Automatic Generation of Geographically Accurate Transit Maps

Hannah Bast[1], Patrick Brosi[1], and Sabine Storandt[2]

[1] University of Freiburg (Germany)
{bast,brosi}@informatik.uni-freiburg.de
[2] JMU Würzburg (Germany)
storandt@informatik.uni-wuerzburg.de

Abstract. We present LOOM, an algorithm for the fully automatic generation of esthetically pleasing and geographically accurate transit maps. The input to our algorithm is data about the lines of a given transit network, namely for each line, the sequence of stations it visits and the geographical course the vehicles of this line take. We evaluate our algorithms on the GTFS data of six cities around the world. The construction process is fast (less than 30 seconds, even for large cities), and the resulting transit maps are esthetically pleasing. Indeed, they are surprisingly close to hand-made transit maps and superior to the automatic transit maps provided by Google Maps. Previous research work has focused on minimizing the number of line crossings irrespective of the geographical course of the lines and without producing actual maps as an end result.

1 Introduction

Cities with a subway or tram network usually have an iconic map which illustrates the network and is posted at all stations. Figure 1 provides an excerpt from (our version of) the map of the light rail network of the city of Stuttgart. These maps have to satisfy the following main criteria:

1. They should accurately depict the topology of the network from the point of view of a public transit user: which lines are available, which stations do they serve in which order, and which transfers are possible.

2. The map should be neatly arranged and esthetically pleasing (because very many people will look at it every day).

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, they usually take some liberty, either to make the map fit into a certain format or to simplify the layout, or both. Some maps also have an artistic touch to them.

The goal of this paper is to produce such maps fully automatically, but adhering to 3 rather strictly: within a given tolerance, the lines on the map should be drawn according to their geographical course.

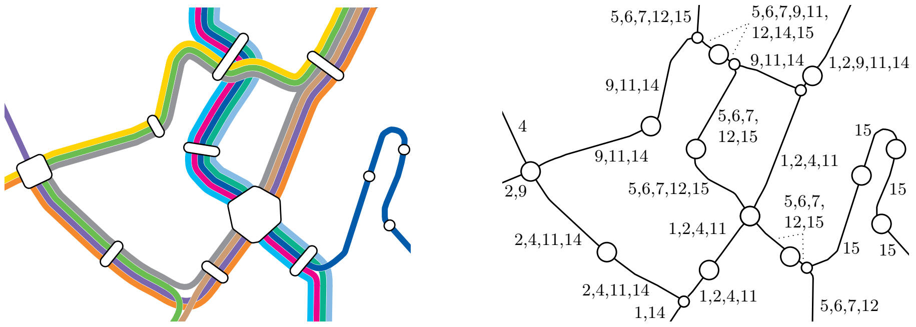

Fig. 1. Left: An excerpt from our transit map for the light rail network of the city of Stuttgart from 2015 (which happens to look very similar to the actual map of the transit agency). Right: The corresponding excerpt from the line graph which LOOM constructs from the given line data. Each edge corresponds to a segment of the line network, where the same set of lines take the same geographical course. Segment boundaries are often station nodes (large) but can also happen at intermediate nodes (small). The ordering of the lines for each edge (computed in LOOM's central optimization step) determines how the lines are drawn in the final map, and where crossing are necessary.

1.1 Overview and Definitions

LOOM proceeds in three phases, which we briefly describe in the following along with introducing some notation and terminology that will be used throughout the paper.

Input: The input to LOOM is a set S of stations and a set L of lines. Each station has a geographic location. Each line has a unique ID (in our examples, we use numbers), the sequence of stations it visits, and the geographical course between these stations. This data is usually provided as part of a GTFS feed of a transit network.[3]

Line graph construction: The *line graph* is an undirected labeled graph $G = (V, E, \ell)$, where $V \supseteq S$ (each station is a node, but here may be additional nodes), E is the set of edges, and each $e \in E$ is labeled with a subset $\ell(e) \subseteq L$ of the lines. Intuitively, edges correspond to segments of the network, where the same set of lines take the same geographical course (within a certain tolerance, see Section 2), and there is a node wherever a set of lines splits up in different directions. Figure 1 shows the line graph for an excerpt from the network of Stuttgart. We will see that the complexity of our algorithm depends on $M = \max_{e \in E} |\ell(e)|$, the maximal number of lines per segment / edge. The line graph construction is described in Section 2.

Line ordering optimization: This phase computes an *ordering* for the line graph, which is simply an ordering of $\ell(e)$ for each $e \in E$. This ordering de-

[3] If the information about the geographical course of the lines is missing, a good approximation is to compute shortest paths on the relevant network. However, this is a problem of separate interest and out of scope for this paper.

termines where crossing occur, and is hence critical for the appearance of the final map; see again Figure 1. We formulate this as an Integer Linear Program (ILP). This phase of LOOM solves a problem that is similar to but distinctly different from the metro-line crossing minimization (MLCM) [4] and its variants; this is discussed in Section 1.3. Section 3 provide the details about this step (in particular, about the ILP).

Rendering: Given a line graph and an ordering, the final step is to draw the transit map. 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 station. The node front for an edge e has $|\ell(e)|$ so-called *ports*. Drawing the maps then amounts to connecting the ports (according to the ordering computed in the previous step) and drawing the station polygons. Note that, just like in previous work, crossing must be explicitly visible and may not be "hidden" under stations. This is crucial such that individual lines can be easily traced through the network. Section 4 provides more details.

1.2 Contributions

We consider the following as our main contributions and ideas:

- A new algorithm, called LOOM (Line-ordering-optimized Maps) for the automatic generation of esthetically pleasing and geographically accurate transit maps. The input is basic schedule data as provided in a GFTS feed.
- This is, as far as we can tell, the first research paper on this problem. Previous research work considers a sub-problem of this task (ignoring the geographical course of the lines). For previous applied work no publications of the details of the underlying algorithm exist. See Section 1.3.
- Our approach resolves some issues with the MLCM model, 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). Also, unlike previous work, we do not only take crossings into account but also separation (that is, lines which are drawn right next to each other in some segment, but have other lines drawn between them in an adjacent segment). See again Section 1.3.
- We evaluate LOOM on the transit network of six cities around the world, including New York City. For each city, our algorithm takes below one minute in total. The resulting maps look surprisingly close to hand-made transit maps and superior to the automatic transit maps provided by Google Maps.[4]
- Our maps are publicly available under `loom.cs.uni-freiburg.de` . In particular, the maps there can interactively toggle the display of stations, edges, and node connections. This helps understanding the line graph of LOOM and how the final map is generated from it.

[4] For New York City, the transit maps from Google Maps looks conspicuously better than for other cities. We assume that this particular map has been constructed or at least improved manually.

1.3 Related Work

The problem of minimizing intra-edge crossings in transit maps was introduced in [5], 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 metro-line crossing minimization problem (MLCM) was coined in [4]. In that paper, optimal layouts for path and tree networks were investigated but arbitrary graphs were left as an open problem. In [1], [9], [2], several variants of MLCM were defined and efficient algorithms were presented for some of those, often with a restriction to planar graphs. In [3], an ILP formulation for MCLM under the periphery condition (see Sec. 3.4) was introduced. The resulting ILP was shown to have a size of $\mathcal{O}(|L|^2 \cdot |E|)$ with L being the set of lines and E the set of edges in the derived graph. In [8], it was observed that many (unavoidable) 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 MCLM problem [7].

All the aforementioned research is oblivious to the geographical course of the lines. It is also purely theoretical and lacks an experimental evaluation. There is also some applied work on transit maps, 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 [6]. Google Maps and Here Maps also feature transit maps generated (most likely, automatically) from real data. However, from an esthetic point of view, these maps are considerably less sophisticated than the maps of Dubreau and our maps.

2 Line Graph Construction

Public transit data is usually given as a set of vehicle trips. A trip is an ordered list of served stations, optionally given with a geographical path. In recent years, the General Transit Feed Specification (GTFS) has become the dominant format for exchanging schedules[5]. Because each trip is given explicitly, the graph induced by this data has many overlapping edges that may (partially) share the same path. The network topology is usually lost [TODO: what exactly is lost?].

Let e_1, e_2 be two edges in T with their geometrical paths τ_{e_1} and τ_{e_2}. We define a parametrization $\tau(\lambda) : [0, 1] \mapsto \mathbb{R}^2$ which maps the progress λ to a point p_d on τ. For some distance threshold $\hat{d}$, we say $((\lambda_1, \lambda_2), (\lambda'_1, \lambda'_2))$ is a shared segment of e_1 and e_2 if

$$\forall(\lambda, \lambda'), \lambda \in [\lambda_1, \lambda'_1], \lambda' \in [\lambda_2, \lambda'_2] : \|\tau_{e_1}(\lambda) - \tau_{e_2}(\lambda')\| \leq \hat{d}.$$

We restore the network topology and transform T into a well-formed transit graph T' by repeatedly combining shared segments between two edges e_1, e_2 into

[5] `https://developers.google.com/transit/gtfs/reference/`

a single new edge e_{12} until no more shared segments can be found. If for any of the edges the start of the shared segment λ lies not at the beginning of τ_e, we split $e = (u, v)$ at λ into two edges e' and e'' and introduce a new non-station node v' such that $e' = (u, v')$ and $e'' = (v', v)$ as well as $\tau_{e'}(1) = \tau_{e''}(0) = \tau_e(\lambda)$.

To find the shared segments between τ and τ', we sweep over τ in n steps of some $\Delta\lambda$, measuring the distance d between $\tau(i \cdot \Delta\lambda)$ and τ' at each $i < n$ along the way. If $d \leq \hat{d}$, we start a new segment. If $d > \hat{d}$ and a segment is open, we close it. The algorithm can be made more robust against outliers by allowing d to exceed $\hat{d}$ for a number of n steps. It can be sped up by indexing every linear segment of every path in a geometric index (for example, an R-Tree).

3 Line Ordering Optimization

The esthetic appeal of a transit map mostly depends on a clever ordering of the embedded lines. As described above, we formalize the search for the best ordering as the problem of minimizing the number of line crossings. Because there are already $|L(e)|!$ line permutations for a single edge, the number of ordering configurations even for a small network defies any brute force attack. For example, the Freiburg tram network (Table 1) has roughly $1.13 \cdot 10^{14}$ configurations. For the Stuttgart network, the number of configurations exceeds the number of atoms in the observable universe by a factor of 10^{27}. Finding the optimal configuration is the main challenge in both MLCM and MLTCM. In this section, we first build a baseline ILP from T to solve MLTCM with $\mathcal{O}(|E| \cdot M^2)$ variables and $\mathcal{O}(|E| \cdot M^6)$ constraints. We then define an improved ILP with only $\mathcal{O}(|E| \cdot M^2)$ constraints and describe an extension which solves MLTCM under the line separation constraint.

3.1 Baseline ILP

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

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

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

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

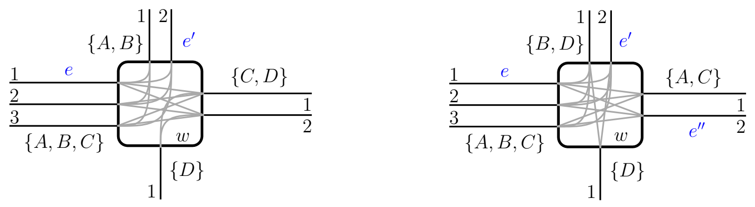

Fig. 2. Example instance.

Avoiding crossings Let A, B be two lines belonging to an edge $e = \{v, w\}$ and both extend over w. We distinguish between two cases: either A and B continue along the same adjacent edge e' (Fig. 2, left), or they continue along different edges e' and e'' (Fig. 2, 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:

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

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. 2, 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 the order of e' and e'' itself [TODO: itself refers to what?]. 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 $i < j$ because in that case a crossing would occur. We add $x_{ee'e''AB}$ to the objective function.

ILP size For mapping lines to positions at each edge we need $\leq |E| \cdot M^2$ variables and $\leq 2|E| \cdot M$ constraints. To minimize crossings, we have to consider $\leq M^2$ pairs of lines per edge, and introduce a decision variable for each such pair. That makes $\leq |E|M^2$ 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 $\leq \binom{M}{2}^2 < M^4$ constraints per line pair per edge, so $\leq |E|M^6$ in total. In summary, we have $\mathcal{O}(|E|M^2)$ variables and $\mathcal{O}(|E|M^6)$ constraints.

3.2 Improved ILP Formulation

The $\mathcal{O}(|E|M^2)$ variables in the baseline ILP seem to be reasonable, as indeed $\Omega(|E|M^2)$ crossings could occur. But the $\mathcal{O}(|E|M^6)$ 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.

Alternative line-position assignment Instead of a decision variable which encodes the exact position of a line, we know 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:

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

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:

$$\forall p \in \{1, ..., |L(e)|\}: \sum_{l\in L(e)} x_{el\leq p} = p. \tag{2}$$

So exactly for one line $x_{e.\leq 1} = 1$ is true, for two lines $x_{e.\leq 2} = 1$ (of which one has to fulfill $x_{e.\leq 1} = 1$) and so on.

Crossing Oracle We reconsider the example from 3.1, 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:

$$\sum_{p=1}^{|L(e)|} x_{eA\leq p} - \sum_{p} x_{eB\leq p} + x_{eB<A}M \geq 0 \tag{3}$$

$$x_{eB<A} + x_{eA<B} = 1. \tag{4}$$

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

$$abs(x_{eA<B} - x_{e'A<B}) - x_{ee'AB} \leq 0 \tag{5}$$

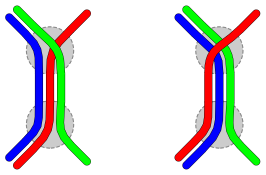
Fig. 3. Crossings are minimized in the left example (1), but the right example with 2 crossings better indicates line pairings.

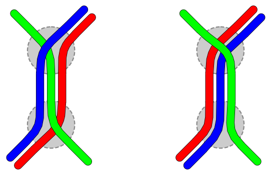
Fig. 4. Both orderings have the same number of crossings (2), but in the right example the crossing is done in one pass.

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 unequal and hence the absolute value of their difference is 1, enforcing $x_{e'AB} = 1$. As absolute value computation can not be part of an ILP we use the following equivalent standard replacement:

$$x_{eA<B} - x_{e'A<B} - x_{ee'AB} \leq 0 \tag{6}$$

$$-x_{eA<B} + x_{e'A<B} - x_{ee'AB} \leq 0. \tag{7}$$

Complexity of the improved ILP For the line-position assignment, we need at most $\leq |E|M^2$ 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 (see above). Hence the total number of variables and contraints in the improved ILP is $\mathcal{O}(|E|M^2)$.

3.3 Placement of Inevitable Crossings

In the ILPs described above, the placement of inevitable crossings largely depends on the solver's strategy and/or the input ordering.

As described above, we want to favor crossings in non-station nodes. We also found that maps appear more well-arranged if we prefer nodes with larger degree as crossing locations. Since these criteria are invariant to the orderings given to the $L(e)$, we can control them by adding a weighting factor $k_{ee'}$ for each $x_{ee'll'}$ in the objective function.

3.4 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 next to each other. Fig. 3 and Fig. 4 provide two examples. In both cases, the problem could be addressed by punishing the separation of lines. For two adjacent edges e and e' and a line pair (A, B) that continues from e to e', if A and B were placed next to each other in e (were partners in e) but not anymore in e', we want to add some penalty to the objective function. For this, we introduce a variable $x_{eA\|B} \in \{0, 1\}$. Let p_{eA} be the position of A in e and p_{eB} the position of B in e. We want $x_{eA\|B}$ to be 0 if $|p_e(A) - p_e(B)| = 1$ (if they occur next to each other) and 1 otherwise. As $x_{eA\|B} = x_{eB\|A}$, we define a set $U(e)$ of unique line pairs in e (that is, $(l, l') \in U(e) \Rightarrow (l', l) \notin U(e)$). To get the desired assignments, we add the following constraints per line pair (A, B) in $U(e)$:

$$\sum_{p=1}^{|L(e)|} x_{eA \le p} - \sum_{p} x_{eB \le p} - x_{eA\|B} M \le 1 \tag{8}$$

$$\sum_{p=1}^{|L(e)|} x_{eB \le p} - \sum_{p} x_{eA \le p} - x_{eA\|B} M \le 1. \tag{9}$$

If $|p_e(A) - p_e(B)| = 1$, then the difference is ≤ 1 and $x_{eA\|B}$ can be 0. If $|p_e(A) - p_e(B)| > 1$, then either (8) or (9) enforce $x_{eA\|B} = 1$. To prevent the trivial solution where $x_{A\|B}$ is always 1, we introduce the following additional constraint per edge e:

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

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

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 split between e and e' and 0 otherwise:

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

Interestingly, punishing line separations also addresses a special case of the periphery condition introduced in [3]. In general, this condition holds if lines ending in some node are always drawn at the left- or rightmost position in each incident edge. For nodes with degree ≤ 2, the periphery condition is always ensured by the line separation constraint (Fig. 5). For other nodes, however, the periphery condition may not be guaranteed anymore (Fig. 6).

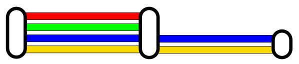
Fig. 5. Periphery condition guaranteed by separation constraint.

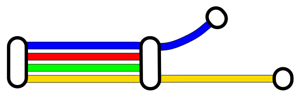
Fig. 6. Periphery condition not guaranteed by separation constraint.

Complexity We add one additional constraint per edge and a constant number of additional constraints and variables per line pair in each edge to the ILP. Therefore, the total number of variables and constraints remains $\mathcal{O}(|E|M^2)$.

3.5 Core problem graph

It is possible to further simplify the optimization problem. We observe that the minimum number of crossings or separations in T is exactly the number of crossings or separations that are inevitable and have to occur somewhere.

Lemma 1. *An inevitable crossing or separation between two lines J and K can only exist if there is a node v_{jk} with one of the following properties:*

1. *v_{jk} is adjacent to $e : J \notin L(e) \wedge K \in L(e')$ and $e' : J \in L(e') \wedge K \notin L(e)$.*
2. *$|\{e \in adj\,(v_{jk}) \mid J \in L\,(e) \wedge K \in L\,(e)\}| > 2$.*

Proof. If no such v_{jk} exists, it holds for all $v \in V$ that J and K either both extend over v on the same (two) edges or do not extend over v at all. In the latter case, no crossing (or separation) can occur and we can thus disregard it, which leaves us only with the former case. But if J and K extend over all nodes on the same two edges, then their relative ordering is irrelevant - we can just combine J and K into a new line I and give them an arbitrary global ordering.

We call nodes that comply to (1) a liaison node and nodes that comply to (2) a T-node for J and K. Additionally, we call a node in which one or more lines end a terminus node. Because of Lemma 1, we can be sure that for each inevitable crossing, there will be either a liaison node or a T-node where the crossing can occur in. Using this, we can greatly simplify the input graph:

1. collapse lines that always occur together into a single new line
2. remove each edge $e = (u, v)$ where u and v are termini for all $l \in L(e)$.
3. delete each node v with degree 2 and $L(e) = L(e')$ and combine $e = (u, v)$ and $e' = (v, w)$ into a single new edge $ee' = (u, w)$ with $L(ee') = L(e) = L(e')$.

Because we never delete a liaison nor a T-node, these operations preserve optimality. As we only delete nodes with degree 2, they will also not affect the optimality under the penalty heuristic described in Sect. 3.3, since nodes with higher degree are favored there for crossings/separations. This core problem graph may

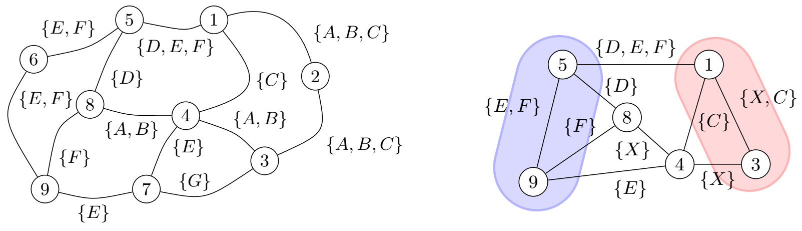

Fig. 7. Left: transit graph T with 7 lines, Right: core optimization graph of T with highlighted ordering-relevant connected components with more than 1 node.

be further broken down into ordering-relevant connected components using the rules below (Fig. 7). The subgraphs can then be optimized separately.

1. cut each edge $e = (u, v)$ with $|L(e)| = 1$ into two edges $e' = (u, v')$ and $e'' = (v'', v)$. Note that v' and v'' are not connected.
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'.

4 Rendering

We split the process of rendering a given transit graph into four basic steps, as illustrated in Fig. 8. Each $l \in L(e)$ can be rendered by perpendicular offsetting the edge's polyline by $-w\,|L(e)|/2 + w\,(p_e(A) - 1)$, where w is the desired line width and. We make room for the line connections between edges by expanding the node fronts (and thus the node polygon).

After that, the line connections in the node can be rendered independently by connecting all port pairs. In our experiments, we used cubic Beziér curves, but for schematic maps a circular arc or even a straight line might be preferable.

For station rendering, we found that the (buffered) convex hull of the station's node fronts yields reasonable results, 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.

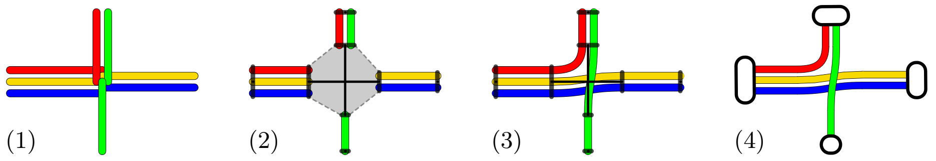

Fig. 8. The four steps of rendering a given transit graph: (1) render ordered lines as edges, (2) free node area, (3) render inner connections, (4) render station overlays.

5 Evaluation

We tested our approach [TODO: give it a name!] on the public transit schedules of four cities: Freiburg, Stuttgart, Chicago, New York. Table 1 provides the basic dimensions of each dataset and the time needed to extract the transit graph from the GTFS data.

For each dataset, we consider two versions of the transit graph: the baseline graph and the core optimization graph. For each graph, we considered three ILP variants: the baseline ILP (B), the improved ILP (I) and the improved ILP under the separation constraint (S). For each ILP, we evaluated two solvers: the GNU Linear Programming Kit (GLPK) and the COIN-OR CBC solver.

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`).

For each node v, we added the following penalties to the objective function: $3 \cdot \deg(v)$ for each inside-station crossing at v, $4 \cdot \deg(v)$ for each crossing between line pairs at v, and $4 \cdot \deg(v)$ for each line separation at v [TODO: reason for the factors 3, 4, 4?] Note that this counts each crossing or separation event twice: once for once for (A, B), and once for (B, A). Also note that separations are only considered in ILP variant (S) and disregarded in variants (B) and (I).

Table 1. Dimensions of transit graphs for our six datasets and extraction times from GTFS. $V_{\sigma=1}$ are station nodes, $V_{\sigma=0}$ non-station nodes. E are the graph edges, L the transit lines, and M the maximum number of lines per edge.

		Transit graph					Core optim. graph			
	t_{extr}	$\lvert V_{\sigma=1}\rvert$	$\lvert V_{\sigma=0}\rvert$	$\lvert E\rvert$	$\lvert L\rvert$	M	$\lvert V\rvert$	$\lvert E\rvert$	$\lvert L\rvert$	M
Freiburg (Trams)	1.54 s	74	6	81	5	4	20	21	5	4
Dallas (Light Rail)	9.16 s	108	9	118	7	4	24	24	7	4
Chicago (‘L’ Train)	21.9 s	143	10	154	8	6	23	24	8	6
Stuttgart (Trams)	8.79 s	192	31	235	15	8	51	60	15	8
Turin (Trams)	28.6 s	339	59	435	14	5	89	122	14	5
New York (Subway)[6]	23.3 s	456	61	548	26	9	110	138	23	9

Table 2 shows that for (I), the optimal order could be found in under 1 second for each dataset, and for (S) in under 1.5 minutes. For medium-sized networks an optimal solution for (S) was usually found in under 5 seconds. As expected, core graph reduction made the ILPs smaller. On average, the number of rows decreased by 64 % and the number of columns by 61 % for (I). For (S), the average decrease for the number of rows was 60 % and for the number of columns by 57 %. Although the ILPs for (S) were only slightly larger than for (I), optimization on the core graph took 35.4 times longer on average on COIN. CBC outperformed glpk for larger datasets, sometimes dramatically. (B) could not be optimized in under 12 hours for all datasets except Freiburg and Chicago on CBC. The time needed for graph extraction exceeded the optimization times for most datasets. This is no surprise, as we focused on the line-ordering optimization in this paper. Still, it indicates that further work is necessary here.

6 Conclusions

[TODO: mention thorough aesthetical evaluation via crowd-sourcing?]

[TODO: mention station rendering, which is sub par atm]

[6] with uncollapsed express lines

[†] depended on input ordering and solving strategy

Table 2. Sizes of ILPs generated with baseline approach (B), improved approach (I) and added line separation penalty (S), with or without reduction to core graph. Solving times on “wall-clock”. The numbers of crossings (×) and separations ($\Vert$) as well as the objective value are given after optimization.

		without core graph reduction				with core graph reduction						
		rows	cols	t_{glpk}	t_{CBC}	rows	cols	t_{glpk}	t_{CBC}	×	$\Vert$	obj.
Freiburg	B	4,034	347	198.6 s	34.2 s	380	124	375 ms	717 ms	5	1-2†	60
	I	641	453	484 ms	70 ms	208	160	**6 ms**	26 ms	5	1-2†	60
	S	771	514	2.1 s	2.1 s	237	173	**88 ms**	89 ms	7	0	96
Chicago	B	41,242	861	>12 h	>12 h	8,272	266	>12 h	82 m	22	4-7†	120
	I	1,691	1.151	19.6 s	786 ms	484	352	**163** ms	190 ms	22	4-7†	120
	S	2,151	1,372	(70 m)	22.3 s	595	405	39.2 s	**4.5 s**	27	0	160
Stuttgart	B	224,396	2,455	>12 h	>12 h	44,356	999	>12 h	>12 h	-	-	-
	I	5,089	3,381	((12.4 m))	4.5 s	1,923	1,355	10 s	**400 ms**	65	10-15†	276
	S	6,620	4,118	>12 h	2.3 m	2,473	1,618	??	**44.9 s**	69	3	426
New York	B	228,596	5,181	>12 h	>12 h	95,622	2,296	>12 h	>12 h	-	-	-
	I	10,575	7,081	((6.5 h))	3.1 s	4,480	3,126	19.3 m	**741 ms**	127	9†	308
	S	13,518	8,482	((??))	((3.7 h))	5,555	3,586	((??))	**76.5 s**	132	2	382

[TODO: mention that transit maps are not only useful for transit, but also to visualize for example flowcharts or storylines, layouts of electronic circuits]

References

1. Evmorfia Argyriou, Michael A Bekos, Michael Kaufmann, and Antonios Symvonis. Two polynomial time algorithms for the metro-line crossing minimization problem. In *International Symposium on Graph Drawing*, pages 336–347. Springer, 2008.
2. Evmorfia N Argyriou, Michael A Bekos, Michael Kaufmann, and Antonios Symvonis. On metro-line crossing minimization. *J. Graph Algorithms Appl.*, 14(1):75–96, 2010.
3. Matthew Asquith, Joachim Gudmundsson, and Damian Merrick. An ilp for the metro-line crossing problem. In *Proceedings of the fourteenth symposium on Computing: the Australasian theory-Volume 77*, pages 49–56. Australian Computer Society, Inc., 2008.
4. Michael A Bekos, Michael Kaufmann, Katerina Potika, and Antonios Symvonis. Line crossing minimization on metro maps. In *International Symposium on Graph Drawing*, pages 231–242. Springer, 2007.
5. Marc Benkert, Martin Nöllenburg, Takeaki Uno, and Alexander Wolff. Minimizing intra-edge crossings in wiring diagrams and public transportation maps. In *International Symposium on Graph Drawing*, pages 270–281. Springer, 2006.
6. Anton Dubreau. Transit Maps: Apple vs. Google vs. Us. `https://medium.com/transit-app/transit-maps-apple-vs-google-vs-us-cb3d7cd2c362`, 2016.
7. Martin Fink and Sergey Pupyrev. Metro-line crossing minimization: Hardness, approximations, and tractable cases. In *Graph Drawing*, volume 8242, pages 328–339, 2013.

8. Martin Fink and Sergey Pupyrev. Ordering metro lines by block crossings. In *International Symposium on Mathematical Foundations of Computer Science*, pages 397–408. Springer, 2013.
9. Martin Nöllenburg. An improved algorithm for the metro-line crossing minimization problem. In *International Symposium on Graph Drawing*, pages 381–392. Springer, 2009.