data are usually provided as part of a network's GTFS feed.

Line graph construction (Section 2): In the first stage, LOOM computes a *line graph*. This is an undirected labeled graph $G = (V, E, L)$, where $V \supseteq \mathcal{S}$ (each station is a node, but there may be additional nodes), E is the set of edges, and each $e \in E$ is labeled with a subset $L(e) \subseteq \mathcal{L}$ of the lines. Intuitively, each edge corresponds to a segment of the network, where the same set of lines takes the same geographical course (within a certain tolerance), and there is a node wherever such a set of lines splits up in different directions. Figure 1 (left) shows the line graph for an excerpt from the light rail network of Stuttgart, Germany. We will see that the complexity of our algorithms in Sections 3 and 4 depends on $M = \max_{e \in E} |L(e)|$, the maximum number of lines per segment. The line graph construction is described in Section 2.

Line graph simplification (Section 4): In the second stage, LOOM transforms the line graph such that the optimization problem of the third stage becomes simpler. Two of these transformations are relatively simple: pruning of nodes of degree 2 (thus making the graph smaller) and cutting the graph into independent components (which can then be solved independently). The most powerful transformation is what we call *graph untangling*. It consists of a set of local transformation rules that can be applied iteratively as long as the graph has local structures where one of the rules apply. A detailed description of these rules (along with a figure for each rule) is provided in Section 4.3. The ILP can be formulated on the original line graph, as well as on the transformed line graph. It turns out that the ILP on the transformed line graph can be solved much

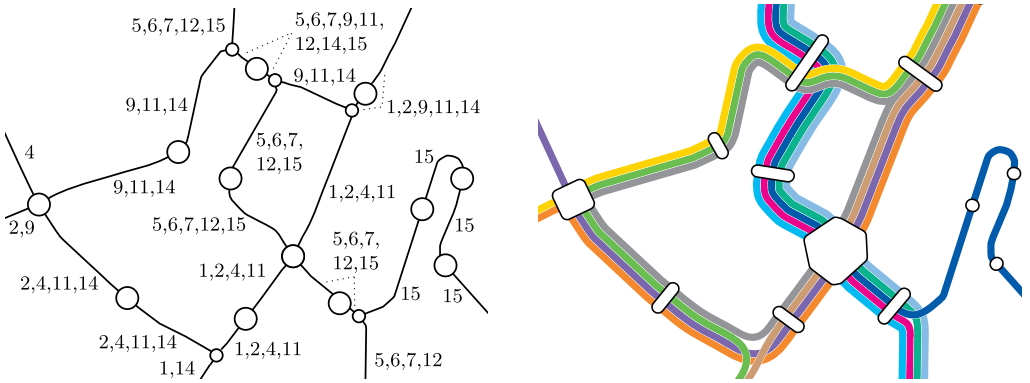

Fig. 1. Left: Excerpt from a line graph that LOOM constructed from the GTFS data for the 2015 light rail network of the city of Stuttgart, Germany. Each edge corresponds to a segment of the network where the same set of lines takes the same geographical course. Segment boundaries are often station nodes (large) but may also be intermediate nodes (small). The line ids for each segment are given in ascending order. LOOM's central optimization step computes a line ordering for each segment. This determines how the lines are drawn in the map and where line crossings and separations occur. Right: The corresponding excerpt from LOOM's transit map.

more efficiently. The line ordering optimization problem on the transformed line graph may even be solvable by a simple exhaustive search for some real-world instances.

Line ordering optimization (Section 3): In the third stage, LOOM computes an *ordering* of $L(e)$ for each $e \in E$. This ordering determines where line crossings and separations occur and is hence critical for the final map appearance. Previous research referred to the problem of minimizing crossings as the metro-line crossing minimization problem (MLCM), see Section 1.3. We formulate two strongly related problems: the metro-line node crossing minimization problem (MLNCM) and a variant with a line separation penalty (MLNCM-S) and give a concise Integer Linear Program (ILP) to solve instances of these problems.

Note that we describe the line graph simplification stage (Section 4) *after* the line-ordering optimization (Section 3), although, technically, the simplification is done before the line-ordering optimization in LOOM. We do this for didactic reasons: The line-ordering optimization can also be solved without the simplification, just much less efficiently so, and the intuition behind the simplification rules is much better understood after it is clear how the line-ordering optimization works.

Rendering (Section 5): In the fourth stage, LOOM draws the transit map based on the line graph from stage 1 and the ordering from stages 2 and 3. Each station node v is drawn as a polygon, where each side of the polygon corresponds to exactly one incident edge of v. We call this side the *node front* of that edge at that node. The node front for an edge e has $|L(e)|$ so-called *ports* (Figure 3). Drawing the map then amounts to connecting the ports (according to the ordering computed in stages 2 and 3) and drawing the station polygons. Figure 1 (right) shows a rendered transit map after layout optimization.

1.2 Contributions

We consider the following as the main contributions of our article.

- We present a new automatic map generator, called LOOM, for geographically accurate transit maps. The input is basic schedule data as provided in a GTFS feed. This is, as far as we

know, the first research paper on this problem in its entirety. Previous research work considers only parts of this problem (oblivious either to the geographical course or to the order of the lines) and does not yield maps that can be used for tiles and overlays in typical map services.
- We provide a new problem formulation that resolves several issues from previous formulations. In particular, our approach is not restricted to planar graphs and it does not require an artificial grouping of crossing (it happens naturally with our approach).
- We provide a sophisticated ILP formulation, which can be solved in practice even for our largest network. As we show, the straightforward ILP formulation can only be solved for small networks.
- We provide a set of graph transformation rules, in particular what we call graph untangling, which speed up ILP solution times by up to an order of magnitude.
- We evaluate LOOM on the transit network of three cities in Europe and three cities in the U.S., the largest being New York. Using our graph transformation rules and the sophisticated ILP formulation, an optimal line ordering can be computed for each network in a fraction of a second.
- For comparison, we also evaluate three standard heuristic optimization methods (exhaustive search, steepest-ascent hill climbing, simulated annealing) for the line-ordering optimization problem.
- Our maps are publicly available online.[1]

1.3 Related Work

Our work is related to previous work on map construction, edge bundling, crossing minimization in metro maps, graph untangling of planar graphs, and drawing of schematic metro maps.

1.3.1 Map Construction and Edge Bundling. The first step in our pipeline—the line graph construction—is closely related to map construction and edge bundling.

The goal of *map construction* algorithms is producing the graph of an underlying (street) network from vehicle trajectory data. There is a variety of map construction algorithms described in the literature; see Reference [1] for an overview. For example, in Reference [2], an incremental approach is used that starts with an empty map and incrementally updates the network graph with new trajectories. New trajectories are partially map-matched to existing graph segments with a global distance threshold and their geometries updated accordingly, while unmatched parts introduce new edges (and thus intersection nodes). The main difference between existing work (on street networks) and our approach is that our input data already represents a multigraph (with stations as intersection nodes) and is usually quite sparse.

The goal of *edge bundling* in general networks is to group edges to save ink when drawing the network. Usually, the embedding of the edges is not fixed *a priori* but can be chosen such that many bundles occur (possibly respecting side constraints, like edges being short). For example, in Reference [15] a force-directed heuristic was described where edges attract other edges to form bundles automatically. For our problem, we are not allowed to embed edges arbitrarily, as we want to maintain the geographical course of the vehicle trajectories. In Reference [19], edge bundling in the context of metro line map layouts was discussed, also considering orderings within the bundles to minimize crossings. But for their approach to work, the underlying graph has to fulfill a set of restrictive properties. For example, the so-called *path terminal property* demands that a node in the graph cannot be an endpoint of one line and an intermediate node of another line at the same

[1] http://loom.informatik.uni-freiburg.de.

time. But this structure regularly appears in real-world datasets. For example, a local train might end at the main station of a town, while a long-distance train might have this station only as an intermediate stop. Also, self-intersections are forbidden, which excludes instances with cyclic subway lines. With these additional properties required in Reference [19], the problem becomes significantly easier but is no longer compatible with most real-world datasets. In contrast, our line graph construction and subsequent crossing minimization algorithms are compatible with real-world inputs of arbitrary structure.

1.3.2 Crossing Minimization. Previous research on the metro-line crossing minimization problem (MLCM), as briefly summarized in the following, typically comes without experimental evaluations and without the production of actual maps. The problem of minimizing intra-edge crossings in transit maps was introduced in Reference [7], with the premise of not hiding crossings under station markers for aesthetic reasons. A polynomial time algorithm for the special case of optimizing the layout along a single edge was described. The term MLCM was coined in Reference [6]. In that paper, optimal layouts for path and tree networks were investigated, but arbitrary graphs were left as an open problem. In References [3, 4, 17], several variants of MLCM were defined and efficient algorithms were presented for some of these variants, often with a restriction to planar graphs. In Reference [5], an ILP formulation for MLCM under the periphery condition (lines ending in a station must be drawn at the left- or rightmost position in incident edges) was introduced. The resulting ILP was shown to have a size of $O(|L|^2|E|)$ with L being the set of lines and E the set of edges in the derived graph. In Reference [13], it was observed that crossings scattered along a single edge are also not visually pleasing, and hence crossings were grouped into so-called block crossings. The problem of minimizing the number of block crossings was shown to be NP-hard on simple graphs just like the original MLCM problem [12]. Our adapted MLNCM problem has the same complexity as MLCM and is hence also NP-hard.

1.3.3 Graph Untangling. In Section 4, we describe a set of so-called *graph untangling* rules that may be applied to a MLNCM problem instance to reduce the overall search space size. To the best of our knowledge, these graph untangling rules have not been described in the context of the metro-line crossing minimization problem so far.

The term graph untangling has previously been used in the literature to denote the untangling of planar graphs [8, 14], with the goal of producing straight-line drawings of those graphs without any lines crossing each other by moving some (or the minimal subset of) nodes. While our line graph model may indeed be understood as a multigraph, we are not allowed to move nodes freely and generally look for a line drawing with a minimal number of crossings, which must not necessarily be zero. Additionally, we do not require our line graphs to be planar.

1.3.4 Schematic Metro Maps. Another line of research focuses on drawing *schematic* metro maps, for example, by restricting the representation of transit lines to octilinear polylines [16] or Bézier curves [11]. See Reference [18] for a recent survey on automated metro map layout methods. These approaches strongly abstract from the geographical course of the lines (and often also from station positions), and the minimization of line crossings or separations is usually not part of the problem. In particular, the resulting maps cannot be used for tiles or overlays in typical map services.

1.3.5 Other Work. There is also some applied work on transit maps, but without publications of the details. One approach that seems to use a model similar to ours was described by Anton Dubreau in a blog post [10] although without a detailed discussion of their method. As far as we are aware there are no papers on MLCM that deal with real public transit data.

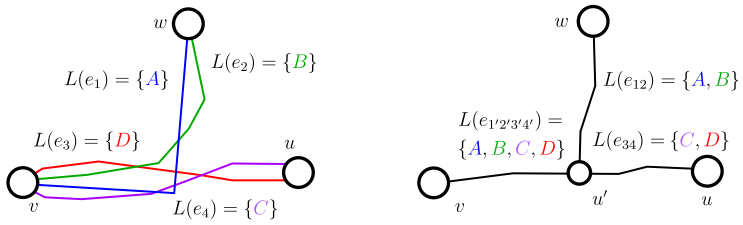

Fig. 2. Left: Input multigraph G^0 created from schedule data (GTFS). Nodes represent stations, and each edge holds a single line that occurs between two stations. There may be many overlapping edge segments. Right: Line graph G constructed from G^0 by repeatedly combining shared edge segments into a single, new edge. The overlapping segments have been collapsed, and a new node u' was introduced at the segment boundaries.

2 LINE GRAPH CONSTRUCTION

This section describes stage 1 of LOOM: Given line data, construct the line graph. We assume that the data are given in the GTFS format [9]. In GTFS, each trip (that is, a concrete tour of a vehicle of a line) is given explicitly and the graph G^0 formed by all station coordinates and the trips between them has many overlapping edges that may (partially) share the same path (Figure 2 (left)).

Let e_1, e_2 be two edges in G^0 with geometrical paths τ_1 and τ_2. For each τ, we define a parametrization $p_\tau(t) : [0, 1] \mapsto \mathbb{R}^2$, which maps the progress t to a point on τ (e.g., if the length of τ is 10 meters, then $p_\tau(\frac{1}{2})$ returns the point we would reach after traveling on τ for 5m). We call $(t, t'), t' \geq t$ a segment of e. To decide whether a segment $(t_1, {t_1}')$ of e_1 is similar to a segment $(t_2, {t_2}')$ of e_2, we use a simple approximation. For a distance threshold $\hat{d}(e_1, e_2)$, we say $((t_1, t_2), (t_1', t_2'))$ is a shared segment of e_1 and e_2 if

$$\forall u \in [t_1, {t_1}'] : \exists u' \in [t_2, {t_2}'] : \|p_{\tau_1}(u) - p_{\tau_2}(u')\| \leq \hat{d}(e_1, e_2), \tag{1}$$

that is, if for every point $p_{\tau_1}(u)$ on τ_1, there exists a corresponding point $p_{\tau_2}(u')$ on τ_2 within the threshold distance $\hat{d}(e_1, e_2)$.

As we want to avoid overlapping lines during rendering, we have to chose $\hat{d}(e_1, e_2)$ in such a way that there will be enough space between the edges in the final line graph. Let w be the desired width of a single line in the rendered map. The definition

$$\hat{d}(e_1, e_2) = \frac{w|L(e_1)| + w|L(e_2)|}{2} \tag{2}$$

satisfies this, as we need $w|L(e)|/2$ map units of space on either side of e to render all $l \in L(e)$ with width w (see Section 5).

We transform G^0 into a line graph G by repeatedly combining a shared segment between two edges $e_1 = \{u_1, v_1\}$ and $e_2 = \{u_2, v_2\}$ into a single new edge e_{12} until no more shared segments can be found. The path τ_{12} of e_{12} is averaged from the shared segments on e_1 and e_2, and we set $L(e_{12}) = L(e_1) \cup L(e_2)$ (Figure 2 (right)). Two new non-station nodes u' and v' that mark the beginning and end of the shared segment are introduced and split e_1 and e_2 such that $e_1 = \{u_1, u'\}$, $e_2 = \{u_2, u'\}$, $e_1' = \{v', v_1\}$, $e_2' = \{v', v_2\}$, and $e_{12} = \{u', v'\}$. Note that the new non-station nodes v' and u' will always have a degree of 3. After each iteration, we obtain from G^i a new graph G^{i+1}. If the distance between a node v' added to G^{i+1} and an existing node v in G^i is smaller than $\hat{d}(e_1, e_2)$ after collapsing an e_1 and e_2, then we merge v and v' to avoid cluttering the graph with many start and end nodes of shared segments.

To find the shared segments between e_1 and e_2, we sweep over τ_1 in n steps of some Δt, measuring the distance d between $p_{\tau_1}(i \times \Delta t)$ and τ_2 at each $i < n$ along the way. If $d \leq \hat{d}(e_1, e_2)$, then we start

a new shared segment. If $d > \hat{d}(e_1, e_2)$ and a shared segment is open, then we close it. For our test datasets, we found that a Δt of 10m is usually small enough to achieve satisfying results.

The algorithm can be made more robust against outliers by allowing d to exceed $\hat{d}(e_1, e_2)$ for a number of k steps. It can be sped up by indexing every linear segment of every path in a geometric index. Just like in previous work on incremental map construction, the results of our algorithm depend on the order in which the segments are combined. For our evaluation in Section 6, we used a random order.

3 LINE ORDERING OPTIMIZATION

This section describes stage 2 of LOOM, namely how to solve the metro-line node crossing minimization problem (MLNCM): Given a line graph, compute an ordering of the lines for each edge such that the total number of crossings in the final map is minimized. Contrary to the classic MLCM problem, which imposes a right and left ordering for the $L(e)$ on each edge $e = \{u, v\}$ (one ordering for u, one ordering for v) and allows crossings to occur anywhere on an edge if the two orderings do not match, MLNCM imposes exactly one ordering for each edge and restricts crossings to nodes. This will prove advantageous during rendering, see Section 5. As the set of stations $\mathcal{S}$ is only a subset of V in our model (Section 1.1), we can still avoid line crossings in them.

3.1 Baseline ILP

For each edge e, there are $|L(e)|!$ many orderings, and therefore the total number of combinations for the whole graph is immense. We formulate an ILP to find an optimal solution. We first define a baseline ILP, which explicitly considers line crossings and has $O(|E|M^2)$ variables and $O(|E|M^6)$ constraints. We then define an improved ILP with only $O(|E|M^2)$ constraints and which also considers line separations (MLNCM-S).

For every edge $e \in E$, we define $|L(e)|^2$ decision variables $x_{elp} \in \{0, 1\}$, where e indicates the edge, $l \in L(e)$ indicates the line, and $p = 1, \ldots, |L(e)|$ indicates the position of the line in the edge. We want to enforce $x_{elp} = 1$ when line l is assigned to position p, and 0 otherwise. This can be realized with the following constraints:

$$\forall l \in L(e) : \sum_{p=1}^{|L(e)|} x_{elp} = 1. \tag{3}$$

To ensure that exactly one line is assigned to each position, we need the following additional constraints:

$$\forall p \in \{1, \ldots, |L(e)|\} : \sum_{l \in L(e)} x_{elp} = 1. \tag{4}$$

Let A, B be two lines belonging to an edge $e = \{v, w\}$ and both extend over w. We distinguish two cases: Either A and B continue along the same adjacent edge e' (Figure 3 (left)), or they continue along different edges e' and e'' (Figure 3 (right)).

In the first case, A and B induce a crossing if the position of A is smaller than the position of B in $L(e)$, so $p_e(A) < p_e(B)$ (where $p_e(l)$ denotes the position of a line $l \in L(e)$ in e) but vice versa in $L(e')$. We introduce the decision variable $x_{ee'AB} \in \{0, 1\}$, which should be 1 in case a crossing is induced and 0 otherwise. To enforce this, we create one constraint per possible crossing. For example, a crossing would occur if we have $p_e(A) = 1$ and $p_e(B) = 2$ as well as $p_{e'}(A) = 2$ and $p_{e'}(B) = 1$. We encode this as follows:

$$x_{eA1} + x_{eB2} + x_{e'A2} + x_{e'B1} - x_{ee'AB} \leq 3. \tag{5}$$

In case the crossing occurs, the first four variables are all set to 1. Hence their sum is 4 and the only way to fulfill the ≤ 3 constraint is to set $x_{ee'AB}$ to 1. In the example given in Figure 3, six such

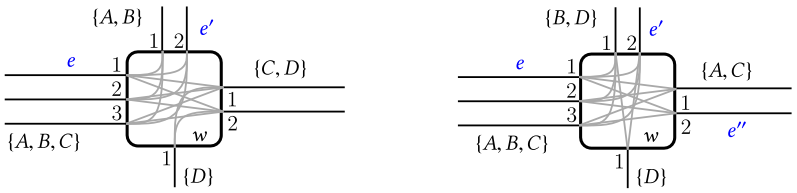

Fig. 3. Example instances. Both station polygons have four *node fronts*, each corresponding to an incident edge. Each node front has exactly one port $(1, 2, \ldots)$ for each line traversing through its edge. Gray lines depict possible inner node connections. Left: A, B extend from e to e' over w and may introduce a crossing, if the position of A is smaller than the position of B in e, but not in e' (or vice versa). Right: A crossing between A and B only depends on the line ordering in e, but not on the orderings in e' and e''.

constraints are necessary to account for all possible crossings of the lines A and B at node w. The objective function of the ILP then minimizes the sum over all variables $x_{ee'AB}$.

In the second case, the actual positions of A and B in e' and e'' do not matter, but just the order of e' and e''. We introduce a split crossing decision variable $x_{ee'e''AB} \in \{0, 1\}$ and constraints of the form $x_{eAi} + x_{eBj} - x_{ee'e''AB} \leq 1$ for all orders of A and B at e with $i < j$ as in that case a crossing would occur. We add $x_{ee'e''AB}$ to the objective function.

For mapping lines to positions at each edge, we need at most $|E|M^2$ variables and $2|E|M$ constraints, where $M = \max_{e \in E} |L(e)|$ (the maximum number of lines per segment). To minimize crossings, we have to consider at most M^2 pairs of lines per edge and introduce a decision variable for each such pair. That makes at most $|E|M^2$ additional variables, which all appear in the objective function. Most constraints are introduced when two lines continue over a node in the same direction. In that case, we create no more than $\binom{M}{2}^2 < M^4$ constraints per line pair per edge, so at most $|E|M^6$ in total. In summary, we have $O(|E|M^2)$ variables and $O(|E|M^6)$ constraints.

3.2 Improved ILP Formulation

The $O(|E|M^2)$ variables in the baseline ILP seem to be reasonable, as indeed $\Omega(|E|M^2)$ crossings could occur. But the $O(|E|M^6)$ constraints are due to enumerating all possible position inversions explicitly. If we could check the statement *position of A on e is smaller than the position of B* efficiently, then the number of constraints could be reduced. To have such an oracle, we first modify the line-position assignment constraints.

3.2.1 Alternative Line-Position Assignment. Instead of a decision variable encoding the exact position of a line, we now use $x_{el \leq p} \in \{0, 1\}$, which is 1 if the position of l in e is $\leq p$ and 0 otherwise. To enforce a unique position, we use the constraints:

$$\forall l \in L(e)\ \forall p \in \{1, \ldots, |L(e)| - 1\} : \quad x_{el \leq p} \leq x_{el \leq p+1}. \tag{6}$$

This ensures that the sequence can only switch from 0 to 1, exactly once. To make sure that at some point a 1 appears and that each position is occupied by exactly one line, we additionally introduce the following constraints:

$$\forall p \in \{1, \ldots, |L(e)|\} : \quad \sum_{l \in L(e)} x_{el \leq p} = p. \tag{7}$$

So for exactly one line l, $x_{el \leq 1} = 1$, for exactly two lines l' and l'', $x_{el' \leq 2} = x_{el'' \leq 2} = 1$ (where for one $l \in \{l', l''\}$, $x_{el \leq 1} = 1$) and so on.

3.2.2 Crossing Oracle. We reconsider the example in Figure 3 (left). Before, we enumerated all possible positions that induce a crossing for A, B at the transition from e to e'. But it would be

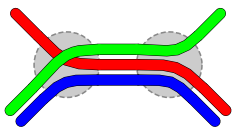

Fig. 4. Minimized crossings in the left example, but the right example better indicates line pairings.

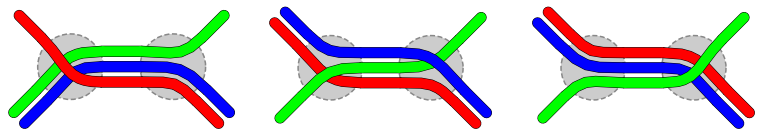

Fig. 5. Both orderings have two crossings, but in the right example they are done in one pass.

sufficient to have variables that tell us whether the position of A is smaller than the position of B in e, and the same for e', and then compare those variables. For a line pair (A, B) on edge e we call the respective variables $x_{eB<A}, x_{eA<B} \in \{0, 1\}$. To get the desired value assignments, we add the following constraints:

$$\sum_{p=1}^{|L(e)|} x_{eA\leq p} - \sum_{p} x_{eB\leq p} + x_{eB<A}M \geq 0, \tag{8}$$

$$x_{eB<A} + x_{eA<B} = 1. \tag{9}$$

The equality constraints make sure that not both $x_{eA<B}$ and $x_{eB<A}$ can be 1. If the position of A is smaller than the position of B, then more of the variables corresponding to A are 1, and hence the sum for A is higher. So if we subtract the sum for B from the sum for A and the result is ≥ 0, then we know the position of A is smaller and $x_{eB<A}$ can be 0. Otherwise, the difference is negative, and we need to set $x_{eB<A}$ to 1 to fulfill the inequality. It is then indeed fulfilled for sure as the position gap can never exceed the number of lines per edge.

To decide if there is a crossing, we would again like to have a decision variable $x_{ee'AB} \in \{0, 1\}$, which is 1 in case of a crossing and 0 otherwise. The constraint

$$|x_{eA<B} - x_{e'A<B}| - x_{ee'AB} \leq 0 \tag{10}$$

realizes this, as either $x_{eA<B} = x_{e'A<B}$ (both 0 or both 1) and then $x_{ee'AB}$ can be 0, or they are not equal and hence the absolute value of their difference is 1, enforcing $x_{ee'AB} = 1$. As absolute value computation cannot be part of an ILP we use the following equivalent standard replacement:

$$x_{eA<B} - x_{e'A<B} - x_{ee'AB} \leq 0, \tag{11}$$

$$-x_{eA<B} + x_{e'A<B} - x_{ee'AB} \leq 0. \tag{12}$$

If the values are equal, then nothing changes in the argumentation. If the values are unequal, then either Equation (11) or Equation (12) will produce a 1 as the sum of the first two terms, enforcing $x_{ee'AB} = 1$ as desired.

3.2.3 Complexity of the Improved ILP. For the line-position assignment, we need at most $|E|M^2$ variables and constraints just like before. For counting the crossings, we need a constant number of new variables and constraints per pair of lines per edge. Hence the total number of variables and constraints in the improved ILP is $O(|E|M^2)$.

3.3 Preventing Line Partner Separation

So far, we have only considered the number of crossings. Another relevant criterion for esthetic appeal is that "partnering" lines are drawn side by side. Figure 4 and Figure 5 provide two examples. We address this by punishing line separations and call this extension to our original MLNCM problem MLNCM-S. For two adjacent edges e and e' and a line pair (A, B) that continues from e to e', if A and B are placed alongside in e but not in e', we want to add a penalty to the objective function. For this, we add a variable $x_{eA\|B} \in \{0, 1\}$, which should be 0 if $|p_e(A) - p_e(B)| = 1$ (if they

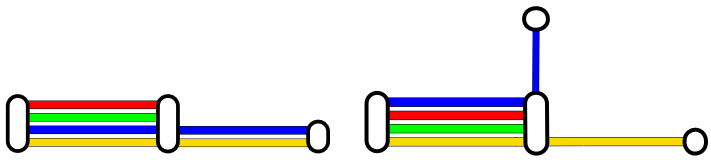

Fig. 6. Left: Periphery condition enforced by separation penalty. Right: Periphery condition not enforced by separation penalty.

are partners in e) and 1 otherwise. As $x_{eA\|B} = x_{eB\|A}$, we define a set $U(e)$ of unique line pairs such that $(l, l') \in U(e) \Rightarrow (l', l) \notin U(e)$. We add the following constraints per line pair (A, B) in $U(e)$:

$$\sum_{p=1}^{|L(e)|} x_{eA\leq p} - \sum_{p} x_{eB\leq p} - x_{eA\|B} M \leq 1 \tag{13}$$

$$\sum_{p=1}^{|L(e)|} x_{eB\leq p} - \sum_{p} x_{eA\leq p} - x_{eA\|B} M \leq 1. \tag{14}$$

If $|p_e(A) - p_e(B)| = 1$, then the sum difference is ≤ 1 and $x_{eA\|B}$ can be 0. If $|p_e(A) - p_e(B)| > 1$, then either Equation (13) or Equation (14) enforce $x_{eA\|B} = 1$. To prevent the trivial solution where $x_{eA\|B}$ is always 1, we add the following constraint per edge e:

$$\sum_{(l,l')\in U(e)} x_{el\|l'} \leq \binom{|L(e)|}{2} - |L(e)| - 1, \tag{15}$$

as there are $\binom{|L(e)|}{2}$ line pairs $(l, l') \in U(e)$ of which $|L(e)| - 1$ are next to each other.

Like in Section 3.2, we add a decision variable $x_{ee'A\|B}$ to the objective function that should be 1 if A and B are separated between e and e' and 0 otherwise:

$$x_{eA\|B} - x_{e'A\|B} - x_{ee'A\|B} \leq 0 \tag{16}$$

$$-x_{eA\|B} + x_{e'A\|B} - x_{ee'A\|B} \leq 0. \tag{17}$$

As we only add 1 constraint per edge and a constant number of constraints and variables per line pair in each edge, the total number of variables and constraints remains $O(|E|M^2)$.

3.3.1 Periphery Condition. Interestingly, punishing line separations also addresses a special case of the periphery condition introduced in Reference [5]. In general, this condition holds if lines ending in a station are always drawn at the left- or rightmost position in each incident edge. For nodes with degree ≤ 2, the periphery condition is enforced in MLNCM-S (Figure 6 (left)). For other nodes, however, it is not guaranteed (Figure 6 (right)).

3.4 Placement of Crossings or Separations

The placement of crossings or separations may be fine-tuned by adding node-based weighting factors $w_\times(v)$ (for crossings) and $w_\|(v)$ (for separations) to the objective function to prefer nodes or to break ties. For example, $w_\times(v)$ may depend on the node degree.

As described above, we especially want to prevent crossings or separations in station nodes. This can be achieved by adding constant global weighting factors $w_{\mathcal{S}\times}$ and $w_{\mathcal{S}\|}$ to each $x_{ee'll'}$ and $x_{ee'l\|l'}$ in the objective function if l and l' continue over a node $v_s \in \mathcal{S}$. These factors have to be chosen high enough so that a crossing or separation in any other node $v \notin \mathcal{S}$ is never more expensive than in v_s. As all $w_\times(v)$ and $w_\|(v)$ appear as coefficients in the objective function, they

have to be invariant to the actual line orderings. We can thus determine the maximum possible costs $\hat{w}_\times$ and $\hat{w}_\parallel$ prior to optimization and choose $w_{S\times} = \hat{w}_\times$ and $w_{S\parallel} = \hat{w}_\parallel$.

4 LINE GRAPH SIMPLIFICATION

It is possible to further simplify the optimization problem. In this section, we describe a set of transformations that may be applied to the line graph without affecting the global optimality of the line ordering. We call this simplified version of the line graph the optimization graph. In our experiments, these transformations reduced the number of constraints in the resulting ILPs by a factor between 2 and 6, and also enabled us to cut the optimization problem into smaller subproblems, resulting in even smaller individual ILPs. Line graph simplification led to significantly lower solution times (see Section 6). We first prove Lemmas 4.1—4.4 and use them to derive a set of pruning (Section 4.1), cutting (Section 4.2), and untangling rules (Section 4.3).

Lemma 4.1. *If for some set $\mathcal{B} = \{A, B, C, \ldots\} \subseteq \mathcal{L}$ it holds for all $l \in \mathcal{B}, e \in E : l \in L(e) \Rightarrow \mathcal{B} \subseteq L(e)$ (that is, all $l \in \mathcal{B}$ always appear together), then for each order $l_1, l_2, \ldots, l_{|\mathcal{B}|}$ on the $l \in \mathcal{B}$ a globally optimal line ordering solution exists in which for any $e \in E$ with $\mathcal{B} \subseteq L(e)$ and $i > 1$ it holds that $p_e(l_i) = p_e(l_{i-1}) + 1$ (in each edge in which they appear, all $l \in \mathcal{B}$ are positioned next to each other with a fixed global ordering).*

Proof. Let $L \in \mathcal{B}$ be the line in $\mathcal{B}$ that induces the minimal number of crossings and separations for some solution σ. We can move any line $L' \in \mathcal{B}, L' \neq L$ to the right of L (such that $p_e(L') = p_e(L) + 1$ for all e with $\mathcal{B} \subseteq L(e)$) without negatively affecting global optimality: Because $p_e(L') - p_e(L) = 1$ for each e on which they appear, and because all $l \in \mathcal{B}$ take the exact same path through the network, this placement will not induce any additional crossings or separations between L and L'. For the same reason, the number of crossings L' induces at the new position will be equivalent the to number of crossings L induces. The number of separations L' induces at the new position will be 0, as all separations that may be induced by L' at the new position were already induced by L. This new solution σ' will therefore always be better than or equal to σ. The same argument holds for moving any $L' \in \mathcal{B}, L' \neq L$ to the *left* of L. Since we can thus place any $l \in \mathcal{B}, l \neq L$ either to the left or to the right of L, and as we can freely choose the order in which we move them, a globally optimal solution like described in Lemma 4.1 can be constructed for each order on the $l \in \mathcal{B}$. □

Lemma 4.2. *Given a solution σ with an optimal ordering for each $L(e)$. We say a node v belongs to W if $deg(v) = 2$ and for its adjacent edges e and e' the set of lines $L(e)$ is equal to $L(e')$. A crossing or a separation in some $v \in W$ can always be moved from v to a node $v' \notin W$ without negatively affecting optimality.*

Proof. We set $L^* = L(e) = L(e')$ and first consider crossings. There are two possible cases: (1) All $l \in L^*$ always occur together in each edge. Then Lemma 4.1 holds, and there is an optimal solution in which the ordering of $L(e)$ is the same as of $L(e')$, inhibiting any crossings in v. We can thus ignore this case. (2) The lines in L^* separate in some node $v' \neq v$. Then they either diverge into separate edges at v', or a subset of them ends in v'. If they diverge, then the degree of v' has to be at least 3, implicating $v' \notin W$. If some (not all) of them end in v', then v' has to be adjacent to at least 2 edges e, e' with $L(e) \neq L(e')$, again implicating $v' \notin W$. Therefore, such a v' will indeed always exist. Under a uniform crossing penalty, we can trivially move the crossing from v to v' without affecting optimality. Under the penalty described in Section 3.4, optimality will also not be affected negatively, because $\deg(v)$ is always 2, implying that v is a station (Section 2). The same argument holds for line separations. □

Lemma 4.3. *If for some edge e all $l \in L(e)$ end (or start) in a node $v \in e$ or $|L(e)| = 1$, then the ordering of $L(e)$ will not affect the number of crossings or separations in v. Figure 7 gives an example.*

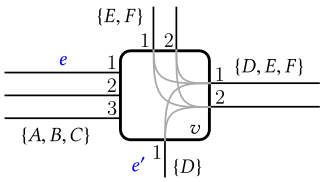

Fig. 7. Illustration of Lemma 4.3. The ordering of the lines on e or e' will have no impact on the number of crossings or separations in v, as v is a terminus node for all lines $L(e) = \{A, B, C\}$ and the number of lines on e' is 1.

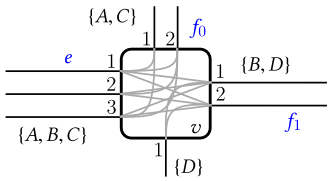

Fig. 8. Illustration of Lemma 4.4. The clockwise edge positions (starting at e) $\pi_e^v(f_0) = 0$ and $\pi_e^v(f_1) = 1$ induce a partial ordering on $L(e)$, which will not cause any crossing or separation between A, B or C, B in v, namely that $p_e(A) < p_e(B)$ and $p_e(C) < p_e(B)$.

Proof. In the first case, no $l \in L(e)$ extends over v, so they cannot induce any crossing or separation. In the second case, all orderings of $L(e)$ are equivalent (there is only one). □

Lemma 4.4. *For some edge $e = \{u, v\}$ with lines $L(e)$, if v has n additional adjacent edges $f_i \neq e$ with lines $L(f_i)$ such that $L(e) \subseteq \bigcup_{i=0}^{n-1} L(f_i)$ and $\forall i < n : L(e) \cap L(f_i) \neq \emptyset$ (the lines on e branch into n edges), and $\forall l \in L(e), l \in L(f_i) : \nexists f_j \neq f_i : l \in L(f_j)$ (each line on e only extends over v into a single edge), then a partial ordering on $L(e)$ based on the clockwise enumeration of the f_i will not induce any crossing or separation at v between two lines $l, l' \in L(e)$, which continue over v along two different edges. Figure 8 gives an example.*

Proof. We say that $\pi_e^v(f)$ is the clockwise position of edge f at node v, beginning at edge e. As previously defined, the position of some line $l \in L(e)$ on e is denoted by $p_e(l)$.

Without loss of generality, we consider lines A and B in the example given in Figure 8. A and B induce a crossing at v if $p_e(A) < p_e(B)$ and $\pi_e^v(f_0) > \pi_e^v(f_1)$ or vice versa. As A and B extend over v into different edges, no separation between them may ever occur in v. The π_e^v therefore induce a partial ordering of $L(e)$, which does not cause any crossings or separations between any such line pair $l, l' \in L(e)$ in v. □

4.1 Pruning Rules

Using Lemmas 4.1–4.3, we may simplify the input line graph with a set of pruning rules described in this section. Each pruning rule is followed by a correctness proof showing that its application does not negatively affect the optimality of the line ordering.

As later rules will split line graph nodes, we first define v^* to be the original line graph node we constructed v from. (Note that for most nodes that were not eligible for any such rule, $v^* = v$). Crossing and separation penalties that are computed after the graph has been simplified are always based on v^*.

Pruning rule 1 (*Node Contraction*): *Delete each node v with degree 2 and adjacent edges $e = \{u, v\}$, $e' = \{v, w\}$, where $L(e) = L(e')$ (the lines in both edges are the same). If $\deg(v^*) \neq 2$ and $|L(e)| = |L(e')| > 1$, then additionally check if $w_\times(v^*) \geq w_\times(u^*)$ and $w_\parallel(v^*) \geq w_\parallel(u^*)$ (crossings and separations in v can be moved to u at equal or lower cost) or $w_\times(v^*) \geq w_\times(w^*)$ and $w_\parallel(v^*) \geq w_\parallel(w^*)$ (crossings and separations in v can be moved to w at equal or lower cost). If that is the case, then combine the adjacent edges $e = \{u, v\}$, $e' = \{v, w\}$ into a single new edge $ee' = \{u, w\}$ with $L(ee') = L(e) = L(e')$ (Figure 9).*

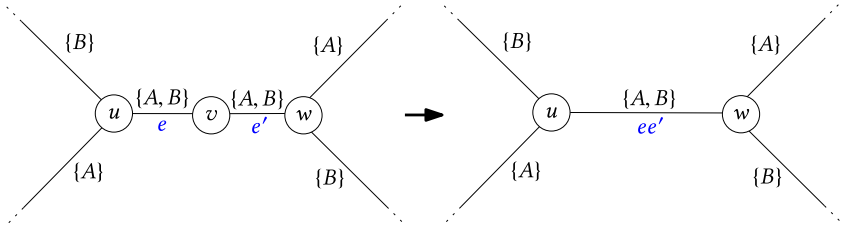

Fig. 9. Illustration of Pruning Rule 1 (Node Contraction). Left: e and e' share the same set of lines and are connected by degree 2 node v. Right: Node v has been contracted without affecting line ordering optimality.

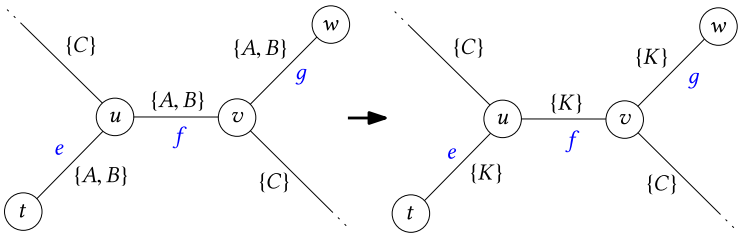

Fig. 10. Illustration of Pruning Rule 2 (Line Partner Collapsing). Left: A and B always occur together on edges e, f, and g. Right: A and B have been collapsed into K.

Correctness Proof. If $\deg(v^*) = 2$, then Lemma 4.2 holds for v, and we can be sure that a node v' exists were any crossing or separation that could have occurred in v can occur at lower or equal cost. If $\deg(v^*) > 2$, then we are only contracting v if we can find an adjacent node in which any crossing or separation that may have occurred in v can occur at equal or lower cost. □

Pruning rule 2 (*Line Partner Collapsing*): *Collapse each set of lines $l \in \mathcal{B}$, where all l always appear together in each edge through which they traverse into a single new line K (Figure 10). Weight crossings with K by the number of lines $|\mathcal{B}|$ it combines to avoid distorting penalties. Assign the $l \in \mathcal{B}$ a random relative ordering.*

Correctness Proof. Lemma 4.1 states that a globally optimal line ordering exists where all $l \in \mathcal{B}$ are positioned next to each other in each edge in which they appear. In this case, each $l \in \mathcal{B}$ will have the same number of crossings as any other $l' \in \mathcal{B}, l' \neq l$. The number of crossings all $l \in \mathcal{B}$ induce is thus the number of crossings any single line $L \in \mathcal{B}$ induces times $|\mathcal{B}|$. Additionally, per Lemma 4.1 a globally optimal solution exists for any relative ordering of $\mathcal{B}$, so a random ordering may be assigned to them. □

Pruning rule 3 (*Double Termini Pruning*): Remove each edge $e = \{u, v\}$ where u and v are termini for all $l \in L(e)$ (Figure 11).

Correctness Proof. According to Lemma 4.3, the ordering of $L(e)$ will not have any effect on the number of crossings in both u and v. We can thus ignore e during the optimization and give $L(e)$ a random ordering. □

We call the resulting graph the pruned graph of G. Figure 14 (top right) gives an example of a pruned graph after pruning rules 1–3 were applied.

4.2 Cutting Rules

The pruned graph may then be further broken down into ordering-relevant connected components using the cutting rules below. The components can then be optimized separately and in parallel

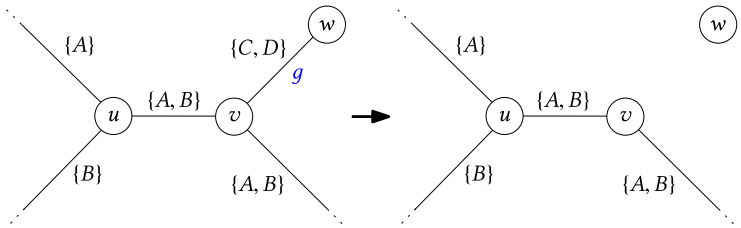

Fig. 11. Illustration of Pruning Rule 3 (Double Termini Pruning). Left: v and w are termini for all lines on $g = \{v, w\}$, their ordering is irrelevant for the number of crossings or separations in v or w. Right: g has been removed prior to optimization.

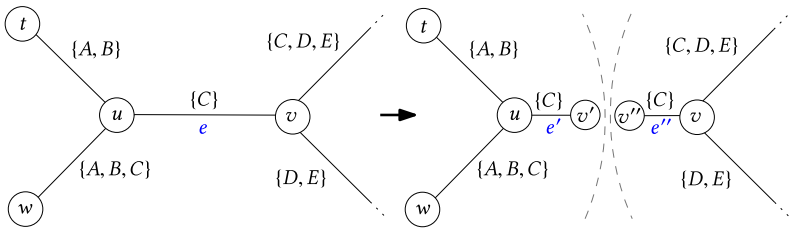

Fig. 12. Illustration of Cutting Rule 1 (single line cut). Left: Only line C continues over e, and therefore the ordering of $L(e)$ is irrelevant to the number of crossings in u and v. Right: e has been split into e' and e'', and the resulting graph components can be optimized separately.

(Figure 14 (bottom)). Just like for the pruning rules described in Section 4.1, each cutting rule is followed by a correctness proof showing that its application will not negatively affect optimality.

Cutting rule 1 (*Single Line Cut*): *Cut each edge $e = \{u, v\}$ with $|L(e)| = 1$ into two edges $e' = \{u, v'\}$ and $e'' = \{v'', v\}$ with $L(e') = L(e'') = L(e)$, where v' and v'' are new nodes (Figure 12).*

Correctness Proof. As $|L(e)| = 1$, Lemma 4.3 holds and the ordering of $L(e)$ does not affect the number of crossings or separations in u and v. As $|L(e')| = |L(e'')| = 1$, the orderings of the new split edges are still irrelevant for the number of crossings or separations in u and v. The degree of the newly inserted nodes v' and v'' is 1, so no crossing or separation can occur in them. Therefore, the overall number of crossings and separations will indeed stay the same. □

Cutting rule 2 (*Terminus Detachment*): *Replace each edge $e = \{u, v\}$ where v has a degree >1 and is the first or last stop (the* terminus*) for each $l \in L(e)$ with an edge $e' = \{u, v'\}$, where v' is a newly inserted node that is only connected to e'. The original node v is replaced by a new node v'' connected to the remaining adjacent edges of v (Figure 13).*

Correctness Proof. As per Lemma 4.3, the ordering of $L(e)$ does not affect the number of crossings or separations in v. We can thus detach e from v without affecting optimality. The degree of the newly inserted node v' is 1, so no crossing or separation can occur in it. □

Figure 14 (bottom) gives an example of a line graph after Cutting Rules 1 and 2 were applied.

4.3 Graph Untangling

While the pruning rules described so far are able to significantly reduce the complexity of the line-ordering optimization problem, they still miss some situations in which further simplification is possible. Likewise, the cutting rules described so far may miss some situations in which the problem may be further broken down into subproblems.

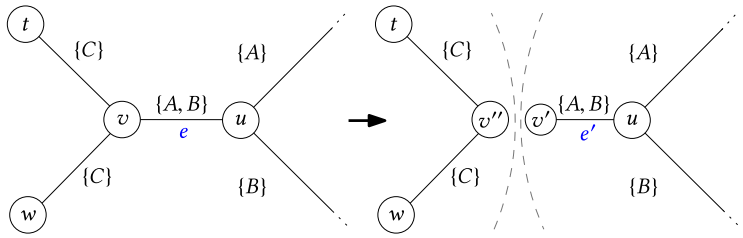

Fig. 13. Illustration of Cutting Rule 2 (terminus detachment). Left: v is a terminus node for both $A, B \in L(e)$, and therefore the ordering of $L(e)$ is irrelevant to the number of crossings in v. Right: e has been detached from v, and the resulting graph components can be optimized separately.

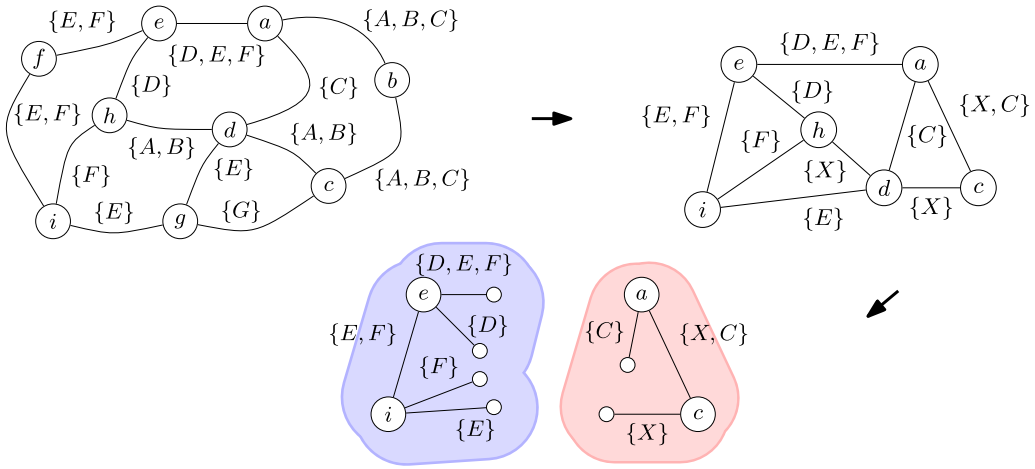

Fig. 14. Top left: Line graph with seven lines A, B, C, D, E, F, G. Top right: Line graph after pruning rules were applied. Note that b and f have been contracted by Pruning Rule 1 and lines A, B were collapsed into X. Bottom: Ordering-relevant connected components of G after applying cutting rules. In particular, edge $\{e, a\}$ has been detached from a by Cutting Rule 1, edges $\{e, h\}, \{d, h\}, \{i, h\}$ have been detached from h by Cutting Rule 1 and edges $\{i, d\}, \{h, d\}, \{a, d\}, \{c, d\}$ have been detached from d by Cutting Rule 1. The edge resulting from detaching $\{h, d\}$ from both h and d has been removed by Cutting Rule 2. The graph now consists of two components that can be optimized separately.

This section describes six line graph untangling rules addressing such situations. We do not claim that the list of possible untangling rules given in this section is complete. Again, like for the pruning rules described in Section 4.1 and the cutting rules described in Section 4.2, each untangling rule is followed by a correctness proof showing that its application will not negatively affect optimality. As we will see in Section 4.5, these rules (combined with the pruning and cutting rules from the previous sections) are able to completely solve the line ordering optimization problem for specific line graph instances. In some cases, they are able to reduce the search space to a size that can be explored by a simple exhaustive search.

4.3.1 Full X Structure. We now consider the situation in Figure 15 (left). It is easy to see that if in some node v we can identify a pair $\{e, f\}$ of adjacent edges with $L(e) = L(f)$, and if neither e nor f have a line that continues into another adjacent edge $g \neq e, g \neq f$, then the orderings in e and f

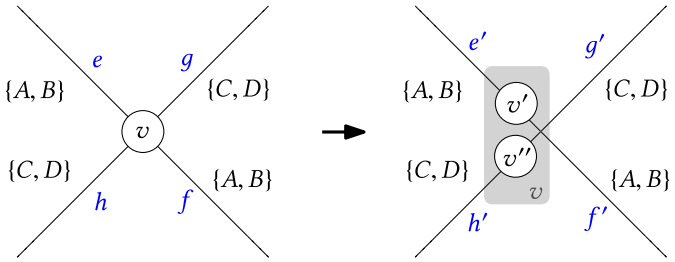

Fig. 15. Illustration of Untangling Rule 1 (Full X). Left: Full X structure in a line graph. $\{A, B\}$ continue from e to f through node v without interfering with $\{C, D\}$ on h and f. Right: Node v has been split into two nodes v' and v'', without affecting line ordering optimality.

cannot affect the number of crossing between any $l \in L(e)$ and any $l' \notin L(e)$, only the number of crossings between themselves. Using this, we can state the following untangling rule:

Untangling rule 1 (*Full X*): *For some node v with $\deg(v) > 2$ in the line graph and its adjacent edges $e_0, e_1, \ldots, e_{\deg(v)-1}$, if we can identify two edges $e_a = \{v, u\}$ and $e_b = \{v, w\}$ with $L(e_a) = L(e_b)$ and from both e_a and e_b the $L(e_a)$ do not (partially) continue into any other edge $e_i \notin \{e_a, e_b\}$, then split v into two newly inserted nodes v' and v''. Node v' gets connected to u with an edge e'_a and to w via e'_b. We set $L(e'_a) = L(e_a) = L(e'_b) = L(e_b)$. Node v'' gets connected to the remaining nodes u_i v was originally connected to via $e'_i = \{v'', u_i\}$. We set $L(e'_i) = L(e_i)$. Figure 15 (right) gives an example.*

Correctness Proof. Any crossing or separation between lines on e_a and lines on e_b that may have occurred in v can still occur in v', as e_a and e_b are still adjacent in v'. Any crossing or separation between lines on any edge $e_i \notin \{e_a, e_b\}$ that may have occurred in v can still occur in v'', as all $e_i \notin \{e_a, e_b\}$ are still adjacent in v''. Any crossing or separation between a line on some edge $e_i \notin \{e_a, e_b\}$ and a line on either e_a or e_b is prohibited. However, such a crossing or separation was already not possible before, as (by prerequisite of Untangling Rule 1) $\forall e_i \notin \{e_a, e_b\} : L(e_i) \cap L(e_a) = \emptyset$. □

Note that this rule alone will not have any effect on ILP sizes, as we would, for example, not add any constraints or variables for crossings between a line on edge h a line on edge e in Figure 15 (left), as described in Sections 3.1 and 3.2. However, v' and v'' may now be eligible for contraction according to Pruning Rule 1. Additionally, this rule may cause the optimization graph to break down into two connected components. An example of such a structure in the real-world map of the New York subway network is shown in Figure 22.

4.3.2 Y Structures. Figure 16 (left) gives an example of what we call a Y structure in the line graph. An edge $e = \{u, v\}$ branches at v into two edges f and g. As u is a terminus, the line ordering in e can always be adjusted to match the line orderings of the minor legs f and g with zero crossings. More specifically, the relative ordering of f and g at v already induces a partial optimal ordering of the lines on e, namely that $p_e(A) < p_e(C)$ and $p_e(B) < p_e(C)$, where $p_e(l)$ is the position of line $l \in L(e)$ on edge e like defined in Section 3.1. Any ordering violating these constraints would induce an unnecessary crossing (or separation) in v.

We again say that $\pi_e^v(f)$ is the clockwise position of edge f at node v, beginning at edge e. For example, in Figure 16 (left), $\pi_e^v(f) = 0$ and $\pi_e^v(g) = 1$.

Untangling rule 2 (*Full Y Structure*): *If for some node v we can identify an edge $e = \{v, u\}$ where u is a terminus and that completely branches at v into $\deg(v) - 1$ edges $e_0, e_1, \ldots, e_{\deg(v)-2}$ such that $L(e_0) \cup L(e_1) \cdots \cup L(e_{\deg(v)-2}) = L(e)$ and all $L(e_i)$ are pairwise disjoint, then split v and u into nodes v', v'' and u', u'' like shown in Figure 16 (right). We call e the* major *leg and $e_0, e_1, \ldots, e_{\deg(v)-2}$ the*

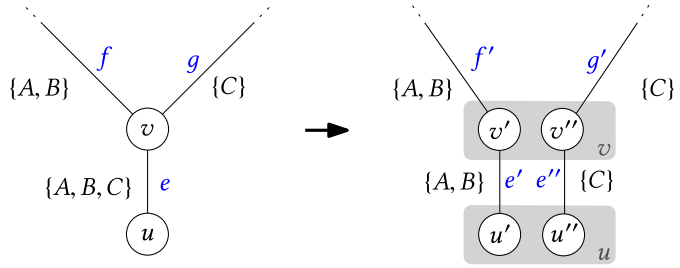

Fig. 16. Illustration of Untangling Rule 2 (Full Y). Left: Y structure in the line graph. The relative ordering of $\{A, B\}$ and $\{C\}$ in e is determined by the ordering of f and g at v. Right: v and u have been split to settle the relative ordering of $\{A, B\}$ and $\{C\}$. Note that if g was the segment with the largest number of lines in the graph, then we now have reduced M, the maximum number of lines per edge, from 3 to 2.

minor *legs. For simplicity, we assume that the minor leg edges are already sorted in ascending order by their* π_e^v *values. Nodes* v' *and* u' *are connected with an edge* e'*, where* $L(e') = L(e_0)$ *(the lines of the leftmost minor leg). Nodes* v'' *and* u'' *are connected with an edge* e''*, where* $L(e'') = \bigcup_{i=1}^{\deg(v)-2} L(e_i)$ *(the lines of the remaining minor legs to the right). Additionally,* v' *gets connected to the node that* v *was originally connected to via the first minor leg* e_0 *with a new edge* e'_0*, where* $L(e_0) = L(e'_0)$*. Similarly, all remaining branches are re-connected to* v'' *via edges* $e'_i, i > 0$*.*

Correctness Proof. As per Lemma 4.4, the implicit partitioning of $L(e)$ into $L(e')$ and $L(e'')$ will not induce any crossings or separations in v between two lines from different minor legs. As u is a terminus, Lemma 4.3 holds and the ordering of $L(e)$ is irrelevant to the number of crossings or separations in u. It remains to show that the splitting of u and v does not prohibit any previously possible crossings between lines $l \notin L(e)$: As all $L(e_i)$ are pairwise disjoint, no crossing or separation between them was previously possible at v. As u is a terminus node, no crossing or separation was previously possible at u. □

To be able to later deduce the ordering of the original major leg e, we additionally store an ordering of the new major leg edges e' and e'', which is just the original clockwise ordering between e_0 and the remainder of the minor legs. The ordering of the lines in the original edge e can then be constructed as follows: Take the ordered lines (after optimization) in e', and then take the ordered lines (after optimization) in e''.

Note that this rule only untangles the leftmost minor branch and that a repeated application is necessary to completely untangle more than two branches.

4.3.3 Partial Y Structures. A special case of Y structure can be seen in Figure 17 (left). The major leg e completely branches into two minor legs f and g at v. However, the lines of the minor legs are not completely contained in the major leg: Line D continues through node v from f to g but not to e. But the ordering of f and g still induces a partial ordering of e. We call this situation a *Partial* Y.

Untangling rule 3 (*Partial Y Structure*): *For some node* v*, if we can identify a major leg edge* $e = \{v, u\}$ *where* u *is a terminus and that completely branches at* v *into* n *minor leg edges* $e_0, e_1, \ldots, e_{n-1}$ *(note that this implicates* $n > 1$ *and* $\forall i < n : L(e) \cap L(e_i) \neq \emptyset$*) such that* $L(e) \subsetneq L(e_0) \cup L(e_1) \cdots \cup L(e_{n-1})$*, and if no line in* $L(e)$ *continues over* v *into two minor leg edges, then split* u *into nodes* u', u''*. For simplicity, we again assume that the minor leg edges are already sorted in ascending order by their* π_e^v *values. Similarly as in Untangling Rule 2 (Full Y),* v *and* u' *are connected with an edge* e'*, where* $L(e') = L(e_0)$*, and* v *and* u'' *are connected with an edge* e''*, where* $L(e'') = \bigcup_{i=1}^{n-1} L(e_i)$ *(Figure 17 (right)).*

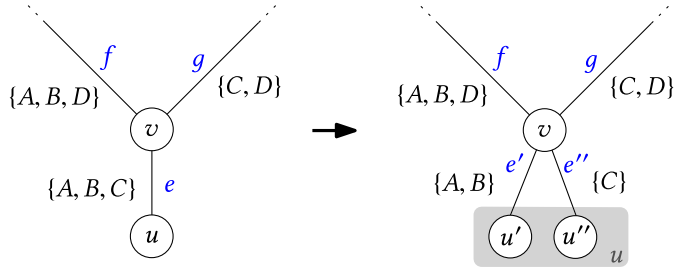

Fig. 17. Illustration of Untangling Rule 3 (Partial Y). Left: Partial Y structure. The relative ordering of $\{A, B\}$ and $\{C\}$ in e is again determined by the ordering of f and g at v, just like in a Full Y structure. However, we cannot split v, because an additional line D passes through it. Right: Only u has been split to settle the relative ordering of $\{A, B\}$ and $\{C\}$.

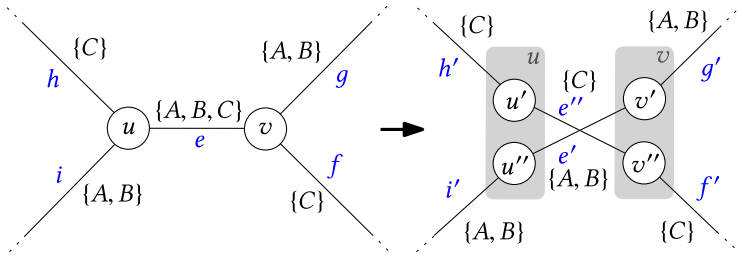

Fig. 18. Illustration of Untangling Rule 4 (Full Double Y). Left: Full Double Y structure. $\{A, B\}$ and $\{C\}$ join at node u, follow e together, and depart again at v in inverted directions. Crossings in either u or v between $\{A, B\}$ and $\{C\}$ are unavoidable. Right: u and v have been split up to settle the relative ordering of $\{A, B\}$ and $\{C\}$ in e.

Correctness Proof. The correctness proof is analogous to that of Untangling Rule 2, but as v is not split, we do not have to show that no crossings or separations at v between any two lines on two different minor legs are prohibited. □

Just like with Full Y structures, we store the order of e' and e'' to be able to later deduce the line ordering in the original major leg e.

Note that Untangling Rule 2 (Full Y) is equivalent to an application of Untangling Rule 3 (Partial Y), followed by Untangling Rule 1 (Full X).

4.3.4 Double Y Structures. A more complex structure that is commonly found in real-world input data is depicted in Figure 18 (left). Two line threads (in the example, $\{C\}$ and $\{A, B\}$) on two edges h and i join at some node u, continue together for a single segment e, and branch again at some node v into g and f. We call situations like these *Double Y* structures.

In Figure 18, it is easy to see that there is no reason for $\{C\}$ and $\{A, B\}$ to be intertwined in e, for example by setting the ordering on e to (A, C, B): This would only induce an unnecessary splitting between A and B and would not be optimal, regardless of how the rest of the line graph looks like. Additionally, it is also easy to see that the ordering of the two line threads at u and v imposes a lower bound on the sum of crossings in u and v: In Figure 18 (left), because $\pi^u_e(i) < \pi^u_e(h)$ and $\pi^v_e(g) < \pi^v_e(f)$, two crossings between C and A, as well as C and B in either u or v are unavoidable, regardless of the actual orderings in h, i, g, and f. In Figure 19 (left), no crossing is necessary at all—the two line threads can always continue through u and v next to each other, regardless of their internal ordering.

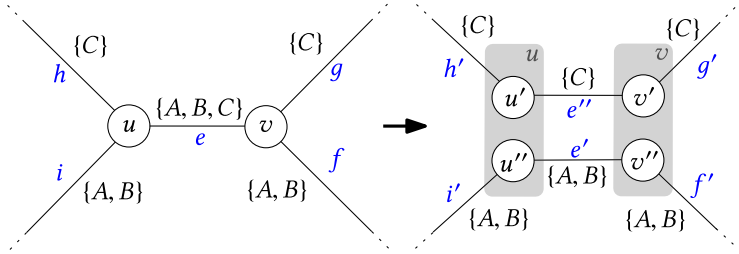

Fig. 19. Left: Full Double Y structure like in Figure 18, but a crossing between $\{A, B\}$ and $\{C\}$ is not necessary. Right: u and v have been split up to settle the relative ordering of $\{A, B\}$ and $\{C\}$ in e.

We transform structures like this with the following rule:

Untangling rule 4 (*Double Y Structure*): *If some major leg edge $e = \{u, v\}$, with $\deg(u) = \deg(v) \geq 3$ branches at u into $n = \deg(u) - 1$ left minor leg edges $e_0^u, e_1^u, \ldots, e_{n-1}^u$ and at v into n right minor leg edges $e_0^v, e_1^v, \ldots, e_{n-1}^v$ and if there is a bijection $a(i) \mapsto j$ with $i, j \in [0, n-1]$ such that $L(e_i^u) = L(e_{a(i)}^v)$ (e branches into the exact same left and right branches), then we say e is a Full Double Y structure. We additionally require that $\bigcup_{i=0}^{n-1} L(e_i^u) = \bigcup_{j=0}^{n-1} L(e_j^v) = L(e)$ (the combined lines of all left and of all right minor legs are equal to the lines contained in the major leg) and that for both the right and left minor legs, $L(e_i^u)$ and $L(e_j^v)$ are pairwise disjoint (there are no lines continuing through u and v to any other edge than e). We untangle both u and v like in Untangling Rule 2 (but u' and u'' are now also connected to the original left minor leg edges adjacent to u) and split e into two edges e' and e'', where e' now holds the lines of the first left minor leg and e'' the lines of the remaining minor legs. Figure 18 (right) gives an example.*

Just like with Untangling Rules 2 (Full Y) and 3 (Partial Y), we have to store an ordering of e' and e'' to later deduce the line ordering in the original line graph edge e. However, there are now 2 possible orderings we could store: We can either base the ordering of e' and e'' on the ordering position of the left minor leg e_0^u at u or on the ordering position of the right minor leg e_0^v at v. If the two orderings are inverse, that is, if for all $1 \leq i < n$ it holds that $\pi_e^u(e_0) < \pi_e^u(e_i) \Rightarrow \pi_e^v(e_{m(0)}) > \pi_e^v(e_{m(i)})$, then it does not matter, because there are no unavoidable crossings we have to consider (Figure 19 (right)). If that is not the case, and an unavoidable crossing occurs (Figure 18 (right)), then we have to base the ordering on the node with smaller crossing penalty to not compromise optimality. For example, in Figure 18 (right) $\pi_e^u(h) > \pi_e^u(i)$. Assume that $w_\times(u) > w_\times(v)$. Then we set the ordering of e' and e'' in the original line graph edge e to (e'', e'), making sure that that after optimization, the ordering in e is either (C, A, B) or (C, B, A), depending on the final ordering of A and B. The unavoidable crossings between threads $\{A, B\}$ and $\{C\}$ would then occur in v. If $w_\times(u) < w_\times(v)$, then the ordering would be set to (e', e''), and the crossings would appear in u.

Correctness Proof. We again prove that the implicit partitioning of the major leg lines $L(e)$ does not affect optimality. We assume without loss of generality that node u has an equal or higher crossing penalty than v and base the partial ordering of $L(e)$ on the left minor leg positions. We consider two cases: (1) the ordering of the left minor legs is exactly inverse to the ordering of their right minor leg counterparts specified by m (Figure 19 (left)) and (2) the ordering of the left minor legs is not inverse to their right leg counterparts (Figure 18 (left)). Per Lemma 4.4, the partitioning will not induce any crossings or separations between two lines from different left minor legs in u. In case (1), because of the symmetry, it will also not induce any crossing or separations between two lines from different right minor legs in v. For case (2), the partial ordering of $L(e)$ will induce inter-partition crossings at v, but these crossings are caused by the inversions of the left minor and right minor edges and are unavoidable. Moving them to v is optimal, as we assumed that $w_\times(u) \geq w_\times(v)$.

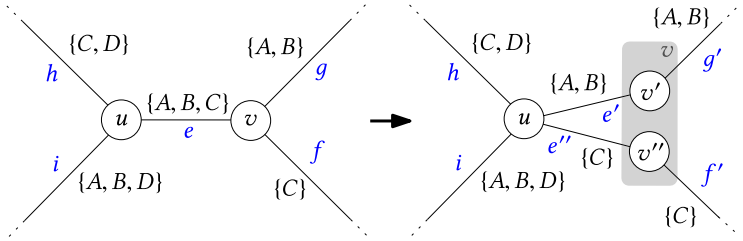

Fig. 20. Illustration of Untangling Rule 5 (Partial Double Y). Left: Partial Double Y structure. The relative ordering of $\{A, B\}$ and $\{C\}$ in e can be determined from the ordering of h, i, g, and f at u and v, but we cannot split u, because D passes through it. Right: Only v has been split up, settling the relative ordering of $\{A, B\}$ and $\{C\}$ in e.

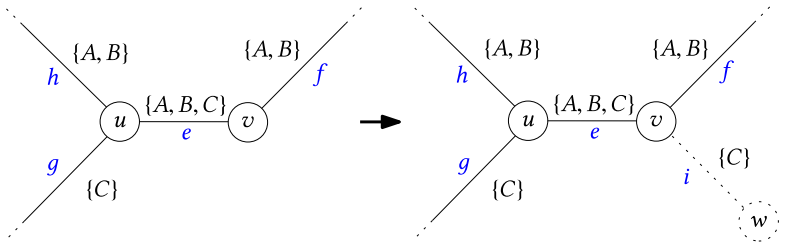

Fig. 21. Illustration of Untangling Rule 6 (Stump). Left: Stump structure. C ends in v and its optimal position in e is determined by the position of g at u. Right: Stump structure transformed into a full double Y structure, which can then be untangled by applying Untangling Rule 4 (Double Y structure).

As all $L(e_i^u)$ (and all $L(e_j^v)$) are pairwise disjoint, no crossing or separation between any minor legs was possible before at u or v, and thus no optimal crossing or separation is prohibited by splitting them. □

4.3.5 Partial Double Y Structures. Just as with Y structures, there may also be partial Double Y structures (Figure 20 (left)). These are Double Y structures where *one* of the nodes u and v fulfills the criteria described in Section 4.3.4, and the other node fulfills the criteria described for v in Partial Y structures (Section 4.3.3, that is, the major leg branches at v into the same minor legs as at u, but v may have additional edges or lines on the minor legs that are not contained in the major leg).

Untangling rule 5 (*Partial Double Y Structure*): *In cases like the one described above, we only split the node fulfilling the criteria described in Section 4.3.4, just like we broke up only u in Section 4.3.3 (Figure 20 (right)).*

Correctness Proof. The correctness proof is analogous to that of Untangling Rule 4, but as v is not split, we do not have to show that no crossing or separation at v between any two lines on two different right minor legs is prohibited. □

4.3.6 Stump Structures. An additional class of structures, which we call *stumps*, can be transformed into a Double Y structure without affecting optimality and with minimal changes to the graph. Figure 21 (left) gives an example. Line C on minor leg g attaches itself to the major leg e in node u but already terminates in node v. Regardless of the ordering of A and B in h, e, and f, the obvious optimal ordering in e is either (A, B, C) or (B, A, C), because we can always put C at the "bottom" of e without introducing a line crossing or a line separation.

Formally, if some major leg edge $e = \{u, v\}$ with $\deg(u) \geq 3$ branches at u into $n = \deg(u) - 1$ left minor leg edges $e_0^u, e_1^u, \ldots, e_{n-1}^u$ and at v into $m = \deg(v) - 1 < n$ right minor edges

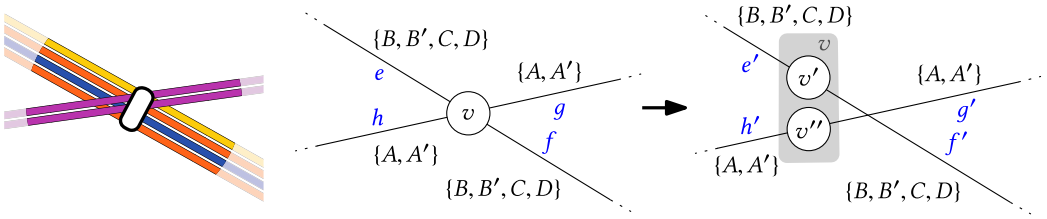

Fig. 22. Left: Excerpt from the New York subway system map. Middle: Line graph corresponding to the highlighted area of the map. Right: Optimization graph (exact edge geometries have been omitted) after applying Untangling Rule 1 (Full X) to v. Note that v was split into two nodes v' and v''.

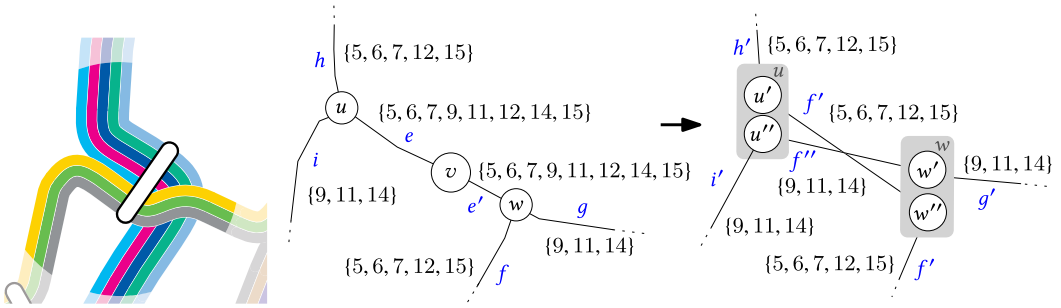

Fig. 23. Left: Excerpt from the Stuttgart light rail system map. Middle: Line graph corresponding to the highlighted area of the map. Right: Optimization graph (exact edge geometries have been omitted) after applying Pruning Rule 1 (Node Contraction) to v and Untangling Rule 3 (Full Double Y) to the resulting new edge. Note that e and e' were first merged into a new edge f by Pruning Rule 1, which was then split into f' and f'' by Untangling Rule 3. The number of possible line ordering solutions for the central segment consisting of e and e' was reduced from $|L(e)|! \times |L(e')|! = 8! \times 8! = 1.63 \times 10^9$ to $|L(f')|! \times |L(f'')|! = 5! \times 3! = 720$.

$e_0^v, e_1^v, \ldots, e_{m-1}^v$, if all $L(e_i^u)$ are pairwise disjoint and if there is an injection $a'(j) \mapsto i$ with $j, i \in [0, m-1]$ such that $L(e_j^v) = L(e_{a'(j)}^u)$ (for each right minor leg, there is an equivalent left minor leg at u, but there may be left minor legs continuing over u that end in v), and if for all $j, j' < m$ it holds that $\pi_e^v(e_j) < \pi_e^v(e_{j'}) \Rightarrow \pi_e^u(e_{a'(j)}) > \pi_e^u(e_{a'(j')})$ (the relative ordering of the left minor legs is inverse to the relative ordering of the right minor legs), then we say the left minor legs without a corresponding right minor leg are *stumps* in e.

Untangling rule 6 (*Stump Structure*): *We transform a situation like described above into a Full Double Y structure by introducing $n - m$ dummy edges g' for each stump g with with $L(g') = L(g)$ from v to an additional dummy node w in such a way that for each stump g, its dummy counterpart g' will be at a position in v that is inverse to the position of g in u. Figure 21 (right) gives an example. By applying Untangling Rule 4 (Double Y) afterward, we detach the stump line from the major leg.*

Correctness Proof. If the right minor legs are extended in such a way that they exactly mirror the left minor legs, then both sides induce an equivalent partial ordering of $L(e)$ without any crossings or separations between two lines a, b from different left minor legs, as per Lemma 4.4. Any crossing or separation between two lines a, b on the same minor leg (including the original stump edges) can still occur. Optimality is thus not negatively affected. □

Figure 22 and Figure 23 give two examples in real-word transit networks where graph untangling can be applied.

4.4 Complexity of Line Graph Simplification

The real power of the untangling rules described in the previous section lies in their repeated application, together with the pruning and cutting rules described in Sections 4.1 and 4.2. But how long does it take until an input line graph is fully untangled, pruned, and cut into connected components?

We first examine the complexity of applying a single pruning, cutting, or untangling rule. Afterward, we analyze the complexity of iteratively applying each of those rules until none of them may be applied anymore. We assume that the lines in each $L(e)$ are ordered (for example, by some internal line id) and that the adjacency list of each node is ordered in clockwise fashion by the outgoing edge angle. The maximum node degree of our input line graph G is denoted as $D = \max_{v \in V} \deg(v)$. By $M = \max_{e \in E} |L(e)|$, we denote the maximum number of lines per segment. We can safely remove nodes with degree 0 from the line graph prior to optimization and thus assume that $2|E|$ is an upper bound for $|V|$.

We call the application of a single rule on all edges of a graph a *round* of this rule.

4.4.1 Complexity of Pruning Rules. We first consider the complexity of the pruning rules. For each node contraction according to Pruning Rule 1, we have to check whether $L(e) = L(e')$, which can be done in $O(M)$. A single round of Pruning Rule 1 can therefore be applied in $O(|E|M)$, as we have to contract at most $2|E|$ nodes.

For Pruning Rule 2 (Line Partner Collapsing), we have to do at most $\binom{M}{2} < M^2$ depth-first searches in the line graph. A single round of Pruning Rule 2 can therefore be applied in $O(|E|M^2)$.

Pruning Rule 3 (Double Termini Pruning) is a matter of checking for each edge $e = (u, v)$ if all $L(e)$ end in either u or v, which means that for both u and v, we have to check if the pairwise intersection between $L(e)$ and the lines on at most $D - 1$ other adjacent edges is empty. This can be done in $O(MD)$, as we assumed the sets of lines to be sorted. A single round of Pruning Rule 3 can therefore be applied in $O(|E|MD)$.

4.4.2 Complexity of Cutting Rules. The complexity of a single round of Cutting Rule 1 (single line cut) is trivially $O(|E|)$. For Cutting Rule 2 (terminus detachment) we again have to check for each edge $e = (u, v)$ if all $L(e)$ end in either u or v. The complexity is thus the same as for Pruning Rule 3, and a single round of Cutting Rule 2 can be applied in $O(|E|MD)$.

4.4.3 Complexity of Line Untangling Rules. In Untangling Rule 1 (Full X), for a single edge, we have to check $\binom{D}{2} < D^2$ partners of adjacent edges for line equivalency in the worst case, which can be done in $O(MD^2)$ A single round of Untangling Rule 1 can therefore be applied in $O(|E|MD^2)$.

In Untangling Rule 2 (Full Y), we have to check for the non-terminus node if the $L(e)$ of all minor leg edges are completely contained in the major leg. This can be done in $O(MD)$, as the number of minor legs is always smaller than D, and we can check whether the $L(e)$ of a single edge is contained in the major leg in time linear in M. We also have to check whether the minor leg edges are pairwise disjoint, which takes $O(MD^2)$. A single round of Untangling Rule 2 can therefore be applied in $O(|E|MD^2)$

In Untangling Rule 4 (Double Y) and 5 (Partial Double Y), we have to apply the same checks as for rules 2 and 3 twice, but we also have to establish the bijection m between the left and the right legs. This can be done in $O(MD)$ as we assumed the adjacency lists to be sorted. A single round of Untangling Rules 4 and 5 can therefore be applied in $O(|E|MD^2)$.

Untangling Rule 6 (Stumps) has the same complexity as Untangling Rule 4 (Double Y) and Untangling Rule 5 (Partial Double Y). Finding such structures has the same complexity, and dummy edge extension can be done in constant time. A single round of each untangling rule can therefore be applied in $O(|E|MD^2)$.

ALGORITHM 1: Full pruning, cutting, and untangling of a line graph G. $MaxDeg(G)$ returns the current maximum node degree of G.

```
G ⟵ PruningRule2(G);
n ⟵ M;
while n ≥ 0 do
    G ⟵ UntanglingRule[1 − 6](G);
    m ⟵ MaxDeg(G);
    while m ≥ 0 do
        G ⟵ PruningRule[1, 3](G);
        G ⟵ CuttingRule[1 − 2](G);
        m ⟵ m − 1;
    end
    n ⟵ n − 1;
end
```

4.4.4 Algorithm for Full Untangling, Pruning and Cutting. Next, we determine upper bounds for the number of rounds we may apply all the pruning, cutting, and untangling rules together until none of the rules may be applied anymore. We then use these bounds to describe an algorithm for full untangling, pruning, and cutting.

All pruning rules and all cutting rules are idempotent. Therefore, the maximum number of rounds for each of these rules—when applied alone—is trivially 1. When combined, the application of Cutting Rule 2 or Pruning Rule 3 may lead to additional node contraction possibilities (see for example Figure 11 (right), where v may now be eligible for pruning). The application of Pruning Rule 1 may then again lead to additional situations where Cutting Rules 1 and 2 or Pruning Rule 3 may be applied.

However, the number of rounds we can apply Pruning Rule 3 or Cutting Rule 2 together is trivially upper bound by $D - 1$ (where D is again the maximum node degree of G), as no pruning rule and no cutting rule increase D and as we only detach terminus nodes with $\deg(v) > 1$. For any graph G, the maximum number of rounds we can apply *all* pruning and cutting rules together is thus upper bound by D.

We now consider the untangling rules. Each untangling rule either splits an edge e into two edges e' and e'' (Untangling Rules 2–6), or detaches an edge pair from a node (untangling rule 1).

As $L(e')$ and $L(e'')$ are always disjoint and contain both at least one line, the maximum number of rounds we can apply Untangling Rules 2–6 in a standalone fashion is $M - 1$. The number of rounds we can apply Untangling Rule 1 (Full X) in a standalone fashion is 1. However, as Untangling Rule 3 (Partial Y) and Untangling Rule 5 (Partial Double Y) may increase node degrees, they may also make nodes eligible for Untangling Rule 1 again—but this can again only happen $M - 1$ times.

As no untangling rule, no cutting rule and no pruning rule increases M, the maximum number of rounds we can apply all untangling rules together is always upper bound by M, even when combined with pruning or cutting rules.

Algorithm 1 thus completely prunes, cuts, and untangles an input line graph. The outer loop applies untangling rules 1–6 exactly M times, while the inner loop applies the cutting and pruning rules $MaxDeg(G)$ times, which is the maximum node degree of G at this moment. At the end of the inner loop, we can therefore be sure that no pruning or cutting rule may be applied anymore, and at the end of the outer loop, we we can be sure no untangling rule may be applied anymore.

Table 1. Line Graph Dimensions and Line Ordering Search Space Size $|\Omega|$ for Our Test Datasets

	$\lvert\mathcal{S}\rvert$	$\lvert V\rvert$	$\lvert E\rvert$	$\lvert\mathcal{L}\rvert$	M	D	$\lvert\Omega\rvert$
Freiburg	74	80	81	5	4	4	6×10^{13}
Dallas	108	117	118	7	4	4	1×10^{20}
Chicago	143	153	154	8	6	4	4×10^{33}
Stuttgart	192	219	229	15	8	4	3×10^{103}
Turin	339	398	435	14	5	5	1×10^{85}
New York	456	517	548	26	9	5	2×10^{267}

$\mathcal{S}$ are the stations, V the graph nodes, E the graph edges, and $\mathcal{L}$ the transit lines. M is the maximum number of lines per edge. D is the maximum node degree in the graph.

We additionally make use of the fact that no cutting, no pruning and no untangling rules increase the number of lines in any edge, so Pruning Rule 2 has to be applied only once at the beginning.

4.4.5 Complexity of Full Untangling, Pruning and Cutting. In the worst case, the cutting rules may double the number of edges, and the untangling rules may increase the number of edges by a factor of M. So, if $|E|$ is the number of edges in the original input line graph, then the maximum number of edges $|\hat{E}|$ in any intermediate line graph will be $O(|E|M)$. Similarly, if D is the maximum node degree in the original input line graph, then the maximum node degree $\hat{D}$ in any intermediate line graph will be $O(DM)$. The outer loop of our algorithm will hence run at most M times, while the inner loop will run at most $\hat{D} \in O(DM)$ times. The worst case complexity is therefore

$$O\Big(\overbrace{|E|M^2}^{\text{prun. 2}} + M \times \Big(\overbrace{|\hat{E}|M\hat{D}^2}^{\text{untang. 1-6}} + \hat{D} \times (\overbrace{|\hat{E}|M}^{\text{prun. 1}} + \overbrace{|\hat{E}|MD}^{\text{prun. 3}} + \overbrace{|\hat{E}|}^{\text{cut. 1}} + \overbrace{|\hat{E}|M\hat{D})}^{\text{cut. 2}})\Big)\Big) \tag{18}$$

$$= O(|\hat{E}|M^2\hat{D}^2) = O(|E|M^4D^2). \tag{19}$$

For completeness, we recall that we assumed each edge $L(e)$ and each adjacency list to be sorted. The former can be sorted in $O(|E|M \log M)$, the latter in $O(|E|D \log D)$, so the asymptotic worst case running time is not affected by this assumption.

We note that in practice, both M and D are usually very small. For our test datasets, the maximum M was 9, and the maximum D was 5 (Table 1). For all practical purposes, we are confident that both D and M can be considered a constant factor.

4.5 Full Solve through Untangling

For special classes of input line graphs, applying graph untangling and pruning rules is enough to solve the line ordering optimization problem. We consider Figure 24 (left). The input line graph is a tree, where the children of each node v are connected to v by edges e_i in such a way that the $L(e_i)$ are pairwise disjoint and their union is exactly the set of lines on the edge connecting v to its parent. A repeated application of our pruning and cutting rules will then lead to a graph consisting of components of two nodes u, v, an edge $\{u, v\}$ and $|L(\{u, v\})| = 1$. In the example shown in Figure 24 (left), Untangling Rule 1 (Full X) breaks up b and a into b', b'', a', and a'' and adds two edges with $L(\{a', b'\}) = \{A, B\}$ and $L(\{a'', b''\}) = \{C\}$, then Pruning Rule 2 (Line Partner Collapsing) contracts a'' and b''. The same happens at c, resulting in the three edges depicted in Figure 24 (middle).

Another class of line graph that is more likely to occur in real-world transit networks is depicted in Figure 25 (left). Lines progressively join a main branch (e_2 in the example) and then leave the branch again in the exact same order. For example, in Figure 25 (left), the main branch with lines

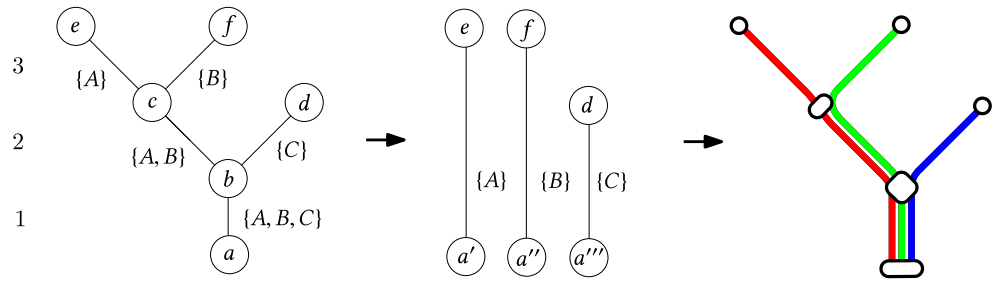

Fig. 24. Left: A tree line graph, the search space size for the line ordering optimization is $3! \times 2! = 12$. Middle: Optimization graph after full untangling (in this case: repeated application of Pruning Rule 2 (Node Contraction) and Untangling Rule 2 (Full Y)), the search space size is now 1. Right: Graph rendered with the ordering obtained from untangling.

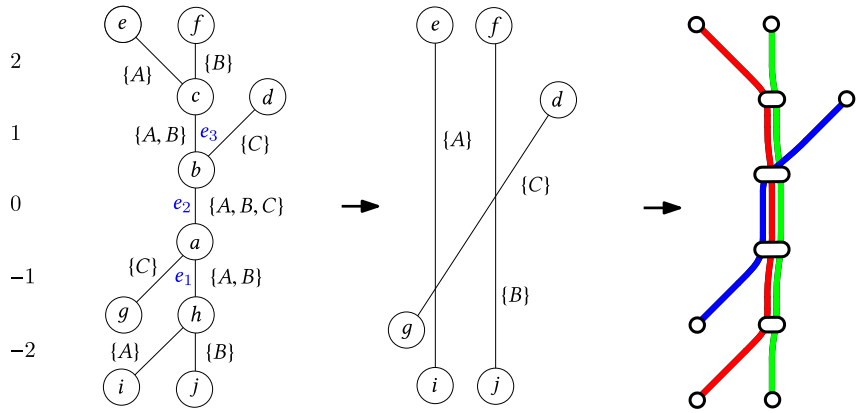

Fig. 25. Left: Special class of line graph in which lines join and leave a main branch in the same order, the search space size for the line ordering optimization is $3! \times 2! \times 2! = 24$. Middle: Optimization graph after full untangling (in this case: repeated application of Pruning Rule 1 (Node Contraction) and Untangling Rule 4 (Double Y structure)), the search space size is now 1. Right: Graph rendered with the ordering stored for e_1, e_2, and e_3 from untangling.

A, B, C is first left by C at node b and then by B at node c (upward direction). Equally, C leaves at a, and B at h (downward direction). A repeated application of Pruning Rule 1 (Node Contraction) and Untangling Rule 4 (Full Double Y) will lead to the optimization graph depicted in Figure 25 (middle).

In both cases, the input line graph is untangled until we only have single threads left that do not require further optimization, because their search space size is 1.

5 RENDERING

This section describes stage 3 of LOOM: Given the line graph as computed in stage 1, and a line ordering for each edge as computed in stage 2, render the actual map. We split this into four basic steps, as illustrated in Figure 26.

In the first step (1), a basic skeleton of the map is rendered. We make use of the fact that only a single ordering is imposed on each $L(e)$ and draw each $l \in L(e)$ by perpendicular offsetting the segment's geometry τ_e by $-w\,|L(e)|\,/2 + w\,(p_e(l) - 1)$, where w is the desired line width. As τ_e is just a piecewise linear curve, any method for offsetting (open) polygons may be used. Each drawn

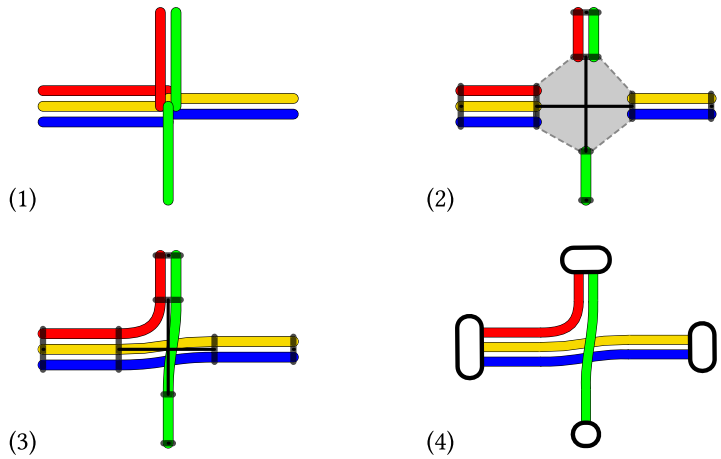

Fig. 26. The four steps of rendering a given line graph: (1) render ordered lines as edges, (2) free node area, (3) render inner connections, and (4) render station overlays.

node v now has $\deg(v)$ node fronts; see Figure 26 (2). The width of each node front depends on the number of lines on the incident edge and on the the line width w.

In the next step (2), we make room for the line connections between these node fronts by expanding them. As a stopping criteria for this expansion, we simply use a maximum distance from the node front to its original position.

In a third step (3), the line connections in the node are then rendered by connecting all port pairs (3). In our experiments, we used cubic Bézier curves for this, but for schematic maps a circular arc or even a straight line might be preferable.

In the last step (4), we render the stations. This is trivial for nodes of degree 1 and 2, but more complicated in larger stations with multiple lines. We found that the buffered node polygon already yields reasonable results here, although with much potential for improvement. We also experimented with rotating rectangles until the total sum of the deviations between each node front orientation and the orientation of the rectangle was minimized. Both approaches can be seen in Figure 1.

In Figure 26 (2), it may happen that during expansion of some node u, we may reach the node front of a neighboring node v. In such a case, the current node front expansion of u has to stop on edge $\{u, v\}$. This incomplete node front expansion may lead to rendering artifacts if not enough space was freed to actually render smooth Bézier curves. A solution to this problem is to merge neighboring nodes if their node fronts collide during expansion. To avoid overlapping station markers on maps with small resolutions, merged station nodes could only be rendered as a simple station marker. The selection of this *master* station marker is a matter of station ranking and can be based, for example, on the number of lines serving a station.

6 EVALUATION

We tested LOOM on the public transit schedules of three cities in Europe and three cities in the U.S.: Freiburg, Dallas, Chicago, Stuttgart, Turin, and New York. Table 1 provides the dimensions of each dataset and the time needed to extract the line graph.

For each dataset, we considered three versions of the line graph: the baseline graph (constructed from the GTFS data), the pruned graph, and the pruned and fully untangled graph. For each graph, we considered three ILP variants: the baseline ILP (b-ILP), the improved ILP (i-ILP), and the improved ILP with added separation penalty (i-ILP*). For each ILP, we evaluated

Table 2. Main Results for a Selection of Our Methods (Those with Marked Differences between Them) on All Six Datasets

	extraction	ordering				quality				×	‖
method		b-ILP	b-ILP	i-ILP*	i-ILP*	Hill*	Hill*	Hill*	i-ILP*	ILP*	
graph		baseline	pruned	pruned	untang	baseline	pruned	untang	untang	untang	
Freiburg	**0.7s**	1s	41ms	13ms	**13ms**	188.4	70.6	70.1	**48**	6	0
Dallas	**3s**	1.4s	0.2s	10ms	**10ms**	170.3	37.7	25.3	**9**	3	0
Chicago	**13.5s**	—	2m	0.3s	**0.2s**	463.3	131.3	123.5	**80**	27	0
Stuttgart	**7.7s**	—	10h	2.1s	**0.3s**	1317.7	434.3	325.7	**156**	64	2
Turin	**4.9s**	14m	2m	0.4s	**0.4s**	1083.9	474.7	430.2	**184**	45	2
New York	**3.7s**	—	—	1.5s	**0.5s**	4487.4	1103.3	511.7	**215**	59	2

Column 2: the method-independent time to extract the graph from the GTFS data. Columns 3–6: the time needed for the line ordering for four different combinations of ILP and graph (best times in bold). Columns 7–10: the objective value for the best local-search method and our best ILP-based method (best values in bold). Columns 11 and 12: the number of crossings and line separations, respectively, of our best ILP-based method. b-ILP stands for our basic ILP, i-ILP for our improved ILP, Hill* for steepest-ascent hill climbing, and the * indicates an optimization with separation penalty.

three solvers: the GNU Linear Programming Kit (GLPK), the COIN-OR CBC solver, and Gurobi (GU). We additionally considered six heuristic optimization methods: exhaustive search (Exh), steepest ascent hill climbing (Hill), and simulated annealing (Ann), and their variants with added separation penalty (Exh*, Hill*, Ann*). Since the cutting rules described in Section 4.2 have only little effect on the baseline and pruned graphs, we only applied them to the untangled graphs.

For each node v, the penalty for a crossing between edge pairs ($\{A, B\}$ in Figure 3 (left)) was $4 \times \deg(v)$, for other crossings ($\{A, B\}$ in Figure 3 (right)) it was $\deg(v)$. The line separation penalty was $3 \times \deg(v)$. We found that these penalties produced nicer maps than a uniform penalty. This would imply $w_{\mathcal{S}\times} = 4 \times \max_{v \in V} \deg(v)$ and $w_{\mathcal{S}\|} = 3 \times \max_{v \in V} \deg(v)$. However, we found that moving some crossings or separations to stations with a degree greater than 2 yielded better looking results. Hence, crossings in $v \in \mathcal{S}$ were punished with $w_{\mathcal{S}\times}$ if $\deg(v) = 2$ and otherwise with $3 \times \deg(v)$ (normal crossing) or $12 \times \deg(v)$ (edge-pair crossing). Similarly, in-station line separations where punished with $w_{\mathcal{S}\|}$ if $\deg(v) = 2$ and $9 \times \deg(v)$ otherwise. Note that Lemma 4.2 still holds, because we did not change the punishment for degree 2 stations. Also note that separations were only considered in i-ILP* and thus depended on the solver and the input order in b-ILP and i-ILP.

We will first give an overview of our main results in Section 6.1. Afterward, we compare our different ILP variants (Section 6.2) and optimization heuristics (Section 6.3) on the baseline graph and after pruning rules were applied. We then evaluate the effects of line graph untangling on both the ILPs and the optimization heuristics in Section 6.4. Finally, a comparison of our maps to manually designed maps published by transportation agencies is done in Section 6.5.

6.1 Results Overview

Our main results can be found in Table 2. The line graph extraction from the input GTFS data took less than 15s for each tested dataset and 5.6s on average. With i-ILP* on the untangled graph we were able to find an optimal line ordering in less than half a second for each dataset and in 0.2s on average.

The results of the heuristic optimization techniques were generally unsatisfactory. However, prior application of both the pruning and untangling rules described Section 4 had significant impact on the final optima found by the heuristics. Without prior graph untangling or pruning, the final objective function values were 2 to 9 times higher than after graph untangling. On average,

Table 3. Dimensions and Solution Times on All Six Datasets for Our Three ILP Variants, Three Solvers, with and Without Graph Pruning

		On baseline graph				On pruned graph					
		rows×cols	GLPK	CBC	GU	rows×cols	GLPK	CBC	GU	×	||
Freiburg	b-ILP	4k×346	12m	0.5m	1s	376×122	0.3ms	0.7s	41ms	4	2
	i-ILP	537×390	0.2s	47ms	15ms	173×131	8ms	0.1s	10ms	4	2
	i-ILP*	667×451	2.6s	0.2s	0.1s	202×144	50ms	0.1s	13ms	6	0
Dallas	b-ILP	5.4k×487	3m	41s	1.4s	1.5k×153	4s	4s	0.2s	3	3
	i-ILP	778×557	0.5s	80ms	16ms	236×171	30ms	39ms	7ms	3	3
	i-ILP*	974×649	1.5s	134ms	35ms	305×203	38ms	125ms	10ms	3	0
Chicago	b-ILP	41k×861	—	—	—	8.2k×266	—	47m	2m	22	4-7
	i-ILP	1.4k×982	9s	1s	41ms	394×285	0.8s	0.1s	10ms	22	4-7
	i-ILP*	1.9k×1.2k	47m	19s	1.8s	505×338	23s	3.8s	0.3s	27	0
Stuttgart	b-ILP	224k×2.4k	—	—	—	44k×950	—	—	10h	60	11
	i-ILP	4.1k×2.8k	—	3.5s	0.1s	1.5k×1k	8s	0.2s	36ms	60	7-15
	i-ILP*	5.6k×3.5k	—	2m	47s	2.1k×1.3k	—	36s	2.1s	64	2
Turin	b-ILP	24k×2.1k	—	—	14m	13k×1k	—	—	2m	42	6
	i-ILP	3.3k×2.4k	2m	0.6s	0.1s	1.6k×1.1k	16s	0.3s	71ms	42	6-10
	i-ILP*	4.3k×2.9k	—	14s	1s	2k×1.4k	—	5.2s	0.4s	45	2
New York	b-ILP	229k×5.2k	—	—	—	96k×2.3k	—	—	—	—	—
	i-ILP	8.6k×6k	—	1.8s	0.2s	3.7k×2.5k	—	0.7s	0.1s	55	6-16
	i-ILP*	12k×7.4k	—	2.5m	12s	4.9k×3.2k	—	50s	1.5s	59	2

The number of constraints (rows) and the number of variables (columns) is given as rows × cols. A time of — means we aborted after 12 hours. The last two columns show the number of crossings (×) and separations (||) after optimization.

the objective function value obtained with the best performing technique (steepest ascent hill climbing with prior line graph untangling) was still over 2 times higher than the optimal value.

With b-ILP on the baseline graph, three of six datasets could not be solved in under 12 hours with the fastest solver (Gurobi). With b-ILP on the pruned graph, one of these three datasets could still not be solved in under 12 hours. Compared to i-ILP* on the pruned graph, graph untangling improved the solution times by a factor of up to 7 for some datasets but had little effect on the solution time for other datasets, despite the fact that it both reduced the search space and the sizes of the resulting ILP quite significantly. This indicates that the impact of graph untangling largely depends on the structure of the underlying network but also that very good solvers may already apply heuristics that implicitly perform something similar to our graph untangling. However, for less sophisticated solvers, the performance impact was more significant (see Section 6.4.2), and the line orderings for some of the smaller datasets could even be optimized with a simple exhaustive search after untangling was applied (see Section 6.4.3).

6.2 Comparison of ILP Variants

Table 3 shows the detailed results of the ordering optimizations with ILPs for our six datasets on the baseline optimization graph and the optimization graph after pruning rules were applied. Tests were run on an Intel Core i5-6300U machine with four cores à 2.4GHz and 12GB RAM. The CBC solver was compiled with multithreading support and used with the default parameters and `threads=4`. The GLPK solver was used with the feasibility pump heuristic (`fp_heur=ON`), the proximity search heuristic (`ps_heur=ON`), and the presolver enabled (`presolve=ON`). We used Gurobi with the default parameters.

6.2.1 ILP Sizes and Solution Times. As expected in Section 3.2, the sizes of i-ILP are significantly smaller than those of b-ILP. On average, the number of rows was reduced by 92%. With added separation penalty (i-ILP*), the number of rows was still reduced by 89%. For i-ILP, the optimal orderings on the pruned graph could be found in under 100ms with Gurobi and in under 1s with CBC, on any dataset. For i-ILP*, the ILP could be solved on the pruned graph in under 2.5s with Gurobi for any dataset and in under 1 minute with CBC. Although the ILPs for i-ILP* were only slightly larger than for i-ILP, optimization on the pruned graph took 19 times longer on average with the fastest solver.

6.2.2 Effects of Line Graph Pruning. On the baseline graph, b-ILP could not be solved for larger datasets except Turin with Gurobi. After graph pruning, a solution was found for Stuttgart, Chicago, and New York in under 12 hours. As expected in Section 4, graph pruning made the ILPs significantly smaller. With i-ILP, both the number of rows and the number of columns decreased by 64% on average. With i-ILP*, the decrease was 64% and 63%, respectively. With the fastest solver and i-ILP, graph pruning lead to speedups by a factor between 4 for Chicago and 2 for New York. With i-ILP*, this speedup factor was between 22.4 for Stuttgart and 2.5 for Turin.

6.3 Performance of Optimization Heuristics

In this section, we evaluate the results of six heuristics for the line ordering optimization: an exhaustive search (Exh), steepest-ascent hill climbing (Hill), and simulated annealing (Ann), as well as the same techniques with added separation penalty (Exh*, Hill*, Ann*).

Exhaustive search simply explores the entire search space Ω to find the optimal line ordering $o \in \Omega$. Even for small input graphs, the size of Ω is immense. For the Freiburg dataset, the size of Ω is $\sim 6 \times 10^{13}$ and for Stuttgart it is $\sim 3 \times 10^{103}$ (Table 1).

Steepest-ascent hill climbing and simulated annealing are local-search heuristics. In each step, they explore the neighborhood of the current ordering. Steepest-ascent hill climbing simply chooses the neighbor closest to the solution. Simulated annealing selects a neighbor at random and stops if the value of the objective function has not changed for the past $k = 5{,}000$ iterations. At state o, we choose the randomly selected neighbor as the next state o' if it improves the overall objective function value θ (that is, if $\theta(o') < \theta(o)$) or else with probability $P(o, o', T) = e^{-(\theta(o')-\theta(o))/T}$, where T is the current annealing temperature. At iteration i, we set $T = 1000/n$.

Each method was evaluated 50 times and the results were averaged. For each evaluation, the initial line orderings of the graph were randomized.

6.3.1 Solution Times. Table 4 shows the results of the heuristic ordering optimization for three of our six datasets: Freiburg, Chicago, and Stuttgart. For brevity, we only chose a representative subset of our testing dataset, consisting of a small network, a medium network, and a large network. On the baseline graph (without any simplification), Exh was infeasible even for the small dataset of Freiburg. If we assume that we can check 10,000 ordering configurations per second, then an exhaustive search for Freiburg would still take about 70,000 days on the baseline graph.

Final ordering configurations and graph scores of both Hill and Ann were comparable, but Hill produced slightly better orderings on average. However, the cost of exploring the entire local neighborhood at each iteration may get very high. The effect of this can be seen in the evaluation for Stuttgart, where Hill took 8 minutes on average. We also experimented with probabilistic hill climbing, which is equivalent to our simulated annealing approach with $T = 0$ but found the results to be generally inferior to both Hill and Ann. With added line separation penalty (Exh*, Hill*, Ann*), solution times where generally slightly larger, even when the number of iterations was smaller. This was because of the additional overhead of checking each ordering configuration for line separations.

Table 4. Dimensions, Solution Times, and Final Graph Scores for Freiburg, Chicago, and Stuttgart and Six Baseline Heuristics for the Line Ordering Problem: Exhaustive Search with (Exh), Steepest-Ascent Hill Climbing (Hill), Simulated Annealing (Ann), and Their Counterparts with Separation Penalty (Exh*, Hill*, Ann*)

		On baseline graph						On pruned graph					
		$\|\Omega\|$	t	iters	$\times$	$\|\|$	θ	$\|\Omega\|$	t	iters	$\times$	$\|\|$	θ
Freiburg	Exh		—	—	—	—	—		2s	55k	5	2	30.0
	Hill		22ms	9.4	13.8	10.1	163.8		5m	2.0	5.6	1.8	44.8
	Ann		45ms	7.2k	13.8	10.9	168.7		80ms	5.9k	5.8	2.0	43.9
	Exh*	6×10^{13}	—	—	—	—	—	6×10^{4}	2.5s	55k	6.0	0.0	48.0
	Hill*		30ms	9.5	12.7	4.1	188.4		4ms	4.1	7.1	1.1	70.6
	Ann*		0.5s	7.6k	12.1	4.7	195.6		0.1s	5.8k	6.9	1.2	70.9
Chicago	Exh		—	—	—	—	—		—	—	—	—	—
	Hill		1.3s	17.4	41.7	37.6	481.4		0.5s	6.8	20.4	7.4	125.6
	Ann		0.1s	12k	45.0	39.0	512.6		0.4s	8.7k	20.1	7.1	120.6
	Exh*	4×10^{33}	—	—	—	—	—	5×10^{9}	—	—	—	—	—
	Hill*		2.1s	16.6	38.5	13.3	463.3		0.5s	6.0	21.6	3.2	131.3
	Ann*		0.2s	15.1k	40.7	15.0	556.2		0.4s	7.9k	21.3	3.1	133.6
Stuttgart	Exh		—	—	—	—	—		—	—	—	—	—
	Hill		5.7m	52.4	127.0	114.6	1648.7		1.8m	20.3	65.4	36.8	485.8
	Ann		0.5s	20k	139.5	128.6	1749.8		0.4s	10.1k	67.2	43.4	548.9
	Exh*	3×10^{103}	—	—	—	—	—	2×10^{38}	—	—	—	—	—
	Hill*		7.4m	50.2	105.5	43.4	1317.7		2m	21.5	64.0	12.9	434.3
	Ann*		0.8s	25.6k	123.3	55.7	1819.3		0.7s	12.5k	65.0	17.1	519.3

Results are given for the baseline graph and the pruned graph. $|\Omega|$ is the search space size, t the solution time. A time of — means we aborted after 12 hours. The number of iterations is shown in column *iters*, $\times$ is the number of crossings in the optimized graph, $\|$ the number of separations, and θ is the final graph score. For optimization without separation penalty, the final graph scores only include crossing penalties. Optimal graph scores can be found in Table 2.

6.3.2 Effects of Line Graph Pruning. If the input line graph was first simplified with pruning rules 1–3 (Section 4.1), then an optimal solution could be found with an exhaustive search for Freiburg in 4.2s. For Ann, Ann*, Hill, and Hill*, graph pruning enhanced the final graph scores by a factor of 3.5 on average. This was to be expected, as pruning lead to a significant reduction of the search space size (for example, pruning reduced the search space size of Freiburg by 9 orders of magnitude). For Hill and Hill*, solution times were on average 10 times faster on the pruned optimization graph than on the baseline graph (with a large improvement for Hill on the Freiburg dataset). Interestingly, solution times for simulated annealing did not improve much or got worse after pruning, despite a lower number of iterations. This is because before pruning, the probability of choosing an edge with only a single line on it (or where both nodes have a degree of 2) as the random candidate for the next state is greater than after pruning, as pruning only leaves intersection nodes in the graph. Checking the scores of these larger nodes takes more time, which leads to larger average iteration times.

6.3.3 Optimality of Results. With prior pruning and added separation penalty, the local optima found by our simulated annealing approach where on average a factor of 3.3 higher than the optimal solutions. With steepest ascent hill climbing, they were on average a factor of 2.1 higher. The average Freiburg score for Ann* on the pruned graph (70.6) was closest to the optimal score

Table 5. Dimensions of Line Graphs with Only Pruning Rules Applied (Pruned Graph) and after Full Untangling Was Applied (Untangled Graph), as well as Time Needed for Untangling

	Pruned graph					Untangled graph						Largest component			
	$\lvert V_p \rvert$	$\lvert E_p \rvert$	M_p	$\lvert \Omega_p \rvert$	t	$\lvert V \rvert$	$\lvert E \rvert$	M	$\lvert \Omega \rvert$	C	C^1	$\lvert \hat{V} \rvert$	$\lvert \hat{E} \rvert$	$\hat{M}$	$\lvert \hat{\Omega} \rvert$
Freiburg	20	21	4	$6{\times}10^4$	3ms	22	19	4	$9{\times}10^3$	3	2	18	17	4	$9{\times}10^3$
Dallas	24	24	4	$2{\times}10^6$	11ms	36	27	4	$2{\times}10^3$	9	8	12	11	4	$2{\times}10^3$
Chicago	23	24	6	$5{\times}10^9$	5.8ms	24	22	6	$1{\times}10^9$	3	2	18	18	6	$1{\times}10^9$
Stuttgart	50	58	6	$2{\times}10^{38}$	15ms	58	52	6	$7{\times}10^{22}$	7	5	24	24	6	$1{\times}10^{12}$
Turin	91	124	5	$5{\times}10^{40}$	15ms	157	134	5	$2{\times}10^{31}$	26	23	70	71	5	$3{\times}10^{29}$
New York	110	138	9	$6{\times}10^{92}$	20ms	107	93	6	$3{\times}10^{36}$	17	13	62	64	6	$5{\times}10^{34}$

$\lvert \Omega_p \rvert$, $\lvert \Omega \rvert$, and $\lvert \hat{\Omega} \rvert$ are the search space sizes. t is the time needed to produce the untangled graph from the original input line graph. For each untangled graph, C is the number of its components, and C^1 the number of components with $M = 1$ (which can be solved trivially). Under largest component, we give the dimensions of the largest component in the untangled graph.

(48), but for Stuttgart, the average Ann* score on the pruned graph (519.3) was 3.33 times higher than the optimal score (156).

6.4 Effects of Line Graph Untangling

To measure the effects of our graph untangling rules described in Section 4.3, we re-ran all of our six heuristic optimizations and our 3 ILPs on the untangled optimization graph of our datasets. We followed the algorithm described in Section 4.4 and thus also applied cutting rules.

6.4.1 Effect on Optimization Graph Size. Table 5 shows the optimization graph dimensions after full untangling for all of our input datasets, compared to the sizes of the optimization graph with only pruning rules applied. For each dataset, we measured the time t needed for full untangling, the overall number of nodes after untangling, the overall number of edges after untangling, the maximum number of lines per edge M and the search space size $|\Omega|$. Additionally, we counted the number of connected components C, the number C^1 of connected components with $|\Omega| = 1$ (which do not need optimization), and the dimensions of the *largest* connected component.

Compared to just applying pruning rules, graph untangling further reduced the size of the search space Ω, sometimes dramatically. For New York, the search size space was reduced by 56 orders of magnitude compared to the pruned graph. Compared to the original line graph, the search size space was reduced by 212 orders of magnitude with graph untangling. For Freiburg, untangling further reduced the search space size to 9,200, which can be explored with an exhaustive search in minimal time (see Section 6.4.3).

The number of connected components retrieved from graph untangling was always larger than 2. For three of our datasets (Freiburg, Dallas, and Chicago), all but one connected component had a solution size of exactly 1 and did not need any further ordering optimization. For the remaining networks, the search space of the largest connected component was 2–10 orders of magnitude smaller than the combined search space of the untangled graph.

For some datasets, the number of nodes and edges in the untangled graph was higher than in the pruned graph. This was to be expected, as we are splitting both nodes and edges in our untangling rules. However, even for those datasets, the search space sizes still went down significantly, as breaking up edges in our untangling rules always reduces the number of lines per edge.

6.4.2 Effect on ILP Sizes and Solution Times. Table 6 shows the solution times and dimensions of the line ordering optimization ILPs for our test datasets after applying our untangling rules,

Table 6. Impact of Graph Untangling on ILP Sizes and Solution Times

		Pruned graph				Untangled graph			
		rows×cols	GLPK	CBC	GU	rows×cols$_{max}$	GLPK	CBC	GU
Freiburg	i-ILP	173×131	8ms	35ms	10ms	145×109	5ms	25ms	10ms
	i-ILP*	202×144	50ms	0.1s	13ms	165×118	7ms	0.1s	13ms
Dallas	i-ILP	236×171	30ms	39ms	7ms	123×90	5ms	22ms	8ms
	i-ILP*	305×203	38ms	125ms	10ms	149×102	17ms	33ms	10ms
Chicago	i-ILP	394×285	0.8s	0.1s	10ms	371×267	0.1s	0.1s	11ms
	i-ILP*	505×338	23s	3.8s	0.3s	482×320	18s	3s	0.2s
Stuttgart	i-ILP	1.5k×1k	8s	0.2s	36ms	470×331	0.2s	0.2s	30ms
	i-ILP*	2.1k×1.3k	—	36s	2.1s	638×411	21.5s	7.3s	0.3s
Turin	i-ILP	1.6k×1.1k	16s	0.3s	71ms	1.2k×864	4.3s	0.3s	73ms
	i-ILP*	2k×1.4k	—	5.2s	0.4s	1.6k×1k	—	8s	0.4s
New York	i-ILP	3.7k×2.5k	—	0.7s	0.1s	1.3k×943	2m	0.3s	80ms
	i-ILP*	4.9k×3.2k	—	50s	1.5s	1.8k×1.1k	—	5.9s	0.5s

For the pruned graph, solution times and graph dimensions are are taken from Table 3. The number of constraints (rows) and the number of variables (columns) is again given as rows × cols. If the untangled graph had multiple components, then rows × cols$_{max}$ gives the dimensions of their *largest* ILP. For multiple components, the respective ILPs were solved iteratively and solution times summed.

compared to our initial approach of just applying pruning rules. If multiple components had to be optimized after untangling, then optimization was done iteratively. For datasets where this was the case, solution times in Table 6 are the summed solution times of all ILP solves.

Compared to only applying pruning rules, untangling further reduced the number of constraints for each of our test datasets. For the datasets where only a single ILP was left to solve after untangling (Freiburg, Dallas, Chicago), the number of constraints was reduced by 23% for i-ILP and by 25% i-ILP*. For the datasets where multiple ILPs where left to solve after untangling (Stuttgart, Turin, New York), the number of constraints of the dominating ILP compared to the number of constraints of the original ILP on the pruned graph was reduced by 53% for i-ILP and by 50% for i-ILP*.

For Stuttgart, of the 2 ILPs to solve after untangling, the largest ILP dominated the second largest ILP by a factor of 1.1 in the number of constraints for both i-ILP and i-ILP* (both ILPs were nearly equal in size). For New York, of the three ILPs to solve after untangling, the largest ILP clearly dominated the second largest ILP by a factor of 67 in the number of constraints for i-ILP and by a factor of 80 for i-ILP*.

The total average reduction of the number of constraints between the dominating ILP of the untangled graph compared to the ILP of the pruned graph was 38% for both i-ILP and i-ILP*.

For Freiburg and Chicago, a solution time improvement was only noticeable with GLPK. With Gurobi, none of the solution times for i-ILP and i-ILP* were better on the untangled graph than on the pruned graph for Freiburg, Chicago, and Turin. For Chicago, the solution time for i-ILP even went slightly up and quite significantly for Turin with i-ILP* on CBC. One explanation for this is the additional overhead of solving multiple ILPs, one for each connected component with $|\Omega| > 1$. For Turin, however, the dominating ILP of the untangled line graph took 1.5 times longer to solve with CBC than the pruned ILP, despite its search space being 2 orders of magnitude smaller.

Graph untangling gave significant performance gains for larger networks, especially with added separation penalty. The solution time of i-ILP* on Stuttgart was greater than 12 hours with GLPK on the pruned graph, but after graph untangling, the ILPs could be solved in 21.5s for Stuttgart. For New York, neither i-ILP nor i-ILP* could be solved in under 12 hours with GLPK on the pruned

Table 7. Impact of Graph Untangling on Selected Baseline Heuristic Solution Times and Objective Function Values, Compared to the Impact of Only Applying Pruning Rules

		Pruned graph				Untangled graph			
		t	$\times$	$\|\|$	θ	t	$\times$	$\|\|$	θ
Freiburg	Exh*	2.5s	6.0	0.0	48.0	0.2s	6.0	0.0	48.0
	Hill*	4ms	7.1	1.1	70.6	3ms	6.4	0.9	70.1
	Ann*	0.1s	6.9	1.2	70.9	80ms	5.9	0.8	68.7
Chicago	Exh*	—	—	—	—	—	—	—	—
	Hill*	0.5s	21.6	3.2	131.3	0.4s	20.8	3.1	123.5
	Ann*	0.4s	21.3	3.1	133.6	0.4s	20.9	3.1	126.7
Stuttgart	Exh*	—	—	—	—	—	—	—	—
	Hill*	2m	64.0	12.9	434.3	0.4s	56.3	8.7	325.7
	Ann*	0.7s	65.0	17.1	519.3	0.6s	54.9	9.2	328.0

t is time needed for optimization, $\times$ is the number of crossings after optimization, $||$ the number of separations after optimization. The final objective function value is given as θ.

graph, but after untangling, a solution for i-ILP was found in 2 minutes. For Stuttgart, solving i-ILP* with Gurobi was 7 times faster with graph untangling, and for New York, it was 3 times faster. With CBC, solution times for i-ILP* on our datasets Stuttgart and New York also went down significantly. For Stuttgart i-ILP* could be solved 5 times faster and for New York 8.5 times faster using graph untangling. With graph untangling, we were able to compute the optimal line ordering for all test datasets in under 10s with CBC and in under 500ms with Gurobi.

On average, solution times without graph untangling were 1.4 times higher for i-ILP on CBC and 3.4 times higher for i-ILP*. For i-ILP, average solution times with Gurobi did not change with graph untangling, but for i-ILP*, they were 2.4 times faster on average with graph untangling. These results give a hint that more sophisticated ILP solvers may already use generic heuristics that implicitly perform a graph untangling.

Recall that for better reproducibility and comparability between different solvers, we solved the ILPs iteratively if the untangled graph had multiple connected components. Solving them in parallel would further bring down the solution times for the untangled graph and may be able to amortize the overhead costs on smaller datasets.

6.4.3 Effect on Baseline Heuristics. We also evaluated the effect of graph untangling on our six baseline heuristics from Section 6.3. In addition to the time needed for the optimization, we measured the effect of untangling on the final objective function values. Table 7 gives an overview of the effects of graph untangling for Exh*, Hill*, and Ann* on our datasets Freiburg, Chicago, and Stuttgart.

For Freiburg, graph untangling brought down the solution time of a simple exhaustive search to 200ms. This provides evidence that for modestly complex real-world line graphs, an ILP is not necessary to find a solution in acceptable time, even when graph untangling does not completely solve the line-ordering optimization problem. However, for all our other datasets, exhaustive search was not able to find a solution in under 12 hours, even after graph untangling.

In general, our optimization heuristics performed better after graph untangling, both in terms of running time and in terms of the final graph score. For Stuttgart, for example, the final graph score for Ann* averaged from 50 runs went down from 519.3 on the pruned graph to 328.0 on the untangled graph, for Hill* it went down from 434.4 go 325.7. However, both scores were still over 2 times larger than the optimal score (which was 156). For Stuttgart, the running time of Hill* was reduced from 2 minutes to 0.4s.

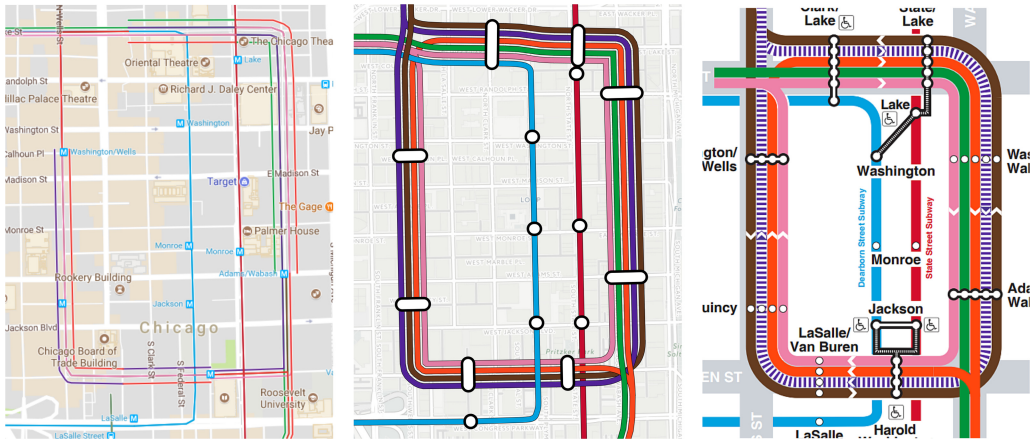

Fig. 27. Left: Google transit map cutout for Chicago. Center: Same area in our automatically generated map. Right: Official CTA map for the same area. Note the near-perfect match of the line-orderings in the official map and our map.

6.5 Comparison to Manually Designed Maps

We did a manual analysis to evaluate the esthetic quality of our work. For our datasets Freiburg, Dallas, Chicago, and Stuttgart, we compared our automatically generated maps to the official maps published by the respective transit agencies.[2] These maps are usually highly simplified and only respect the geographical course of a line to a limited extent. However, they still provide valuable ground truth for the line orderings computed by our ILP (Section 3).

For each official map, we hand-counted the number of line crossings as well as the number of line separations and calculated the overall objective function value in our penalty system. In addition, we counted the number of line swaps T necessary to transform the line ordering of our map into the line ordering of the official map. Line swaps on multiple consecutive edges were only counted once. Figure 27 gives an example of that: Although we have to swap the brown and the purple line on multiple edges between stations to match the official CTA map, we only count a single, consecutive swap.

For our four manually evaluated datasets, we found that a surprisingly low number of line swaps was necessary to transform the line orderings found by our ILP to the line orderings of the official map. Even for the highly complex 2015 Stuttgart map, only four line swaps were required. This is strong evidence that our combination of penalizing line crossings and line separations closely models the esthetics of professional, hand-drawn transit maps.

We also found that our maps always scored better or equal in our penalty system than the official maps, and that only minimal changes to the official map (missed by the designers) would be required to improve the readability. The results can be seen in Table 8. For Dallas, our ILP found a single (trivial) line swap that prevented a line separation at no cost and lowered the objective function value by 66%. For Chicago, our orderings nearly match the ones in the official map, but our ILP found a solution with one additional crossing, but equivalent score. For Stuttgart, four line

[2] http://loom.informatik.uni-freiburg.de/officialmaps/vag.pdf.
http://loom.informatik.uni-freiburg.de/officialmaps/dart.pdf.
http://loom.informatik.uni-freiburg.de/officialmaps/cta.pdf.
http://loom.informatik.uni-freiburg.de/officialmaps/vvs.pdf.

Table 8. Comparison of the Line Orderings in Our Maps and in Manually Designed Official Maps Published by Transportation Authorities

	Official map			Our map			
	×	||	θ	×	||	θ	T
Freiburg	7	1	132	6	0	48	2
Dallas	3	1	27	3	0	9	1
Chicago	26	0	80	27	0	80	1
Stuttgart	65	5	264	64	2	156	4

For the official maps, we hand-counted the number of crossings (×) and separations (||) and calculated the value θ of the objective function in our penalty system. T is the number of line swaps necessary to transform the line orderings in our map into those of the official map. Swaps between the same two lines on consecutive edges were only counted once.

swaps in the official map could reduce both the number of crossings and the number of separations and lower the objective function value by nearly 59%.

7 CONCLUSIONS AND FUTURE WORK

This work presented a complete end-to-end method for producing geographically accurate transit maps from raw schedule data. We evaluated LOOM, a full implementation of this method, and showed that it produces geographically accurate transit maps fast. We demonstrated that our intuition of punishing both line crossings and line separations leads to results that closely resemble the esthetics of manually designed maps.

The biggest challenge was getting the optimal line orderings in acceptable time. We have shown that with an improved formulation of our ILP and several pruning, cutting, and untangling rules we could reduce the solve time by several orders of magnitude for some datasets, compared to our initial approach. With prior graph untangling, an optimal line ordering (with separation penalty) could be found in under 500ms for all our test datasets. This enables our line-ordering optimization techniques to be used in situations where a real-time computation of line orderings is necessary, for example, in a map editor. The whole pipeline (including line graph construction from GTFS schedule data, line-graph minimization, line-ordering optimization and rendering) took less than 15s for all considered inputs.

Compared to the optimal objective function value, the local optima found by two baseline optimization heuristics (steepest ascent hill climbing and simulated annealing) were over 2 times as high, even with prior line graph simplification. However, we showed that even for modestly complex real-world public transit networks, a simple exhaustive search may be enough to optimize both the number of crossings and the number of separations in less than 1s if graph untangling is applied first.

Since the line graph construction required more time than the subsequent line-ordering optimization for some datasets, faster algorithms for extracting the line graph would help to further decrease the running time. It would be interesting to evaluate the adaptability of other map construction algorithms to this problem, both in terms of running time and quality.

As mentioned in Section 5, we see room for improvement in the rendering of station polygons. It may be necessary to enforce a local octilinearity on edges leaving stations for a cleaner look.

Both our line-ordering and rendering steps may be used with any multigraph as input and are not restricted to a geographically accurate network. It may be interesting to evaluate LOOM on schematic transit networks as well.

Last, the ideas behind LOOM may be useful also in a non-transit scenario. For example, one closely related problem is that of wire routing in integrated-circuit design. There, stations correspond to chips and other elements (which in wire routing are indeed of polygonal form), lines correspond to wires, and the geographical course of the lines may correspond to a pre-existing wiring.

REFERENCES

[1] Mahmuda Ahmed, Sophia Karagiorgou, Dieter Pfoser, and Carola Wenk. 2015. A comparison and evaluation of map construction algorithms using vehicle tracking data. *Geoinformatica* 19, 3 (2015), 601–632.

[2] Mahmuda Ahmed and Carola Wenk. 2012. Constructing street networks from GPS trajectories. In *Proceedings of the 20th Annual European Conference on Algorithms (ESA'12)*. Springer, 60–71.

[3] Evmorfia N. Argyriou, Michael A. Bekos, Michael Kaufmann, and Antonios Symvonis. 2008. Two polynomial time algorithms for the metro-line crossing minimization problem. In *Proceedings of the 16th International Symposium on Graph Drawing (GD'08)*. 336–347.

[4] Evmorfia N. Argyriou, Michael A. Bekos, Michael Kaufmann, and Antonios Symvonis. 2010. On metro-line crossing minimization. *J. Graph Algor. Appl.* 14, 1 (2010), 75–96.

[5] Matthew Asquith, Joachim Gudmundsson, and Damian Merrick. 2008. An ILP for the metro-line crossing problem. In *Proceedings of the 14th Symposium on Computing: The Australasian Theory (CATS'08)*, Vol. 77. Australian Computer Society, 49–56.

[6] Michael A. Bekos, Michael Kaufmann, Katerina Potika, and Antonios Symvonis. 2008. Line crossing minimization on metro maps. In *Proceedings of the 15th International Conference on Graph Drawing (GD'07)*. Springer, 231–242.

[7] Marc Benkert, Martin Nöllenburg, Takeaki Uno, and Alexander Wolff. 2007. Minimizing intra-edge crossings in wiring diagrams and public transportation maps. In *Proceedings of the 14th International Conference on Graph Drawing (GD'06)*. Springer, 270–281.

[8] Prosenjit Bose, Vida Dujmović, Ferran Hurtado, Stefan Langerman, Pat Morin, and David R. Wood. 2009. A polynomial bound for untangling geometric planar graphs. *Discr. Comput. Geom.* 42, 4 (2009), 570–585.

[9] Google Developers. 2016. GTFS Static Overview. Retrieved May 15th, 2019 from https://developers.google.com/transit/gtfs.

[10] Anton Dubreau. 2016. Transit Maps: Apple vs. Google vs. Us. Retrieved May 15th, 2019 from https://medium.com/transit-app/transit-maps-apple-vs-google-vs-us-cb3d7cd2c362.

[11] Martin Fink, Herman Haverkort, Martin Nöllenburg, Maxwell Roberts, Julian Schuhmann, and Alexander Wolff. 2013. Drawing metro maps using Bézier curves. In *Proceedings of the 20th International Conference on Graph Drawing (GD'12)*. Springer, 463–474.

[12] Martin Fink and Sergey Pupyrev. 2013. Metro-line crossing minimization: Hardness, approximations, and tractable cases. In *Proceedings of the 21st International Symposium on Graph Drawing (GD'13)*, Vol. 8242. 328–339.

[13] Martin Fink and Sergey Pupyrev. 2015. Ordering metro lines by block crossings. *J. Graph Algor. Appl.* 19, 2, 111–153.

[14] Xavier Goaoc, Jan Kratochvíl, Yoshio Okamoto, Chan-Su Shin, Andreas Spillner, and Alexander Wolff. 2009. Untangling a planar graph. *Discr. Comput. Geom.* 42, 4 (2009), 542–569.

[15] Danny Holten and Jarke J. van Wijk. 2009. Force-directed edge bundling for graph visualization. In *Proceedings of the 11th Eurographics/IEEE–VGTC Conference on Visualization (EuroVis'09)*. The Eurographs Association, 983–998.

[16] Seok-Hee Hong, Damian Merrick, and Hugo A. D. do Nascimento. 2006. Automatic visualisation of metro maps. *J. Vis. Lang. Comput.* 17, 3 (2006), 203–224.

[17] Martin Nöllenburg. 2010. An improved algorithm for the metro-line crossing minimization problem. In *Proceedings of the 17th International Conference on Graph Drawing (GD'09)*. Springer, 381–392.

[18] Martin Nöllenburg. 2014. A survey on automated metro map layout methods. In *Schematic Mapping Workshop*, Essex, UK (2014).

[19] Sergey Pupyrev, Lev Nachmanson, Sergey Bereg, and Alexander E. Holroyd. 2012. Edge routing with ordered bundles. In *Proceedings of the 19th International Conference on Graph Drawing (GD'11)*. Springer, 136–147.

Received November 2018; revised March 2019; accepted May 2019