Efficient Generation of Geographically Accurate Transit Maps

HANNAH BAST, University of Freiburg
PATRICK BROSI, University of Freiburg
SABINE STORANDT, JMU Würzburg

We present LOOM (Line-Ordering Optimized Maps), a fully automatic generator of geographically accurate transit maps. The input to LOOM is data about the lines of a given transit network, namely for each line, the sequence of stations it serves and the geographical course the vehicles of this line take. We parse this data from GTFS, the prevailing standard for public transit data. LOOM proceeds in three stages: (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) construct an ILP that yields a line ordering for each edge which minimizes the total number of line crossings and line separations; (3) based on the line graph and the ILP solution, draw the map. As a naive ILP formulation is too demanding, we derive a new custom-tailored formulation which requires significantly fewer constraints. Furthermore, we present engineering techniques which use structural properties of the line graph to further reduce the ILP size. For the subway network of New York, we can reduce the number of constraints from 229,000 in the naive ILP formulation to about 3,700 with our techniques, enabling solution times of less than a second. Since our maps respect the geography of the transit network, they can be used for tiles and overlays in typical map services. Previous research work either did not take the geographical course of the lines into account, or was concerned with schematic maps without optimizing line crossings or line separations.

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. 2018. Efficient Generation of Geographically Accurate Transit Maps. 1, 1 (November 2018), 27 pages. https://doi.org/10.1145/3274895.3274955

1 INTRODUCTION

Cities with a public transit network usually have a map which illustrates the network and which is posted at all stations. Many map services also feature a transit layer where all lines and stations in an area are displayed. Such a map should satisfy the following main criteria:

(1) It 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.
(2) It should be neatly arranged and esthetically pleasing.

Authors' addresses: Hannah Bast, University of Freiburg, Freiburg, Germany, bast@cs.uni-freiburg.de; Patrick Brosi, University of Freiburg, Freiburg, Germany, brosi@cs.uni-freiburg.de; Sabine Storandt, JMU Würzburg, Würzburg, Germany, storandt@informatik.uni-wuerzburg.de.

XXXX-XXXX/2018/11-ART
https://doi.org/10.1145/3274895.3274955

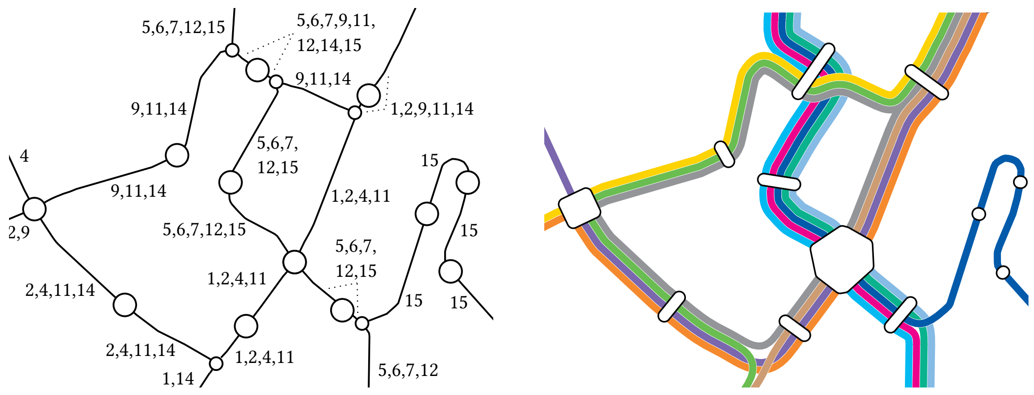

Fig. 1. Left: Excerpt from a line graph which LOOM constructed from the GTFS data for the 2015 light rail network of the city of Stuttgart. Each edge corresponds to a segment of the network where the same set of lines takes the same geographical course. Segment boundaries are often station nodes (large) 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.

(3) It 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, either to make the map fit into a certain format or to simplify the layout, or both.

The goal of this paper 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 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 proceeds in three stages, which we briefly describe in the following along with some notation and terminology that will be used throughout the paper. (1) given the line data, construct the line graph; (2) compute an optimal line ordering for each edge via an ILP; (3) from the line graph and the line ordering, render 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. This data is usually provided as part of a network's GTFS feed.

Line graph construction (Sect. 2): In its 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. Fig. 1, left shows the line graph for an excerpt from the light rail network of Stuttgart. We will see that the complexity of our algorithms in Sect. 3 depends on $M = \max_{e \in E} |L(e)|$, the maximal number of lines per segment. The line graph construction is described in Sect. 2.

Line ordering optimization (Sect. 3): In its second 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 Sect 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). We formulate a concise Integer Linear Program (ILP) to solve instances of these problems.

Rendering (Sect. 5): In its third stage, LOOM draws the transit map based on the line graph from stage 1 and the ordering from stage 2. 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* (Fig. 3). Drawing the map then amounts to connecting the ports (according to the ordering computed in stage 2) and drawing the station polygons. Fig. 1, right shows a rendered transit map after layout optimization.

1.2 Contributions

- We present a new automatic map generator, called LOOM (Line-Ordering Optimized Maps), 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 describe a line-sweeping approach to extract the line graph from a set of (partially overlapping) vehicle trips as they occur in real-world schedule data.
- We phrase the crossing minimization problem in a novel way and provide an ILP formulation to solve it. Our new model resolves some issues with previous models, in particular, the restricted applicability of some algorithms to planar graphs, and the necessity of artificial grouping of crossings (which happens naturally with our approach).
- As a naive ILP formulation turns out to lead to impractically many constraints, we derive an alternative formulation yielding significantly smaller ILPs in theory and practice.
- We describe engineering techniques which allow to further simplify the line graph and hence lead to even smaller ILPs without compromising optimality of the final result.
- We evaluate LOOM on the transit network of six cities around the world. For each city, line graph construction, ILP solution and rendering together take less than 15 seconds.
- Our maps are publicly available online[1].

1.3 Related Work

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.

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

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 [1] for an overview. For example, in [2], an incremental approach is used which 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 in order 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 [13] 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 [17], edge bundling in the context of metro line map layout 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 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 [17], 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 [7], with the premises 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 [6]. In that paper, optimal layouts for path and tree networks were investigated but arbitrary graphs were left as an open problem. In [3, 4, 15], several variants of MLCM were defined and efficient algorithms were presented for some of these variants, often with a restriction to planar graphs. In [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, see Sect. 3.3) 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 [11], 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 [10]. Our adapted MLNCM problem has the same complexity as MLCM and is hence also NP-hard.

1.3.3 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 [14] or Bézier Curves [9]. See [16] 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

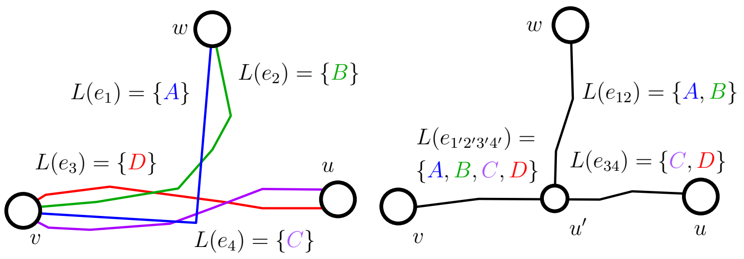

Fig. 2. Left: Input multigraph G^0 created from schedule data (GTFS). Nodes represent stations, and each edge holds a single line that occurs between two stations. There may be many overlapping edge segments. Right: Line graph G constructed from G^0 by repeatedly combining shared edge segments into a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

positions), and the minimization of line crossings or separations is not part of the problem. In particular, the resulting maps cannot be used for tiles or overlays in typical map services.

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 [8] 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.

2 LINE GRAPH CONSTRUCTION

This section describes stage 1 of LOOM: given line data, construct the line graph. We assume that the data is given in the GTFS format [12]. 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 (Fig. 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, $p_\tau(\frac{1}{2})$ returns the point we would reach after travelling on τ for 5 meters). 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 Sect. 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 $L(e_{12}) = L(e_1) \cup L(e_2)$ (Fig. 2, right). Two new non-station nodes u' and v' which mark the

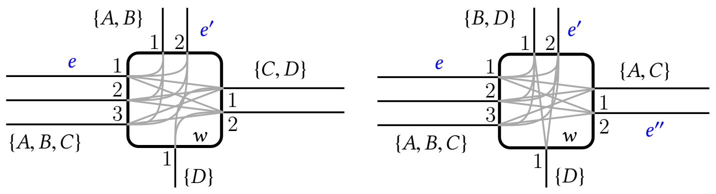

Fig. 3. Example instances. Both station polygons have 4 *node fronts*, each corresponding to an incident edge. Each node front has exactly one port (1, 2, ...) for each line traversing through its edge. Gray lines depict possible inner node connections. Left: A, B extend from e to e' over w and may introduce 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''.

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, 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 $p_{\tau_1}(i \cdot \Delta t)$ and τ_2 at each $i < n$ along the way. If $d \leq \hat{d}(e_1, e_2)$, we start a new shared segment. If $d > \hat{d}(e_1, e_2)$ and a shared segment is open, we close it. For our test datasets, we found that a Δt of 10 meters 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 Sect. 6, we used a random order.

3 LINE ORDERING OPTIMIZATION

This section describes stage 2 of LOOM, namely how to solve 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 on each $L(e)$ and allows crossings to occur anywhere on e, MLNCM only imposes a single ordering on each edge and restricts crossing events to nodes. This will prove advantageous during rendering, see Sect. 5. As the set of stations $\mathcal{S}$ is only a subset of V in our model (Sect. 1.1), we can still avoid line crossings in them.

3.1 Baseline ILP

For each edge e, there are $|L(e)|!$ many orderings, 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 $p = 1, ..., |L(e)|$ 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

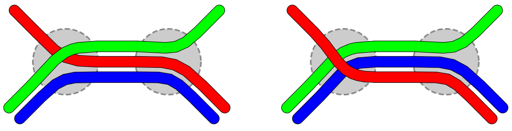
Fig. 4. Minimized crossings in the left example, but the right example better indicates line pairings.

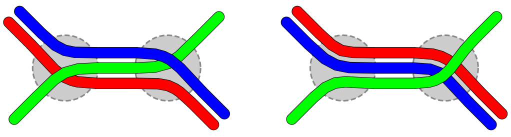
Fig. 5. Both orderings have 2 crossings, but in the right example they are done in one pass.

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' (Fig. 3, left), or they continue along different edges e' and e'' (Fig. 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 $p_e(A) < p_e(B)$, 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 Fig. 3, six such 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.

3.1.1 ILP size. For mapping lines to positions at each edge we need at most $|E|M^2$ variables and $2|E|M$ constraints. 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, 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:

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

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 Fig. 3, left. Before, we enumerated all possible positions which induce a crossing for A, B at the transition from e to e'. But it would be sufficient to have variables which 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:

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

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

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, 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, nothing changes in the argumentation. If the values are unequal, either (11) or (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. Fig. 4 and Fig. 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 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 $U(e)$:

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

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

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 (13) or (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:

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

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 Sect. 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 [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 trivially ensured in MLNCM-S (Fig. 6, left). For other nodes, however, it is not guaranteed (Fig. 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.

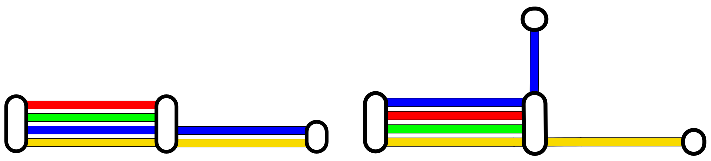

Fig. 6. Left: Periphery condition guaranteed by separation penalty. Right: Periphery condition not guaranteed by separation penalty.

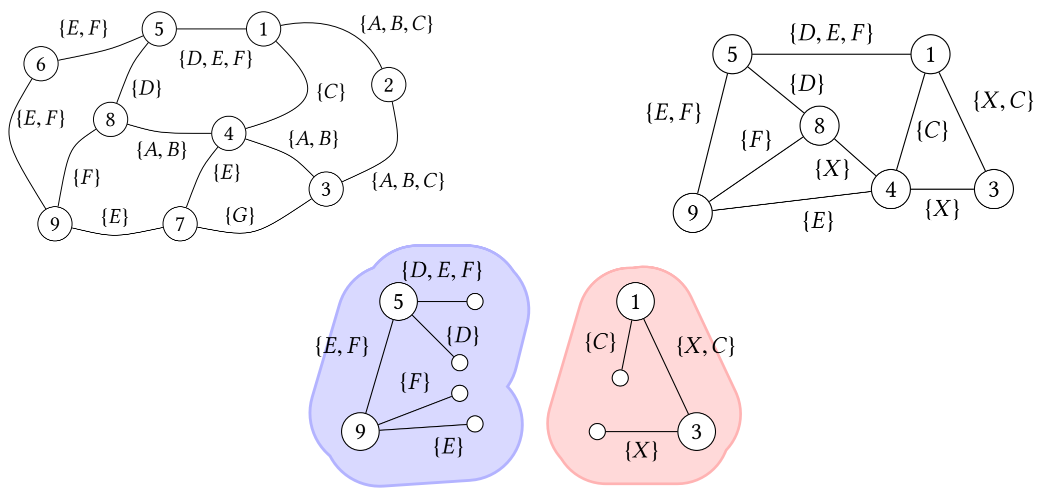

Fig. 7. Top left: line graph G with 7 lines. Bottom: core graph of G after applying pruning rules, $\{A, B\}$ was collapsed into $\{X\}$. Top right: ordering-relevant connected components of G after applying cutting rules.

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 $\hat{w}_\|$ prior to optimization and choose $w_{\mathcal{S}\times} = \hat{w}_\times$ and $w_{\mathcal{S}\|} = \hat{w}_\|$.

4 CORE GRAPH REDUCTION

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 and thus the ILP solution. In our experiments, these transformations reduced the size of the resulting ILPs by a factor between 2 and 4 and led to significantly lower solution times (see Sect. 6). We first prove Lemmata 4.1 - 4.3 and use them to derive a set of pruning and cutting rules.

Lemma 4.1. *If for some set $\mathcal{B} = \{A, B, C, \ldots\} \subseteq \mathcal{L}$ it holds for all $l \in \mathcal{B}, e \in E : l \in L(e) \Rightarrow \mathcal{B} \subseteq L(e)$, then it always exists an optimal ordering in which $A, B, C, \ldots$ are bundled next to each other with a fixed, global ordering.*

Proof. Let $L \in \mathcal{B}$ be the line that induces the minimal number of crossings and separations for some solution σ. Since all $l \in \mathcal{B}$ take the exact same path through the network, a solution can only be better than or equal to σ if it bundles all $l \neq L$ alongside L. □

Lemma 4.2. *Given 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 the optimal ordering of $L(e)$ is the same as of $L(e')$, inhibiting any crossings in v. We can thus ignore this case. (2) Lemma 4.1 does not hold and 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, 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 $v' \notin W$. Such a v' will thus 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 Sect. 3.4, optimality will also not be affected negatively, because $\deg(v)$ is always 2, implying that v is a station (Sect. 2). The same argument holds for line separations. □

Lemma 4.3. *If for some edge e all $l \in L(e)$ end in a node v or $|L(e)| = 1$, the ordering of $L(e)$ will not affect the number of orderings or separations in v.*

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

4.1 Pruning Rules

Using the lemmata from above, we may simplify the input line graph with the following pruning rules:

(Pruning rule 1) delete each node v with degree 2 and $L(e) = L(e')$, and combine the adjacent edges $e = \{u, v\}$, $e' = \{v, w\}$ into a single new edge $ee' = \{u, w\}$ with $L(ee') = L(e) = L(e')$ (Lemma 4.2).

(Pruning rule 2) collapse lines that always occur together into a single new line k (Lemma 4.1). Weight crossings with k by the number of lines it combines to avoid distorting penalties.

(Pruning rule 3) remove each edge $e = \{u, v\}$ where u and v are termini for all $l \in L(e)$ (Lemma 4.3).

We call the resulting graph the core graph of G. Fig. 7, bottom gives an example of a core graph after applying pruning rules $1 - 3$.

4.2 Cutting Rules

The core 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. 7, top right).

(Cutting rule 1) cut each edge $e = \{u, v\}$ with $|L(e)| = 1$ into two edges $e' = \{u, v'\}$ and $e'' = \{v'', v\}$ (Lemma 4.3).

(Cutting rule 2) replace each edge $e = \{u, v\}$ where v has a degree > 1 and is a terminus node for each $l \in L(e)$ with an edge $e' = \{u, v'\}$ where v' is only connected to e' (Lemma 4.3). Special care has to be taken to not make v eligible for a pruning rule (1) contraction, as this may prohibit crossings and thus compromise optimality.

4.3 Graph Untangling Rules

The pruning rules described so far may greatly help to reduce the size of the search space for the line ordering optimization and thus bring down the time required to find an optimal solution. Section 6 evaluates the effects of these rules. However, there are still some simplification situations not covered by these rules. This section will describe 4 simple graph untangling rules which further help to bring down the degrees of freedom of the line ordering optimization problem. As we will see, these untangling rules are able to completely solve the optimization problem for specific line graph instances by cutting it down to many ordering-relevant connected components with a search space size of 1. In many cases, they are able to reduce the search space to a size which can be explored by a simple exhaustive search in minimal time.

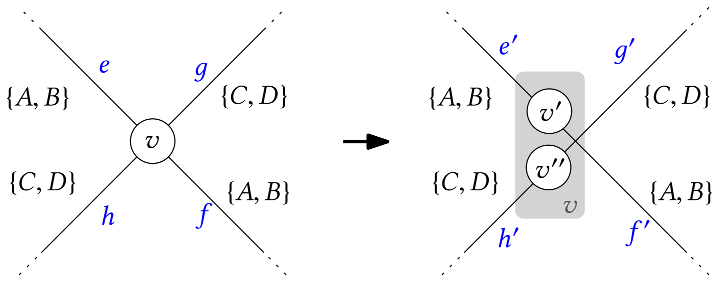

Fig. 8. Left: Full cross situation in a line graph. $\{A, B\}$ continue from e to f through node v without interfering with $\{C, D\}$ on h and f. Right: Node v has been split into two nodes v' and v'', without affecting line ordering optimality.

As these graph untangling rules always break up nodes in the original line graph into multiple nodes in the core optimization graph, we have to update pruning rule 2 from Section 4.1. There, we always contracted degree 2 nodes because Lemma 4.2 guaranteed that a same-segment crossing in these nodes could be moved to another node without affecting optimality. If we break up nodes, this may no longer be the case: degree 2 nodes in the core optimization graph may now stand for nodes of higher degree in the original line graph, and we cannot contract them blindly. Therefore, we first define $v^*(v)$ to be original line graph node we constructed v from. Note that for many nodes which were not eligible for any pruning or untangling rule, $v^*(v) = v$. Using this, we add an additional check to pruning rule 2 and state this updated pruning rule 2a it as follow:

(Pruning rule 2a) 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 $\deg(v^*(v)) \neq 2$ and $|L(e)| = L(e') > 1$, we additionally check if $w_\times(v^*(v)) \geq w_\times(v^*(u))$ and $w_\parallel(v^*(v)) \geq w_\parallel(v^*(u))$ (crossings and separations in v can be moved to u at equal or lower cost) or $w_\times(v^*(v)) \geq w_\times(v^*(w))$ and $w_\parallel(v^*(v)) \geq w_\parallel(v^*(w))$ (crossings and separations in v can be moved to w at equal or lower cost). If that is the case, combine the adjacent edges $e = \{u, v\}$, $e' = \{v, w\}$ into a single new edge $ee' = \{u, w\}$ with $L(ee') = L(e) = L(e')$.

4.3.1 *Full Crossings.*

We now consider the situation in Figure 8, 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 or f (partially) continue over into another adjacent edge, the orderings in e and f 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. We can thus state the following untangling rule:

(Untangling rule 1) For some node v in the line graph and its adjacent edges $e_1, e_2, \ldots, e_{\deg(v)-1}$, if we can identify two edges $e_a = \{v, u\}$ and $e_b = \{v, w\}$ with $L(e_a) = L(e_b)$ and both e_a and e_b do not (partially) continue into any other edge $e_i \notin \{e_a, e_b\}$, break v into two nodes v' and v''. Node v' gets connected to u with an edge e'_a and to w via e'_b, where $L(e'_a) = L(e_a)$ and $L(e'_b) = L(e_b)$. 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 8, right gives an example.

Note that this rule alone will not have any effect on the ILP sizes, as we would for example not add any constraints or variables for crossings between edge e_a and some $e_i \notin \{e_a, e_b\}$, as described in Sections 6.1 and 3.2. However, v' and v'' may now be eligible for contraction according to pruning rule 2a. Additionally, this rule may cause the optimization graph to break down into two connected components.

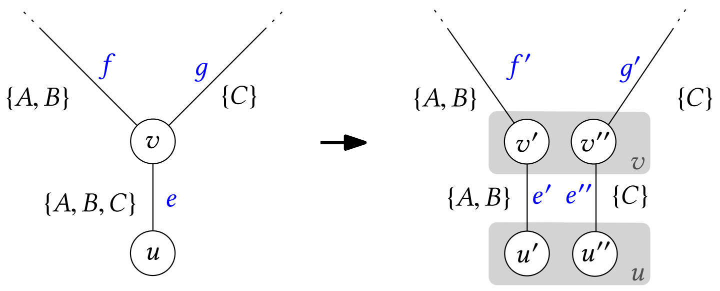

Fig. 9. Left: TODO. Right: Note that if g was the segment with the largest number of lines in the graph, we now have brought down M from 3 to 2.

4.3.2 Y Structures. Figure 9, left gives an example of an Y structure in the line graph. Here, a main leg $e = \{u, v\}$ branches at v into two minor legs 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 minor legs f and g already induces a partial optimal ordering of the lines on e, namely that $p_e(A) < p_e(C)$ and $p_e(B) < p_e(C)$. Any ordering violating these constraints would induce an unnecessary crossing (or separation) in v.

For easier notation, we say that $\rho_e^v(f)$ is the clockwise position of f in node v, beginning at edge e. For example, in Figure 9, left, $\rho_e^v(f) = 0$ and $\rho_e^v(g) = 1$.

(Untangling rule 2) For some node v, if we can identify a main leg edge $e = \{v, u\}$ where u is a terminus and which completely branches at v into $\deg(v) - 1$ minor leg edges $e_0, e_1, ..., e_{\deg(v)-2}$ such that $L(e_0) \cup L(e_1) ... \cup L(e_{\deg(v)-2}) = L(e)$ and all $L(e_i)$ are pairwise disjoint, split v and u into nodes v', v'' and u', u''. For simplicity, we assume that the minor leg edges are already sorted in ascending order by their ρ_e^v values. Nodes v' and u' are connected with an edge e', where $L(e') = L(e_{b_0})$ (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 nodes it was originally connected to via the first minor leg with e_0', where $L(e_0) = e_0'$. Similarly, all remaining branches are re-connected to v'' (Fig. 9, right).

To be able to later deduce the ordering of the original main leg e, we additionally store an ordering of the new main leg edges e' and e'', which is just the original clockwise ordering of e_{b_0} and the remainder of the minor legs. The ordering of the lines in the original edge e is then just the ordered lines (after optimization) in e', followed by the ordering 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 2 branches.

4.3.3 Partial Y Structures. A special case of Y structure can be seen in Figure 10, left. Here, e completely branches into 2 minor legs f and g at v. However, the lines of the minor legs are not completely contained in the main leg: line D continues through node v from f to g, not to e. But the ordering of f and g still induce a partial ordering of e. We call this situation a *partial* Y.

(Untangling rule 3) For some node v, if we can identify a main leg edge $e = \{v, u\}$ where u is a terminus and which completely branches at v into n minor leg edges $e_0, e_1, ..., e_{n-1}$ such that $L(e_0) \cup L(e_1) ... \cup L(e_{n-1}) \subsetneq L(e)$, split u into nodes u', u''. For simplicity, we again assume that the minor leg edges are already sorted in ascending order by their ρ_e^v values. Similar as in untangling rule 2, 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 $L(e'') = \bigcup_{i=1}^{n} L(e_i)$ (Fig. 10, right).

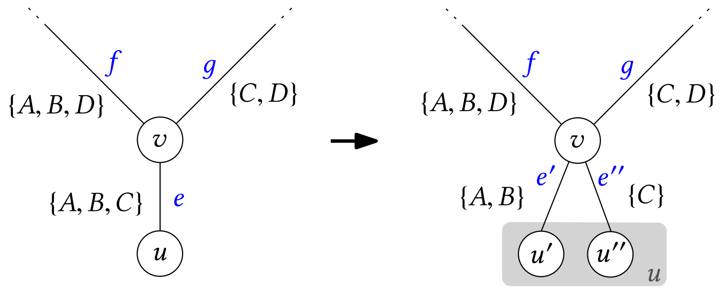

Fig. 10. Left: Partial Y structure. Right: TODO

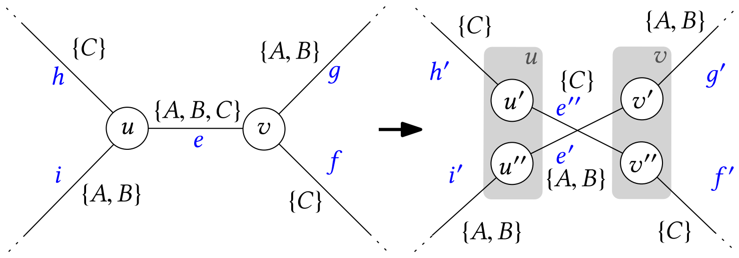

Fig. 11. Left: TODO. Right: TODO

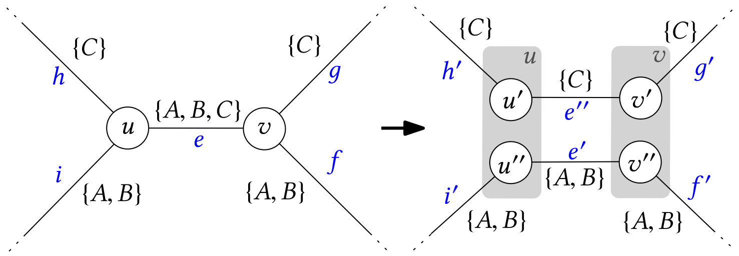

Fig. 12. Left: TODO. Right: TODO

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 main leg e.

4.3.4 Dog-Bone Structures. A more complex structure that is commonly found in real-world input data is depicted in Figure 11, left. Two line threads (in the example, $\{C\}$ and $\{A, B\}$) on two segments 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 this dog-bone structures. An example of such a structure in the real-world map of the Stuttgart light rail network is shown in Figure 13.

Figure 11, it is obvious 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 the ordering of the two threads at u and v imposes a lower bound on the sum of crossings in u and v. In Figure 11, because $\rho^u_e(i) < \rho^u_e(h)$ and $\rho^v_e(g) < \rho^v_e(f)$, two crossings between C and A, as well as C and B in either u or v are unavoidable, regardless of the

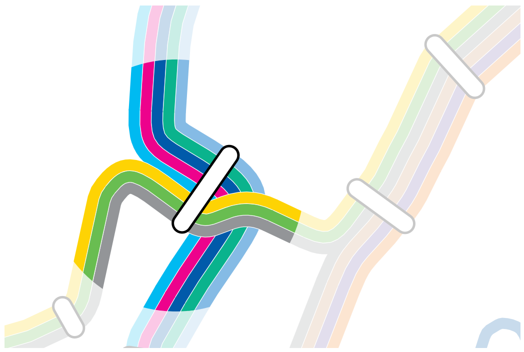

Fig. 13. Dog bone structure in the map of the Stuttgart light rail system. Untangling reduces the number of possible orderings in the highlighted central segment of the network from $8! = 40,320$ to $4! \times 3! = 144$.

actual orderings in h, i, g or f. In Figure **??**, because $\rho_e^u(i) < \rho_e^u(h)$, but $\rho_e^v(f) > \rho_e^v(g)$, 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.

We transform structures like this with the following rule:

(Untangling rule 4) If some main leg edge $e = (u, v)$, with a degree $\deg(u) = \deg(v) = n \geq 3$ branches at u into $\deg(u) - 1$ minor left leg edges $e_0^u, e_1^u, ..., e_n^u$ and at v into n minor right leg edges $e_0^v, e_1^v, ..., e_n^v$ and if there is a mapping $m(i) \mapsto j$ with $i, j \in [0, n]$ such that $L(e_i^v) = L(e_{m(i)}^u)$ (e branches into the exact same left and right branches), we say e is a *dog bone*. We additionally require that $\bigcup_{i=0}^n L(e_i^u) = \bigcup_{j=0}^n L(e_j^v) = L(e)$ (the combined lines of all left and of all right minor legs are exactly the same as the lines contained in the main leg) and that for both the right and left minor legs, both the $L(e_i^v)$ and the $L(e_j^u)$ are pairwise disjoint (there are no lines continuing through u and v to any other edge than e). We then untangle both u and v like u and v 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 11, right gives an example.

Just like with untangling rule 2 and 3, 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 in u, or on the ordering position of the right minor leg e_0^v in v. If the two orderings are inverse, that is if for all $1 \leq i < n$ it holds that $\rho_e^u(e_0) < \rho_e^u(e_i) \Rightarrow \rho_e^u(e_{m(0)}) > \rho_e^u(e_{m(i)})$, it does not matter, because there are no unavoidable crossings we have to consider (Fig. 12, right). However, if that is not the case, and an unavoidable crossing occurs (Fig. 11), right), we have to base the ordering on the node with smaller crossing penalty to not compromise optimality of the final line ordering. For example, in Figure 11, right, $\rho_e^u(h) > \rho_e^u(i)$. If we assume that $w_\times(u) > w_\times(v)$, 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 $w_\times(u) < w_\times(v)$, the ordering would be set to (e', e''), and the crossings would appear in u.

4.3.5 *Partial Dog-Bone Structures.*

For completeness, we note that just as with Y structures, there may also be partial dog-bone structures (Fig. 14, left). These are dog-bone structures where *one* of the nodes u and v fullfills the criteria described in Section 4.3.4, and the other node fullfills the criteria described for v in partial Y structures (Sec. 4.3.3) (that is, the main leg branches at v

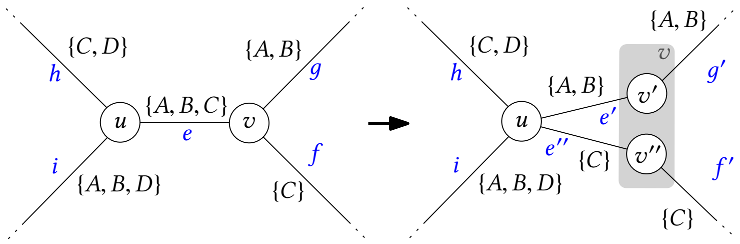

Fig. 14. Left: Partial dog bone structure. Right: TODO

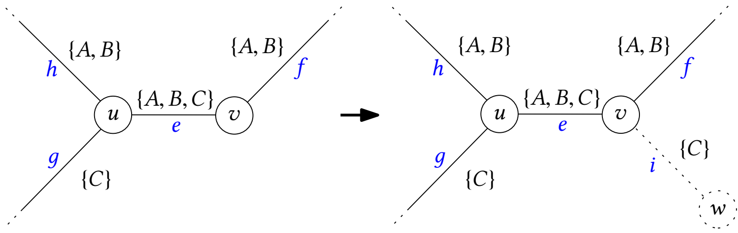

Fig. 15. Left: Stump structure. Right: Stump structure transformed into dog bone.

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 main leg).

(Untangling rule 6) In cases like the one described above, we only break up the node fullfilling the criteria described in Section 4.3.4, just like we broke up only u in Section 4.3.3 (Fig. 14, right). By applying untangling rule 4 afterwards, we can then detach the stump line from the main leg.

4.3.6 Stump Structures. It remains to show that an additional class of structures, which we call *stumps*, can be transformed into a dog bone structure with minimal changes to the graph. Figure 15, left gives an example. Line C on minor leg g attaches itself to the main leg e in node u, but already terminates in node v, $\{C\}$ is a stump in e. Regardless of the ordering of A and B in h, e and f, the 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.

(Untangling rule 6) We can easily transform a situation like this into a standard dog bone by introducing an additional dummy edge i with the same lines as g from v to an additional dummy node w in such a way that $\rho_e^v(i) = \deg(u) - 1 - \rho_e^u(g)$ (i will be at a position in v that is inverse to the position of g in u).

4.4 Complexity of Core Graph Reduction

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 and reduced?

We first consider the complexity of the untangling rules and assume that the lines in each $L(e)$ are already ordered (for example, by some internal line id). We denote the maximum degree of our input line graph G as D. We can safely remove nodes with degree 1 from the line graph prior to optimization and thus assume that $2|E|$ is an upper bound for $|V|$. Finding an edge eligible for any of the untangling rules described in Section 4.3 is a matter of iterating over all edges or all nodes, an thus always takes $O(|E|)$. Each rule breaks up an edge e into two edges e' and e'', which can only be done at most M times in the worst case (if $|L(e)| = M$ and we only detach a single line per untangling rule application). No untangling rule increases the number of lines per edge, so a single untangling rule can be applied at most $O(|E|M)$ times.

In untangling rule 1 (full crossings), we have to check $\binom{D}{2} < D^2$ partners of adjacent edges for line equivalency in the worst case, which can always be done in $O(M)$ because we assume the $L(e)$ to be ordered.

In untangling rule 2 (Y structures), we have to check for the non-terminus node if the $L(e)$ of all minor leg edges are completely contained in the main leg, which can be done in $O(MD)$, again assuming the $L(e)$ are ordered. We also have to check of the minor leg edges are pairwise disjoint, which takes $O(MD^2)$ in the worst case.

In untangling rule 4 and 5 (dog bone structures), it is easy to see that we have to apply the same checks as for rules 2 and 3, but we also have to establish the mapping m between the left and the right legs, which can be considered an intersection problem and solved in $O(M\Delta(G))$ if we assume that the adjacency lists are already ordered in clockwise fashion based on their outgoing angle.

Untangling rule 6 (stump structures) has the same complexity as rule 4 and 5, as the identification has the same complexity as there and the extension by dummy node and edge can be done in constant time.

A single round of each untangling rule can thus always be done in $O(|E|MD^2)$.

Now we consider the complexity of the pruning rules. For each contraction, we have to check if $L(e) = L(e')$, which can be solved in $O(M)$ as we assumed that $L(e)$ and $L(e')$ are sorted and can at least have size M. Pruning rule 1 can thus be applied in $O(|E|M)$, as we have to contract at most $2|E|$ nodes. Pruning rule 2 is a matter of doing at most $\binom{M}{2} < M^2$ depth-first searches in the line graph, which can be done in $O(|E|M^2)$ (again assuming $2|E|$ as an upper bound for $|V|$). Pruning rule 3 is a matter of checking each line in each edge (u, v) if it terminates in either u or v, which can again be done in $O(|E|M)$ as we have to check at least M lines per edges, twice (for u and v).

The complexity of cutting rule 1 is trivially $O(|E|)$, and for cutting rule 2 we have to again check every line in every edge, which can be done in $O(|E|M)$.

We may apply all of the rules in the following fashion:

(1) Set n to M.
(2) Apply pruning rule 2 (collapsing of line partners).
(3) Set n to n - 1
(4) Apply pruning rule 1 and 3, apply all cutting rules.
(5) Apply untangling rules 1 - 6.
(6) While n > 0, goto 3.
(7) Apply pruning rule 1 and 3, apply all cutting rules.

First, we note 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.

Second, we note that pruning rule 1 and 3 always *decrease* the number of edges. In the worst case, the cutting rules may double the number of edges. The untangling rules may increase the number of edge by a factor of M in the worst case, so if $|E|$ is the number of edges in the input line graph, the maximum number of edges $|E'|$ in any intermediate optimization graph will be $O(|E|M)$. We assumed that each edge $L(e)$ and each adjacency list in the input line graph was sorted. The former can be sorted in $O(|E|M \log M)$, the latter in $O(|E|D \log D)$.

The worst case complexity of the entire core graph reduction process, including untangling is thus $O(|E|M^3D^2)$. We note, however, that in practice, both M and D are usually very small. For our testing datasets, the maximum M was 9 (Table 1), and D was always below 10. For all practical purposes, we are confident that both D and M can be considered a constant factor.

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 Fig. 16.

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 node v now has $\deg(v)$ node fronts (Fig. 16.2). 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 large 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 Fig. 1.

5.1 Station Merging

[TODO: This was asked during the conference: what if stations touch during expansion? MERGE THEM]

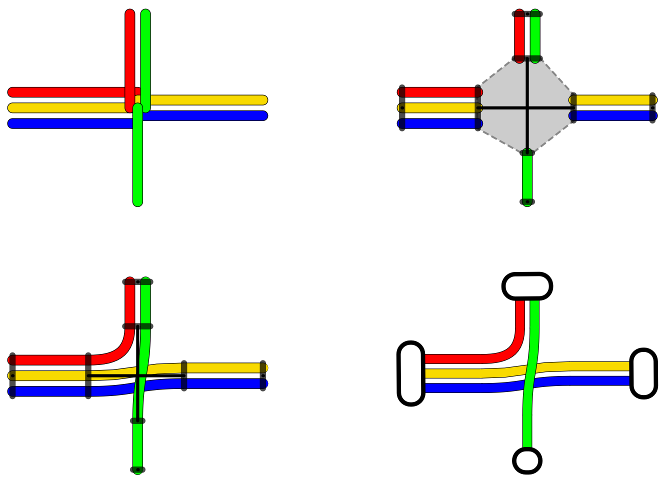

Fig. 16. The four steps of rendering a given line graph: (1) render ordered lines as edges, (2) free node area, (3) render inner connections, (4) render station overlays.

Table 1. Line graph dimensions and line ordering search space sizes $|S|$ for our testing datasets with extraction times from GTFS. $\mathcal{S}$ 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.

	t_{extr}	$\lvert\mathcal{S}\rvert$	$\lvert V\rvert$	$\lvert E\rvert$	$\lvert\mathcal{L}\rvert$	M	$\lvert S\rvert$
Freiburg	0.7s	74	80	81	5	4	$6{\times}10^{13}$
Dallas	3s	108	117	118	7	4	$1{\times}10^{20}$
Chicago	13.5s	143	153	154	8	6	$4{\times}10^{33}$
Stuttgart	7.7s	192	219	229	15	8	$3{\times}10^{103}$
Turin	4.9s	339	398	435	14	5	$1{\times}10^{85}$
New York	3.7s	456	517	548	26	9	$2{\times}10^{267}$

Table 2. Core graph dimensions and line ordering search space sizes $|S|$ for our testing datasets after applying pruning rules.

	$\lvert V\rvert$	$\lvert E\rvert$	$\lvert\mathcal{L}\rvert$	M	$\lvert S\rvert$
Freiburg	20	21	5	4	55k
Dallas	24	24	7	4	$2{\times}10^{6}$
Chicago	23	24	8	6	$5{\times}10^{9}$
Stuttgart	50	58	15	8	$2{\times}10^{38}$
Turin	91	124	14	5	$5{\times}10^{40}$
New York	110	138	23	9	$6{\times}10^{92}$

6 EVALUATION

We tested LOOM on the public transit schedules of six cities in Europe and the US: 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 first considered two versions of the line graph: the baseline graph and the core graph after pruning rules were applied. After that, we evaluated the effects of full graph untangling. For each graph, we first considered 6 heuristical optimization methods: exhaustive search (E), exhaustive search with line separation penalty (E+S), steepest hill climbing (H), steepest hill climbing with line separation penalty (H+S), simulated annealing (A) and simulated annealing with separation penalty (A+S). After that, we considered 3 ILP variants: the baseline ILP (B), the improved ILP (I) and the improved ILP with added separation penalty (I+S). For each ILP, we evaluated three solvers: the GNU Linear Programming Kit (GLPK), the COIN-OR CBC solver and gurobi (GU). As most of the datasets still only had one connected component after applying the splitting rules described in Sect. 4 on the core optimization graph, we did only evaluate their application in the context of graph untangling.

For each node v, the penalty for a crossing between edge pairs ($\{A, B\}$ in Fig. 3, left) was $4 \cdot \deg(v)$, for other crossings ($\{A, B\}$ in Fig. 3, right) it was $\deg(v)$. The line separation penalty was $3 \cdot \deg(v)$. We found that these penalties produced nicer maps than a uniform penalty. This would imply $w_{\mathcal{S}\times} = 4 \cdot \max_{v \in V} \deg(v)$ and $w_{\mathcal{S}\parallel} = 3 \cdot \max_{v \in V} \deg(v)$. 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 $3 \cdot \deg(v)$

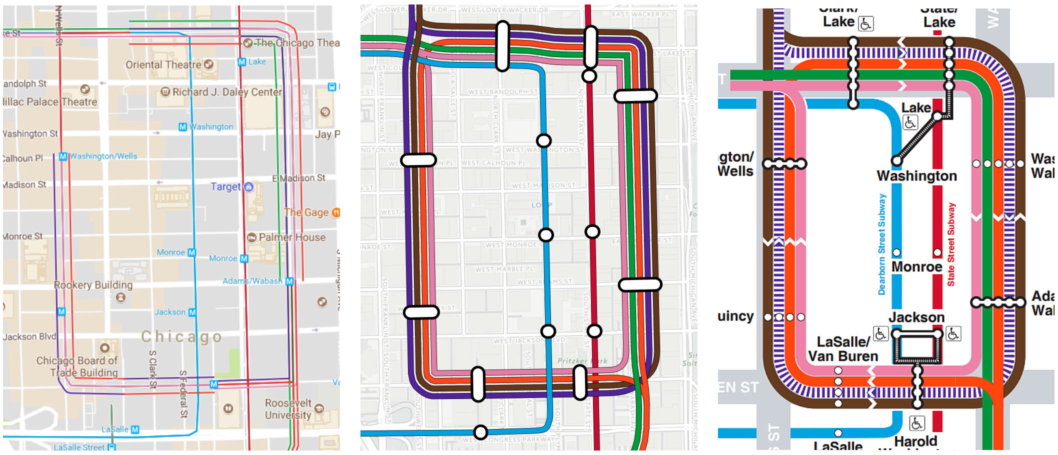

Fig. 17. Left: Google transit map cutout for Chicago. Center: Same area in our automatically generated map. Right: Official CTA map for the same area. Note the near-perfect match of the line orderings in the official map and our map.

(normal crossing) or $12 \cdot \deg(v)$ (edge-pair crossing). Similarly, in-station line separations where punished with $w_{\mathcal{S}\|}$ if $\deg(v) = 2$ and $9 \cdot \deg(v)$ 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+S) and thus depended on the solver and the input order in (B) and (I).

6.1 Performance of Optimization Heuristics

We first evaluated the results of 3 baseline heuristics for the line ordering optimization: an exhaustive search (E), steepest hill climbing (H) and simulated annealing (A).

The exhaustive search (E) always explored the entire search space S and took the optimal line ordering solution. Even for small input graphs, the size of S was immense. For the Freiburg dataset, the size of S was $\sim 6 \times 10^{13}$, for Stuttgart, it was $\sim 3 \times 10^{103}$.

Our steepest hill climbing approach (H) always explored the entire neighborhood of the current ordering state in each iteration and chose the one closest to the solution.

In contrast, our simulated annealing approach (A) selected a neighbor at random and stopped if the value of the target function had not changed for $k = 5,000$ iterations. At state s, we chose the randomly selected neighbor as the next state s' if it improved the overall target function $\theta(s)$ (that is, if $\theta(s') < \theta(s)$), or else with probability $P(s, s', T) = e^{-(\theta(s')-\theta(s))/T}$, where T is the current annealing temperature. At iteration i, we set $T = 1000/n$, allowing to system to escape early local optima.

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

6.1.1 Solution Times. Table 7 shows the results of the heuristic ordering optimizations for 3 of our 6 datasets: Freiburg, Chicago and Stuttgart. On the baseline graph (without reduction to core optimization graph or untangling), an exhaustive search (E) was infeasible even for the small dataset of Freiburg. If we assume that we can check 10,000 ordering configurations per second, an exhaustive search for Freiburg would still take about 70,000 days on the baseline graph.

Final ordering configurations and graph scores of both simulated annealing (A) and steepest hill climbing (H) were comparable, but steepest hill climbing generally 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 (H) 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 than both (H) and (A). With added line separation penalty (E+S, H+S, A+S), solution times where generally slightly larger, even if the number of iterations was smaller. This is because of the additional overhead of checking each ordering configuration for line separations.

6.1.2 Effects of Core Graph Reduction. If the input line graph was first reduced to the core optimization graph with pruning rules 1-3 (Sect. 4.1), an optimal solution could be found with an exhaustive search (E) for Freiburg in 4.2 seconds. For (A), (A+S), (H) and (H+S), prior core graph reduction enhanced the final graph scores by a factor of ~ 3.5 on average. This was to be expected, as core graph reduction lead to a significant reduction of the search space size (for example, core grap reduction reduced the solution space size of Freiburg by 9 orders of magnitude). For (H) and (H+S), soluction times were on average ten times faster on the core optimization graph than on the baseline graph (with a large improvement for H on the Freiburg dataset). Interestingly, solution times for simulated annealing did not improve much or got worse after core graph reduction, despite a lower number of iterations. This is because before core graph reduction, 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 core graph reduction, as core graph reduction only leaves intersection nodes in the graph. Checking the scores of these larger nodes takes more time, which leads to larger average iteration times.

6.1.3 Optimality of Results. With prior core graph reduction and added separation penalty, the local optima found by our simulated annealing approach where on average a factor of 2.16 higher than the optimal solutions (which can be seen in Table 6). The average Freiburg score for (A+S) on the core graph (70.6) was closest to the optimal score (48), but for Stuttgart, the average (A+S) score on the core graph (519.3) was 3.33 times higher than the optimal score (156).

[TODO: Times for hillc when Stuttgart eval is finished]

6.2 Comparison of ILP Variants

Table 8 shows the results of the ordering optimizations with ILPs for 4 of our 6 datasets: Chicago, Stuttgart, Turin and New York. Tests were run on an Intel Core i5-6300U machine with 4 cores à 2.4 GHz and 12 GB 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 Solution Times. With our improved ILP (I), the optimal orderings on the core graph could be found in under 50 milliseconds with gurobi, and in under 1 second with CBC, on any dataset. If line separation was also punished (I+S), the ILP could be solved on the core graph in under 2.5 seconds with gurobi for any dataset, and in under 1 minute with CBC. Although the ILPs for (I+S) were only slightly larger than for (I), optimization on the core graph took 28 times longer on average with the fastest solver.

6.2.2 Effects of Core Graph Reduction. On the baseline graph, (B) could not be optimized for all datasets except Turin with gurobi, and only after core graph reduction was a solution for Stuttgart and Chicago found in under 12 hours. As expected in Sect. 4, core graph reduction made the ILPs significantly smaller. On average, the number of rows decreased by 61 % and the number of columns by 59 % for (I). For (I+S), the decrease was 62 % and 60 %, respectively. With the fastest solver and

Table 3. Dimensions of line graphs with only pruning rules applied (core graph) and after full untangling was applied (untangled graph), as well as time needed for untangling.

	Core graph				Untangled graph							Largest component			
	$\lvert V_c\rvert$	$\lvert E_c\rvert$	M_c	$\lvert S_c\rvert$	t	$\lvert V\rvert$	$\lvert E\rvert$	M	$\lvert S\rvert$	C	C^1	$\lvert\hat{V}\rvert$	$\lvert\hat{E}\rvert$	$\hat{M}$	$\lvert\hat{S}\rvert$
Freiburg	20	21	4	55k	3ms	22	19	4	9.2k	3	2	18	17	4	9.2k
Dallas	24	24	4	2×10^6	11ms	36	27	4	2.3k	9	8	12	11	4	2.3k
Chicago	23	24	6	5×10^9	5.8ms	24	22	6	1×10^9	3	2	18	18	6	1×10^9
Stuttgart	50	58	6	2×10^{38}	15ms	58	52	6	7×10^{22}	7	5	24	24	6	1×10^{12}
Turin	91	124	5	5×10^{40}	15ms	157	134	5	2×10^{31}	26	23	70	71	5	3×10^{29}
New York	110	138	9	6×10^{92}	20ms	107	93	6	3×10^{36}	17	13	62	64	6	5×10^{34}

the improved ILP (I), core graph reduction lead to speedup by a factor between 4 for Chicago, and 2 for New York. For (I+S), this speedup was between 22.4 for Stuttgart, and 2.5 for Turin.

6.3 Effects of Graph Untangling

To measure the effects of our graph untangling rules described in Section 4.3, we re-ran all of our 6 heuristical optimizations and our 3 ILPs on the untangled core optimization graph of our datasets. We followed the algorithm described in XX and thus also applied cutting rules.

6.3.1 Effect on Graph Size. Table 3 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 solution space size $|S|$. Additionally, we counted the number of connected components C, the number C^1 of connected components with $|S| = 1$ (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 S, sometimes dramatically. For New York, the search size space was reduced by 56 orders of magnitude compared to the core 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 just $\sim 9,200$, which can easily be explored with an exhaustive search.

The number of connected components retrieved from graph untangling was always larger than 2. For 3 of our datasets (Freiburg, Dallas and Chicago), all but one connected component hat a solution size of exactly 1, meaning that they did not need any ordering optimization. For the remaining networks, the search space of the largest connected component was again $2-10$ orders of magnitudes 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 core optimization 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, because breaking up edges in our untangling rules always reduces the number of lines per edge.

6.3.2 Effect on ILP Solution Times. Table 4 shows the solution times of some of our ILPs after applying our untangling rules, compared to our initial approach of just applying pruning rules. If

Table 4. Impact of graph untangling on selected ILP solution times.

		Core graph			Untangled graph		
		GLPK	CBC	GU	GLPK	CBC	GU
Freiburg	I	8 ms	?	10 ms	5 ms	?	10 ms
	I+S	10 ms	?	13 ms	7 ms	?	13 ms
Chicago	I	0.8 s	?	10 ms	0.1 s	?	11 ms
	I+S	23 s	?	0.3 s	18 s	?	0.2 s
Stuttgart	I	8 s	?	36 ms	0.2 s	?	30 ms
	I+S	—	?	2.1 s	21.5 s	?	0.3 s
New York	I	—	?	0.1 s	2 m	?	80 ms
	I+S	—	?	1.5 s	4 m	?	0.5 s

multiple components had to be optimized after untangling, optimization was done iteratively. For datasets where this was the case, solution times in Table 4 are the summed solution times of all ILP solves.

For Freiburg and Chicago, an improvement noticable with GLPK. With gurobi, none of the solution times were better than the core graph solution times with (I) and (I+S) for Freiburg and Chicago. For Chicago, the solution time for (I) even went slightly up. This was because of the additional overhead of solving multiple ILPs, one for each connected component with $|S| > 1$.

However, graph untangling gave signficant performance gains for larger networks if we added separation penalties. The solution time of (I+S) on New York and Stuttgart was > 12 h with GLPK on the core graph, but after graph untangling, the ILPs could be solved in 21.5 s for Stuttgart, and 4 m for New York. For Stuttgart, solving (I+S) with gurobi was 7 times faster after graph untangling, and for New York, it was 3 times faster.

6.3.3 Effect on Baseline Heuristics. We also evaluated the effect of graph untangling on our 6 baseline heuristics from Section 6.1. In addition to the time needed for the optimization, we also measured the effect of untangling on the final graph scores. Table 5 gives an overview of the effects of graph untangling for (E+S), (H+S) and (A+S) on our datasets Freiburg, Chicago and Stuttgart.

For Freiburg, applying graph untangling brought down the solution time of a simple exhaustive search to 200 ms. This is evidence that even for modestly complex real-world line graphs, an ILP may not be necessary to find a solution in acceptable time. For all our other datasets, however, exhaustive search was not able to find a solution in under 12 h, even after graph untangling.

Generally, all our baseline heuristics performed better after applying graph untangling, both in terms of running time and in terms of the final graph score. For Stuttgart, for example, the final graph score averaged from 50 runs went down from 519.3 on the core graph to 328 on the untangled graph. However, this was still over 2 times bigger than the optimal score (156).

6.4 Comparison to Manually Designed Maps

We also 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

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

Table 5. Impact of graph untangling on selected baseline heuristic solution times and graph scores.

		Core graph				Untangled graph			
		t	×	||	pen.	t	×	||	pen.
Freiburg	E+S	2.5s	6	0	48	0.2s	6	0	48
	H+S	4ms	7.1	1.1	70.6	3ms	6.38	0.9	70.1
	A+S	0.1s	6.9	1.2	70.9	80ms	5.9	0.8	68.7
Chicago	E+S	—	—	—	—	—	—	—	—
	H+S	0.5s	21.6	3.2	131.3	0.4s	20.84	3.1	123.5
	A+S	0.4s	21.3	3.1	133.6	0.4s	20.9	3.12	126.7
Stuttgart	E+S	—	—	—	—	—	—	—	—
	H+S	?m	?	?	?	0.4s	56.3	8.68	325.7
	A+S	0.7s	65	17.1	519.3	0.6s	54.88	9.2	328

Table 6. Comparison of the line orderings in our maps and in manually designed official maps published by transportation authorities. For the official maps, we hand-counted the number of crossings (×) and separations (||) and calculated the score 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.

	Official map			Our map			
	×	||	pen.	×	||	pen.	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

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 (Sect. 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 score 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. Fig. 17 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 4 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 4 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

http://loom.informatik.uni-freiburg.de/officialmaps/cta.pdf
http://loom.informatik.uni-freiburg.de/officialmaps/vvs.pdf

Table 7. Dimensions, solution times and final graph scores (pen) for Chicago, Stuttgart, Turin and New York and 6 baseline heuristics for the line ordering problem: exhaustive search with (E+S) and without (E) separation penalty, steepest hill climbing with (H+S) and without (H) separation penalty and simulated annealing with (A+S) and without separation penalty (A), all with or without reduction to the core graph. $|S|$ 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, *pen* 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 6.

		On baseline graph						On core graph					
		$\|S\|$	t	iters	$\times$	$\|\|$	pen	$\|S\|$	t	iters	$\times$	$\|\|$	pen
Freiburg	E		—	—	—	—	—		2s	55k	5	2	30
	H		22ms	9.4	13.8	10.1	163.8		5ms	2	5.6	1.8	44.8
	A	$6{\times}10^{13}$	45ms	7.2k	13.8	10.9	168.7	55k	80ms	5.9k	5.8	2	43.9
	E+S		—	—	—	—	—		2.5s	55k	6	0	48
	H+S		30ms	9.5	12.7	4.1	188.4		4ms	4.1	7.1	1.1	70.6
	A+S		0.5s	7.6k	12.1	4.7	195.6		0.1s	5.8k	6.9	1.2	70.9
Chicago	E		—	—	—	—	—		—	—	—	—	—
	H		1.3s	17.4	41.7	37.6	481.4		0.5s	6.8	20.4	7.4	125.6
	A	$4{\times}10^{33}$	0.1s	12k	45	39	512.6	$5{\times}10^{9}$	0.4s	8.7k	20.1	7.1	120.6
	E+S		—	—	—	—	—		—	—	—	—	—
	H+S		2.1s	16.6	38.5	13.3	463.3		0.5s	6	21.6	3.2	131.3
	A+S		0.2s	15.1k	40.7	15	556.2		0.4s	7.9k	21.3	3.1	133.6
Stuttgart	E		—	—	—	—	—		—	—	—	—	—
	H		5.7m	52.4	127	114.6	1648.7		1.8m	20.3	65.4	36.8	485.78
	A	$3{\times}10^{103}$	0.5s	20k	139.5	128.6	1749.8	$2{\times}10^{38}$	0.4s	10.1k	67.2	43.4	548.9
	E+S		—	—	—	—	—		—	—	—	—	—
	H+S		?	?	?	?	?		?	?	?	?	?
	A+S		0.8s	25.6k	123.3	55.7	1819.3		0.7s	12.5k	65	17.1	519.3

required to improve the readability. The results can be seen in Table 6. For Dallas, our ILP found a single (trivial) line swap that prevented a line separation at no cost and lowered the penalty 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, 4 line swaps could reduce both the number of crossings and the number of separations and lower the penalty 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 lead to results that closely resemble the esthetics of manually designed maps.

Table 8. Dimensions and solution times for Chicago, Stuttgart, Turin and New York and our three ILPs: baseline (B), improved (I), and with line separation penalty (I+S), with or without reduction to the core graph, solved with GLPK, COIN-OR CBC and gurobi (GU). A time of — means we aborted after 12 hours. The last two columns show the number of crossings (×) and separations (||) after optimization. [TODO: The number of crossings and separations for New York must be updated] Optimal graph scores can be found in Table 6.

.

		On baseline graph				On core graph					
		rows×cols	GLPK	CBC	GU	rows×cols	GLPK	CBC	GU	×	||
Freiburg	B	?×?	?	?	?	?×?	?	?	?	?	?
	I	?×?	?	?	?	?×?	?	?	?	?	?
	I+S	?×?	?	?	?	?×?	?	?	?	?	?
Chicago	B	41k×861	—	—	—	8.2k×266	—	47m	2m	22	4-7
	I	1.4k×982	9s	1s	41ms	394×285	0.8s	0.1s	10ms	22	4-7
	I+S	1.9k×1.2k	47m	19s	1.8s	505×338	23s	3.8s	0.3s	27	0
Stuttgart	B	224k×2.4k	—	—	—	44k×950	—	—	10h	60	11
	I	4.1k×2.8k	—	3.5s	0.1s	1.5k×1k	8s	0.2s	36ms	60	7-15
	I+S	5.6k×3.5k	—	2m	47s	2.1k×1.3k	—	36s	2.1s	64	2
Turin	B	24k×2.1k	—	—	14m	13k×1k	—	—	2m	79	6
	I	3.3k×2.4k	2m	0.6s	0.1s	1.6k×1.1k	16s	0.3s	41ms	79	6-10
	I+S	4.3k×2.9k	—	14s	1s	2k×1.4k	—	4.2s	0.4s	81	2
New York	B	229k×5.2k	—	—	—	96k×2.3k	—	—	—	—	—
	I	8.6k×6k	—	1.8s	0.2s	3.7k×2.5k	—	0.7s	0.1s	127	6-14
	I+S	12k×7.4k	—	2.5m	12s	4.9k×3.2k	—	50s	1.5s	132	2

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. The whole pipeline (including line graph construction from GTFS schedule data, line ordering optimization and rendering) took less than 15 seconds for all considered inputs.

Compared to two baseline optimization heuristics (steepest hill climbing and simulated annealing), our ILPs produced maps with a final score that was on average XX times better then the results steepest hill climbing, and XX times better than the results of simulated annealing. However, we showed that even for 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 minimal time, if graph untangling is applied first.

Since the line graph construction required more time than the subsequent ILP solution 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 Sect. 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. Another open problem is that of overlapping station nodes on very small resolutions.

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.

Lastly, 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 (01 Jul 2015), 601–632. https://doi.org/10.1007/s10707-014-0222-6

[2] Mahmuda Ahmed and Carola Wenk. 2012. Constructing Street Networks from GPS Trajectories. In *Algorithms – ESA 2012*, Leah Epstein and Paolo Ferragina (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 60–71.

[3] Evmorfia Argyriou, Michael A Bekos, Michael Kaufmann, and Antonios Symvonis. 2008. Two polynomial time algorithms for the metro-line crossing minimization problem. In *International Symposium on Graph Drawing*. Springer, 336–347.

[4] Evmorfia N Argyriou, Michael A Bekos, Michael Kaufmann, and Antonios Symvonis. 2010. On Metro-Line Crossing Minimization. *J. Graph Algorithms 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 fourteenth symposium on Computing: the Australasian theory-Volume 77*. Australian Computer Society, Inc., 49–56.

[6] Michael A Bekos, Michael Kaufmann, Katerina Potika, and Antonios Symvonis. 2007. Line crossing minimization on metro maps. In *International Symposium on Graph Drawing*. Springer, 231–242.

[7] Marc Benkert, Martin Nöllenburg, Takeaki Uno, and Alexander Wolff. 2006. Minimizing intra-edge crossings in wiring diagrams and public transportation maps. In *International Symposium on Graph Drawing*. Springer, 270–281.

[8] Anton Dubreau. 2016. Transit Maps: Apple vs. Google vs. Us. https://medium.com/transit-app/transit-maps-apple-vs-google-vs-us-cb3d7cd2c362.

[9] Martin Fink, Herman Haverkort, Martin Nöllenburg, Maxwell Roberts, Julian Schuhmann, and Alexander Wolff. 2012. Drawing metro maps using Bézier curves. In *International Symposium on Graph Drawing*. Springer, 463–474.

[10] Martin Fink and Sergey Pupyrev. 2013. Metro-Line Crossing Minimization: Hardness, Approximations, and Tractable Cases.. In *Graph Drawing*, Vol. 8242. 328–339.

[11] Martin Fink and Sergey Pupyrev. 2013. Ordering metro lines by block crossings. In *International Symposium on Mathematical Foundations of Computer Science*. Springer, 397–408.

[12] GTFS. [n. d.]. General Transit Feed Specification. https://developers.google.com/transit/gtfs.

[13] Danny Holten and Jarke J Van Wijk. 2009. Force-Directed Edge Bundling for Graph Visualization. In *Computer graphics forum*, Vol. 28. Wiley Online Library, 983–990.

[14] Seok-Hee Hong, Damian Merrick, and Hugo AD do Nascimento. 2006. Automatic visualisation of metro maps. *Journal of Visual Languages & Computing* 17, 3 (2006), 203–224.

[15] Martin Nöllenburg. 2009. An improved algorithm for the metro-line crossing minimization problem. In *International Symposium on Graph Drawing*. Springer, 381–392.

[16] Martin Nöllenburg. 2014. A survey on automated metro map layout methods. *Tech. Rep.* (2014).

[17] Sergey Pupyrev, Lev Nachmanson, Sergey Bereg, and Alexander E Holroyd. 2011. Edge routing with ordered bundles.. In *Graph Drawing*, Vol. 7034. Springer, 136–147.