Drawing Metro Maps using Bézier Curves

Martin Fink[1], Herman Haverkort[2], Martin Nöllenburg[3], Maxwell Roberts[4],
Julian Schuhmann[1], and Alexander Wolff[1]

[1] Lehrstuhl für Informatik I, Universität Würzburg, Germany
[2] Faculteit Wiskunde en Informatica, TU Eindhoven, The Netherlands
[3] Institut für Theoretische Informatik, KIT, Germany
[4] Department of Psychology, University of Essex, Colchester, U.K.

Abstract. The automatic layout of metro maps has been investigated quite intensely over the last few years. Previous work has focused on the *octilinear* drawing style where edges are drawn horizontally, vertically, or diagonally at $45°$. Inspired by manually created curvy metro maps, we advocate the use of the *curvilinear* drawing style; we draw edges as Bézier curves. Since we forbid metro lines to bend (even in stations), the user of such a map can trace the metro lines easily. In order to create such drawings, we use the force-directed framework. Our method is the first that directly represents and operates on edges as curves.

1   Introduction

The problem of drawing metro maps automatically has been investigated by a number of publications over the last decade. It can be stated as follows. The input is a plane *graph* $G = (V, E)$, a map $\Pi \colon V \to \mathbb{R}^2$ that associates with each vertex its *geographic location*, and a *line cover* $\mathcal{L}$ of G, i.e., a set of paths in G with the property that every edge is contained in at least one path. The desired output is a drawing of G that fulfills or optimizes a set of aesthetic constraints. Previous algorithmic approaches [6,12,7] for drawing metro maps have all used a similar set of constraints comprising topology preservation, bend minimization, minimization of geographic distortion, edge length uniformity, non-overlapping station label placement, and *octilinearity*, that is, the requirement that edges must be drawn horizontally, vertically, or diagonally at $45°$.

Octilinearity vs. curvilinearity. A similar set of constraints, including octilinearity, seems to be used by graphic designers; see, for example, the book of Ovenden [8]. Such *schematic* maps potentially offer usability benefits by simplifying line trajectories, and hence reducing the amount of information that is irrelevant for deciding how to travel from one station to another. However, there is often a misbelief that it is merely the use of straight lines and a restricted angle set that benefits the user, and as a consequence many human designers fail to optimize octilinear maps, converting chaotic real-life line trajectories into complex sequences of short straight-line segments and bends [10].

In other instances, the network structure itself makes the benefits of octilinearity difficult to realize. A number of systems worldwide suffer from this, including the Paris metro. In some cases, using a different level of linearity (e.g., based on multiples of $30°$) that better matches the line trajectories permits more effective optimization, but
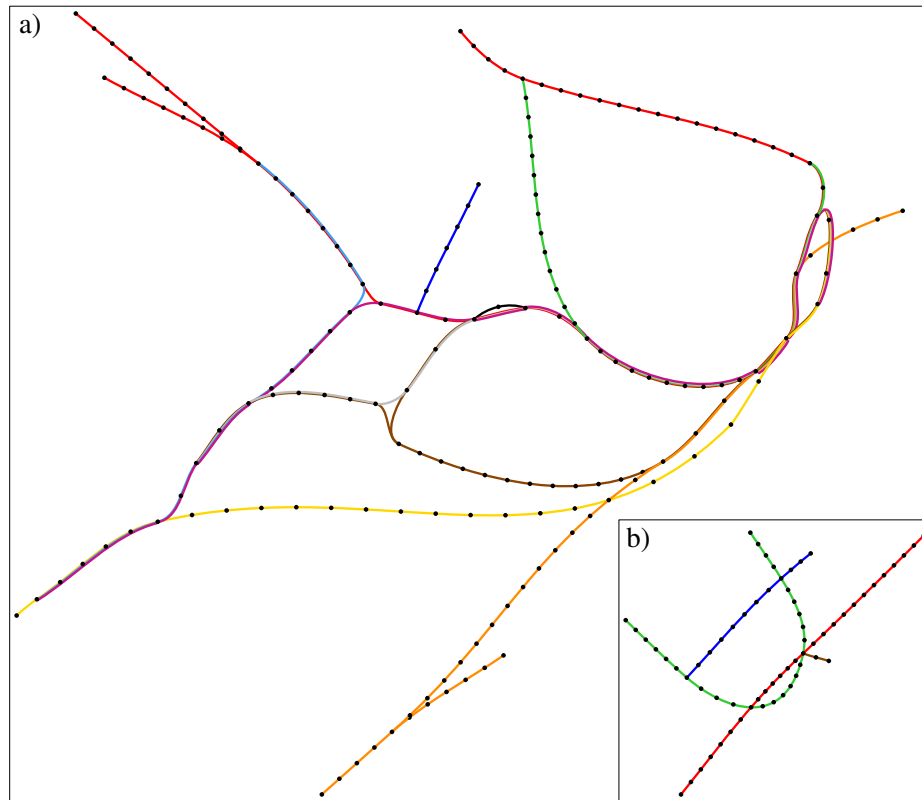
Fig. 1: Metro Networks of a) Sydney and b) Montréal drawn by our algorithm.

in the case of a dense interconnected network, where line trajectories are complex, a linear schematic may simply fail to offer sufficient simplification because of the network structure, irrespective of whether a human or computer attempts the design.

Under such circumstances, where the density of bends cannot be reduced, a curvilinear schematic may be attempted instead; see Fig. 1. Such a map seeks to simplify line trajectories, but using curves rather than straight lines. The underlying logic is that if a linear schematic yields sequences of many visually disruptive bends, then gentle curves with imperceptible radius change are preferable. This translates into using (fixed-degree) Bézier curves subject to the following criteria:

(B1) any pair of Bézier curves that are consecutive on a metro line must meet in a station and must have the same tangent there, and

(B2) the aim for each individual metro line is to consist of the smallest number of Bézier curves necessary in order to maintain interchanges,

(B3) points of inflection should be avoided.

In the specific case of the Paris metro, such a design is able to smooth and to emphasize the orbital lines (lines 2 and 6), simplifying the appearance of the network and making

salient its underlying structure. In a user study, a hand-drawn curvilinear design based on the above criteria out-performed the conventional octilinear Paris metro map, with up to 50% improvement in planning speed [11].

Previous work. Previous algorithmic work on drawing metro maps used (mostly) octilinear polylines rather than smooth curves to represent edges. Hong et al. [6] presented a force-based algorithm [2] for drawing metro maps, where several attracting and repelling forces act upon the vertices. The forces iteratively optimize the drawing until a locally optimal equilibrium is reached; this is generally very fast. Afterwards the authors use an interactive external labeling system to place station labels with few overlaps.

Another approach was suggested by Stott and Rodgers [12], who used multicriteria optimization based on hill climbing for drawing metro maps. Their approach performs local vertex moves as long as they improve the quality measure. They also integrated a label placement heuristic, so that one iteration of vertex movements alternates with one label placement iteration until no more local improvements are possible.

Nöllenburg and Wolff [7] used mixed-integer linear programming (MIP) to produce metro maps. Their approach always satisfies hard constraints like octilinearity and overlapping-free labeling, and optimizes soft constraints, e.g., the number of bends or geographic distortion. The runtime is high and determined by the time needed to solve the MIP with an external solver; an instance of their model may have no feasible solution at all. Yet, the layout quality in their case study is high and judged as the most similar to manually designed maps in an expert survey conducted with 41 participants who compared their layouts with those of Hong et al. [6] and Stott and Rodgers [12].

Wang and Chi [13] presented a system for octilinear on-demand focus-and-context metro maps that highlight routes returned by a route planning system while showing the rest of the network as less important context information. It can also be used to draw non-focused metro maps. They deform the given geographic map by minimizing a set of energy terms modeling the aesthetic constraints. Labeling is performed independently. Their method is both fast and creates good layouts, e.g., for mobile devices.

Our contribution. Our drawing algorithm is based on the well-known force-directed approach. This approach has been applied to drawing graphs with Bézier curves before; Brandes and Wagner [3] used it for visualizing train connection data and Finkel and Tamassia [4] for general-purpose graph drawing. In both cases, the authors turned all control points into vertices of the graph and used algorithms for straight-line drawings. Our algorithm, in contrast, uses additional, new forces that operate on the curves by moving vertices and control points in different ways. Our new forces aim at producing drawings that take the above requirements (B1) to (B3) into account.

We first describe our basic algorithm (see Sect. 3). By construction, it ensures requirement (B1). We improve the visual complexity of the output of the basic algorithm by merging, wherever possible, pairs of Bézier curves that are consecutive along a metro line; see Sect. 4 (and Fig. 8). This optimizes requirements (B2) and (B3). Force-directed algorithms depend a lot on their initial configuration; we run our algorithm on both octilinear drawings and geographic layouts (see Sect. 5). We have implemented our algorithm (in Java) and tested it on a number of real-world metro maps; see Sect. 6.

2 Preliminaries

In what follows, we give a short introduction into the two main ingredients that we make use of: First, we introduce Bézier curves and detail how our algorithm treats them. Second, we quickly recall the well-known force-directed approach.

Bézier curves. Bézier curves are a special type of parametric curves. We use so-called *cubic* Bézier curves. A cubic Bézier curve C is given by the cubic polynomial P_C: $[0,1] \to \mathbb{R}^2, t \mapsto (1-t)^3 p + 3(1-t)^2 t p' + 3(1-t)t^2 q' + t^3 q$, where p, p', q', and q are the *control points* of C [9]; see Fig. 2. We call p and q also the *endpoints* of C. We say that p' is the control point of C *at p* and q' is the control point of C *at q*.

We use the fact that the curve leaves p in the direction of p', i.e., the line pp' is the tangent of C at p. Now, if there is another curve D with a control point $\tilde{p}$ at p such that its tangent is the same but $\tilde{p}$ is on the opposite side of p w.r.t. p', then the concatenation of C and D is smooth in p. Our algorithm will ensure this behavior for consecutive edges of a metro line by construction. This makes it easier for the user of our metro maps to trace metro lines visually. Technically, we encode the position of p' by a unit-length vector $\overrightarrow{p_C}$ that gives the direction of the tangent and by the distance $r_C(p)$ between p and p'. Since we want to share

Fig. 2: Cubic Bézier curve.

a single tangent, as an object, between multiple curves, we allow $r_C(p)$ to be negative.

Our algorithm repeatedly needs to check whether two Bézier curves intersect or come too close to each other. Computing intersections of cubic curves is not easy. Since we just want to ensure that curves are not *too close*, it suffices to test *polygonizations* of the curves at hand. Given a cubic Bézier curve with polynomial P and an accuracy $\lambda \in \mathbb{Z}_{>0}$, we define the polygonization of C to be the polygonal chain connecting the points $P(0), P(1/\lambda), \ldots, P((\lambda - 1)/\lambda), P(1)$. The larger we make λ, the more precise but also the slower our closeness check gets. Since it is known [9] that a Bézier curve is always contained in the convex hull of its control points (see the gray shaded region in Fig. 2), we speed up our check by first testing the convex hulls for intersections.

Force-directed algorithms. Following the force-directed framework [5], our algorithm starts with some initial plane drawing, and then, iteratively, computes *forces* on the vertices (and control points). A force is a desired movement vector. At the end of each iteration, the computed forces are applied and the drawing is modified. Then the next iteration starts. Common forces are repulsive forces between vertices, and attractive forces between adjacent vertices. In general, forces are defined so that they tend to, gradually, improve the drawing. As all the forces together add up to the desired movement vectors, they have to be weighted so that they have the right relative strength. This is done by multiplying the force vectors with some weight factor; finding well-working factors is a matter of testing; see Sect. 6.

While we reuse standard forces known from the literature, we also define new forces that are specific to our drawing style. Whenever we have such a force that works on the shape of a curve, we will use the representation for control points introduced in the
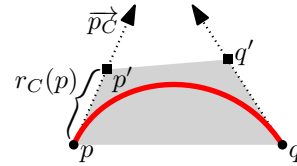

Algorithm 1: Basic structure of the force-directed algorithm using curves

---

> **Input**: plane graph $G = (V, E)$, $\varepsilon > 0$, integer $K > 0$
> obtain initial crossing-free drawing with Bézier curves
> **while** number of iterations $< K$ **and** maximum displacement $> \varepsilon$ **do**
> > compute forces on vertices
> > compute forces on curves
> > apply the forces to the current drawing
>
> **return** improved output drawing

---

previous section: If a force tries to move a control point, then this is represented as a force that tries to rotate the tangent used by this control point, and another force that tries to modify the (signed) distance between vertex and control point.

3 Basic Algorithm

Our algorithm follows the general idea of other force-directed algorithms as described in the previous section; its basic structure is shown in Alg. 1.

Additionally, we have to deal with the metro lines in the given set $\mathcal{L}$. From the point of view of a station, we (usually) want each pair of incident edges that belong to the same metro line to leave the station in opposite directions. Thus, we need extra information. First, we need the set $\mathcal{L}$ of metro lines with access from lines to the edges they use and vice versa. Second, for each vertex, we have a set of tangents given by unit-length vectors pointing away from the vertex. Third, for each edge e incident to a vertex v, we have a pointer to a tangent $\vec{t}$ of this vertex as well as the signed distance $r_e(v)$. Tangent and distance describe the position of the control point of e at v.

Our force-directed algorithm needs an initial drawing which must be crossing-free, with edges drawn as Bézier curves. If several edges incident to the same vertex v are to use the same tangent—but possibly in opposite directions—then this must be indicated in the input. In each iteration of our algorithm—right from the start—we assume that we have such a feasible drawing. In Sect. 5 we describe how to compute an initial Bézier drawing given either a straight-line or an octilinear drawing of the metro network.

3.1 Forces on Vertices

We use the standard forces defined by Fruchterman and Reingold [5]; they strive to move non-adjacent vertices far apart from each other and to make adjacent vertices have a common distance l. The second goal is useful for metro maps as the number of intermediate stops is normally a better indicator for the travel time than the geographic distance. We let any vertex u exert on any vertex v the repulsive force $F^{\text{rep-vtx}}(u, v) = (l/d(u,v))^2 \cdot \vec{uv}$. If v and u are adjacent, vertex u additionally exerts the attracting force $F^{\text{att}}(u, v) = d(u,v)/l \cdot \vec{vu}$ on v.

As a metro map represents a geographic metro network, stations should not be too far away from their actual location on the map. Therefore, we also have, for any vertex v, a force $F^{\text{orig}}(v) = \vec{vp_v}$ attracting v to its geographic position p_v.

3.2 Forces on Tangents and Control Points

Whereas previous force-directed graph-drawing algorithms did not directly operate on curves, we now present new forces for that very purpose—in order to take advantage of the power of Bézier curves.

Improving the shape of a curve. Consider an edge $e = uv$ that is represented as a curve with control point u' at u. If the distance $d(u, u')$ is small compared to the length of e, the curve could be very sharp, and almost have a bend. If, on the other hand, u' is far from u, the curve gets too long. As a compromise, we aim at having $d(u, u') = d(u, v)/3$, which worked well in our tests. To achieve this, we combine an attracting and a repulsive force on u' like the Fruchterman-Reingold forces. We do not want to change the tangent, just the (signed) distance $r_e(u)$ between u and u' in the direction of the tangent vector. Hence, the desired change is

$$F^{\text{shp}}(u, u') = \left(\frac{(d(u,v)/3)^2}{d(u,u')} - \frac{d(u,u')^2}{d(u,v)/3} \right) \cdot \text{sgn}(r_e(u)).$$

Note that this force is a scalar (and, hence, the same type of object as the distance $r_e(u)$).

Additionally, we aim at straightening curves, as a straight-line segment is the simplest type of Bézier curve. To this end, we move vertices as well as tangents.

First, we try to move vertex v so that it lies on the tangent t of $\{u, v\}$ at u. Let v_t be the point on t at distance l (the desired edge length) from u. Now the force

$$F^{\text{str-vtx}}(u, v) = \overrightarrow{vv_t}$$

moves v towards v_t.

Second, we aim at rotating tangent t at vertex u so that v lies on t as shown in Fig. 3. Let α be the signed angle between t and $\overrightarrow{uv}$. There may be multiple edges incident to u using t as their tangent. As a bad curvature of long edges is worse than a bad curvature of short edges, and the control point—and, thus, the curve—changes much more if the distance between vertex and control point is high, we do not simply sum up the individual forces on t, but use a sum that is weighted by the control point distances (as in the law of the lever), see Fig. 4. Let $v_1, \ldots, v_k$ be the vertices whose edges $uv_1, \ldots, uv_k$ use tangent t with control points $c_1, \ldots, c_k$ and imply a desired change of the tangent by angles $\alpha_1, \ldots, \alpha_k$. Then the rotational force is

$$F^{\text{str-tng}}(t, u, v_1, \ldots, v_k) = \frac{\sum_{i=1}^{k} \alpha_i \cdot d(u, c_i)}{\sum_{i=1}^{k} d(u, c_i)}.$$

Again, the force is a scalar, as a rotation is a one-dimensional movement.

Improving the angular resolution. We also aim at a good angular resolution at vertices, i.e., we want to have large angles between tangents. For any pair of different tangents t_1, t_2 at a vertex v we, therefore, add a repulsive force $F^{\text{rep-tng}}(t_1, t_2) = 1/\alpha(t_1, t_2)$ on t_1, where $\alpha(t_1, t_2) \in [-\pi, \pi]$ is the (signed) angle formed by t_1 and t_2. Note that, when measuring the angle, we have to take into account that some vectors are used in both directions while others are just used at one side of the vertex, see Fig. 5.
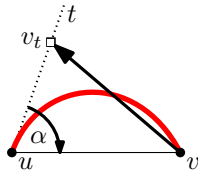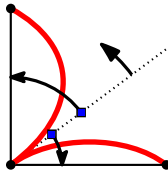
Fig. 3: Straightening a Bézier curve.

Fig. 4: Rotational forces applied to a tangent.
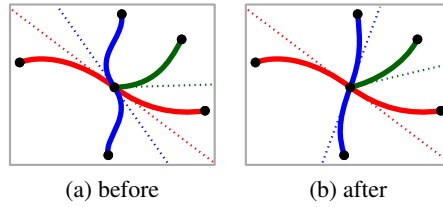

(a) before   (b) after

Fig. 5: Improving the angular resolution.

3.3 Avoiding Crossings by Limiting Forces

In order to avoid edge crossings, we would like to limit the forces that we apply as Bertault [1] does in his force-directed algorithm for straight-line drawings. In contrast to his approach, we cannot compute limitations of movements by using zones, i.e., octants, as intersections of Bézier curves are more difficult to compute and predict than crossings on straight-line segments. Instead, we check, for each pair of edges, whether the intended changes would result in a crossing. If this is the case, we change the movement vector of both endpoints so that the absolute value is half of the original value. We do this until the application of the new forces does not result in a crossing on any edge.

4 Decreasing the Visual Complexity

The main visual complexity of a drawing of a metro map with curves is created by a large number of inflection points (requirement (B3)), especially if adjacent curves of a metro line do not fit well together. Ideally, a metro line consists of just *one* Bézier curve, thus making it easy to trace the line visually. Often, this is not possible as intersections of a line with other lines restrict its shape. We can, however, reduce the number of curves significantly by merging consecutive curves on the same line. In our initial drawing, any edge of the graph representing the metro network is a single Bézier curve. In a step of our algorithm, we replace two consecutive curves by a single curve if this does not change the topology of the network. We can perform the replacement if the two edges are incident to a vertex of degree 2 or 4. We now sketch how we handle the two cases.

Merging curves on intermediate nodes. Suppose a degree-2 vertex v has two incident edges $e_1 = uv$ and $e_2 = vw$ lying on a common metro line ℓ. Then the two edges share a tangent at v. We merge the edges into a new edge $e = uw$. We use the control point of e_1 at u and the control point of e_2 at w and check whether the drawing of e intersects that of any other edge. If this is not the case, we remove e_1 and e_2 and insert e into the graph and its drawing, otherwise we keep e_1 and e_2 and discard e. Theoretically, the chance of avoiding intersections could be increased by testing different values for the control point distances of the merged edge. Our tests, however, suggest that this is not necessary, since, in the final drawings, almost no vertices of degree 2 remained.

To keep track of vertices that are lost by merging edges, we have to maintain a sorted list of such vertices for each edge. As the lists of both e_1 and e_2 may already contain

virtual vertices, we concatenate the two lists with v in between to get the list for e. We do not only use this list to produce the final drawing, placing the virtual vertices evenly distributed along the drawing of e, but we also use the number of intermediate vertices to adjust the desired length of the edge, especially when computing the attraction between u and w. If l is the desired length for simple edges and e contains k virtual vertices, then the desired length of e is $(k+1)l$.

If $\deg(v) = 1$, i.e., v is the *terminal* of some line, and the edge incident to v contains some virtual vertices, then v typically represents a terminal located in a sparsely occupied suburb. We can give more freedom to the drawing of such end edges by decreasing the influence of the force $F^{\mathrm{orig}}(v)$ that attracts v to its original position, e.g., by scaling the force by some value $c < 1$ (in our tests, we used $c = 1/50$). This allows v to be placed closer to the center, which makes the drawing more compact.

Merging curves on simple interchange nodes. Merging pairs of curves that meet at vertices of higher degree is difficult since it is not clear how to ensure that three or more merged curves actually meet in (or close to) a single point. We restrict ourselves to degree-4 vertices in which two lines intersect.

Suppose that a vertex v is adjacent to vertices u, u', w, w' via edges e_1, e_1', e_2, and e_2'. Line ℓ contains edges $e_1 = uv$ and $e_2 = vw$, and line ℓ' contains $e_1' = u'v$ and $e_2' = vw'$. We want to replace the concatenation of e_1 and e_2 by $e = uw$ and that of e_1' and e_2' by $e' = u'w'$. If we manage to do so, we represent v as a virtual vertex, i.e., as the intersection of e and e'. At the same time, we have to make sure that the only new crossing that is introduced is the one representing v. We try to find appropriate curves for e and e' by adjusting the distances of the control points to the respectice endpoints while keeping the tangents (as we did for vertices of degree 2). For distance $r_e(u)$, we test values in the interval $[r_{e_1}(u), d(u, w)]$ at equal distances. It makes sense to require $r_e(u) \geq r_{e_1}(u)$ since the combined curve is longer than e_1 and the new control point should not be too far from u. By testing all different combinations of discretized distances for the four involved control points, we often found feasible solutions.

Note that there is an additional constraint: the crossing that now represents v should divide both new edges e and e' roughly in proportion to the numbers of virtual vertices on e and e', respectively, on the two sides of v. If e contains k virtual vertices left of v and k' virtual vertices right of v, then the intersection with e' should have a distance to u that is $(k+1)/(k+k'+2)$ times the total length of the curve of e. We allow a deviation from this optimal position by a factor of δ (we used $\delta = 20\%$ in our tests) times the length of the part of the curve to the left of v and to the right of v, respectively. We call the allowed range the δ-*zone* of e.

In all further steps of the algorithm, we have to adhere to these zones for crossing edges. Further mergings of lines including e are only allowed if v stays in the allowed δ-zone Furthermore, we do not allow movements that would violate these constraints. So we also consider these zones in the force limitation phase at the end of each iteration.

5   Creating a Feasible Input Drawing

As input, our algorithm expects the embedded graph representing the metro network, the coordinates of stations, and information regarding the metro lines, see Sec. 3. Some

of this information can be guessed automatically. Suppose, e.g., that exactly two of the edges incident to v are used by a line ℓ. Then, we assume that ℓ traverses the station and, hence, the two edges must have the same tangent, leaving v in opposite directions. Otherwise, we assume that the input contains an *annotation* saying, e.g., that the two edges leave v in the same direction.

We need an initial feasible *Bézier* drawing before the first iteration starts, see Sect. 3. This drawing must guarantee that (i) tangents obey their annotations and (ii) the topology (i.e., embedding) is the same as in the plane input graph. We discuss two ways to obtain such an initial drawing depending on the input graph; either from an octilinear drawing or from the straight-line drawing induced by the coordinates of the stations.

Initial drawing from octilinear layout. Suppose that we are given an octilinear layout which may either be computed, e.g., using the mixed integer program of Nöllenburg and Wolff [7], or be a manually generated plan such as an official metro map. If there are bends in edges, we first transform these bends into dummy vertices that do not correspond to stations. Later, the algorithm may delete such dummy vertices by merging the two incident curves. Given the dummy vertices, we now have a straight-line drawing.

To get a drawing using only Bézier curves, we place each control point at its incident endpoint (or, conceptually, at a very small distance) and let each tangent point towards the other endpoint of the edge. Now, we still have the same straight-line drawing, but the edges are technically Bézier curves. In this process, we must respect the annotations of the tangents, so that the right curves have common tangents at a vertex. Unfortunately, there can be situations in which this is not possible, see Fig. 6. In practice, however, such situations are quite unlikely; they never occurred in our tests.
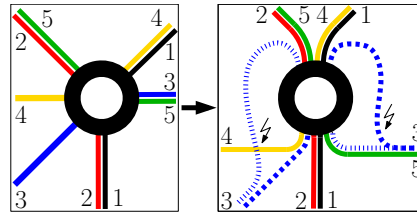

Fig. 6: Example where it is impossible to keep the embedding and ensure that both edges of each line share the same tangent.

At any vertex where the tangents are not yet correct, we now choose new tangents. We do this one after the other, starting at tangents that are shared by edges. We choose the first tangent so that it is closest to the tangents it replaces. We insert any new tangent so that the clockwise order of the adjacent vertices is correct (if possible). Finally, by moving the control points very close to the vertex, we get a drawing that is arbitrarily close to the straight-line drawing and that does not have any crossing.

Initial drawing from geographic layout. If we do not have an octilinear drawing, the initial drawing can also be constructed given just the coordinates of the stations. Similarly to Nöllenburg and Wolff [7], we start with the straight-line drawing induced by the station positions. This drawing may have crossings; we replace them by dummy vertices and get a crossing-free straight-line drawing. This drawing can then be transformed into a crossing-free Bézier drawing as presented in the previous paragraph.

Since the introduced crossings, as dummy vertices, are preserved over all iterations, they will also be present in the output drawing. Fortunately, their number is small

in practice. (From a network point-of-view it indeed makes sense to have stations at crossings.) E.g., the large London network (which was built by competing companies operating single lines) has only four crossings—the same as in the official tube map.

Note that, in the initial drawing, there are only two different tangents at a dummy vertex, each for one of the two crossing lines; this is also the case in the final drawing. Additionally, in the more advanced version of the algorithm, we can even transform the dummy vertex to a (dummy) virtual vertex before the algorithm starts; see Sect. 4.

6   Implementation and Tests

We implemented our algorithm in Java. For testing we used the metro networks of four cities: London (297 vertices, 217 of which have degree 2, 13 metro lines), Vienna (90/71/5), Montréal (69/59/4), and Sydney (173/144/10); see Fig. 1 for the latter two. The input data contained the graph structure as well as information on the lines and geographic positions of stations. We also used octilinear layouts of these cities as initial drawings, which we generated using the MIP approach of Nöllenburg and Wolff [7]. In both cases, tangents were annotated where necessary; see Sect. 5.

Effects of virtual vertices. We were especially interested in how far making vertices virtual influenced the visual complexity of metro maps. Figure 8 shows the power of virtual vertices for the metro map of Vienna. Starting with an octilinear layout (Fig. 7), drawing (a) was computed without virtual vertices and, hence, no curves were merged. Clearly, the drawing does not have any sharp bends. The attraction to the geographic position of vertices, however, caused some unnecessary inflection points. Next, we allowed for virtual vertices of degree 2 (Fig. 8(b)). For Vienna, this worked on all intermediate vertices, reducing the number of Bézier curves significantly. Finally, we also enabled virtual vertices of degree 4 (Fig. 8(c)). This worked for 8 of 9 possible vertices. Three lines were represented by just one curve, while the two other lines need three curves.

Fig. 7: Initial octilinear drawing.

It turned out that including virtual vertices of degree 2 always worked well, and that they were fast to handle. There were almost no remaining vertices of this type even after the very first iteration; hence, testing the remaining degree-2 vertices was fast in all following iterations. In contrast, trying to merge edges at degree-4 vertices was much slower because potentially many combinations of control point positions had to be tested. Additionally, we observed that once many virtual vertices of degree 4 had been added, the drawing did not change much any more. Apparently, the additional constraints on the crossings make the drawing more rigid, and many movements get forbidden. Therefore, we decided to first have many, i.e., hundreds, of iterations without caring about virtual vertices of degree 4, and then treat them in a single (more time-consuming) final iteration.
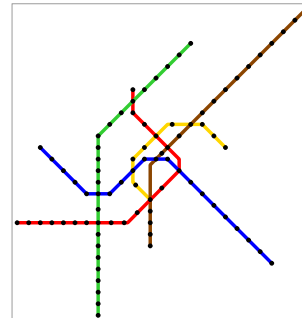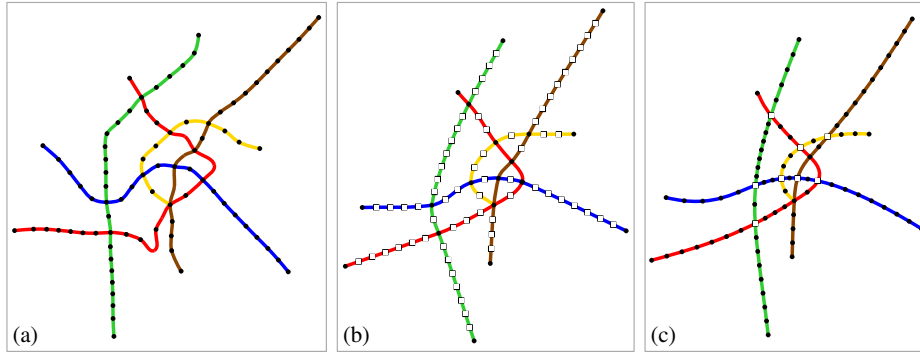
Fig. 8: Metro network of Vienna (a) without virtual vertices, with (b) virtual vertices of degree 2 (highlighted by squares), and (c) additionally with virtual vertices of degree 4 (highlighted).

Running time. On the largest instance, the Underground of London, the running time for creating a drawing starting with an octilinear layout was 224 seconds on a 3 Ghz dual-core computer with 4 GB RAM. This includes the 156 seconds spent on the last iteration, in which curves were merged by inserting virtual vertices of degree 4. In contrast, the first 500 iterations just took 68 seconds.

Weights of forces. As noted in Sect. 2, weight factors are needed that let different forces work well together. We group the forces depending on the object on which they operate. In our tests, the following factors turned out to work well: $F_{\text{vert}}^{\text{res}} = (F^{\text{rep-vtx}} + F^{\text{att}} + 10F^{\text{orig}} + 3F^{\text{str-vtx}})/100$ for vertices, $F_{\text{tan}}^{\text{res}} = 150F^{\text{rep-tng}} + 0.03F^{\text{str-tng}}$ for tangents, and $F_{\text{cpdist}}^{\text{res}} = F^{\text{shp}}/20$ for control point distances.

Initial drawing and F^{orig}. We tested the algorithm both with a straight-line drawing and an octilinear layout as input. When we defined F^{orig} using the geographic station locations, the version using the octilinear layout performed slightly better. The best results, however, we achieved when using the octilinear layout as input and defining F^{orig} w.r.t. the vertex positions in the octilinear input drawing. In this case, the center had more space, and more curves could be merged, which reduced the visual complexity. Figures 1 and 8 were computed this way.

7 Conclusion and Open Problems

The implementation of our algorithm performed well on small and medium-size networks (Vienna, Sydney, and Montreal in our tests). In such cases, many curves could be merged so that, in the end, lines consisted only of few curves. We conclude that spending the extra time for merging as many curves as possible is worth it. In denser regions, such as the center of London, many curves were merged, but there are also a number of vertices that did not allow this, making the drawing more complex.

To further improve our drawings it would help to devise ways to merge curves that intersect in vertices of degree other than 2 or 4. Consider a vertex v of degree 3 that is

traversed by a line ℓ_1 and that is the terminal of another line ℓ_2 with a different tangent. We can then merge the two edges of ℓ_1 and represent v by its position on ℓ_1. The edge of ℓ_2 still has to maintain its own tangent and control point at v. This is also possible if there are more lines whose tangent is not linked to ℓ_1. Similarly, if two lines ℓ_1 and ℓ_2 traverse a vertex v, we can merge their edges incident to v so that their crossing represents v—if none of the remaining edges shares its tangent with ℓ_1 or ℓ_2.

In the future, however, we intend to approximate each metro line globally as one C^2-continuous cubic spline right from the start rather than piece-by-piece for every edge. We want to apply curve-fitting techniques from computer graphics; the challenge will be to additionally define and implement appropriate constraints that allow for a sufficient and maybe context-dependent amount of distortion to smooth unimportant bends and yet ensure, e.g., the desired angular properties in vertices of degree at least 3.

We also intend to incorporate the placement of station labels into our algorithm.

References

1. Bertault, F.: A force-directed algorithm that preserves edge crossing properties. In: Kratochvíl, J. (ed.) GD 1999. LNCS, vol. 1731, pp. 351–358. Springer, Heidelberg (2000)
2. Brandes, U.: Drawing on physical analogies. In: Kaufmann, M., Wagner, D. (eds.) Drawing Graphs: Methods and Models, LNCS, vol. 2025, chap. 4, pp. 71–86. Springer, Heidelberg (2001)
3. Brandes, U., Wagner, D.: Using graph layout to visualize train connection data. J. Graph Algorithms Appl. 4(3), 135–155 (2000)
4. Finkel, B., Tamassia, R.: Curvilinear graph drawing using the force-directed method. In: Pach, J. (ed.) GD 2004. LNCS, vol. 3383, pp. 448–453. Springer, Heidelberg (2005)
5. Fruchterman, T.M.J., Reingold, E.M.: Graph drawing by force-directed placement. Softw. Pract. Exper. 21(11), 1129–1164 (1991)
6. Hong, S.H., Merrick, D., do Nascimento, H.A.D.: Automatic visualisation of metro maps. J. Visual Lang. Comput. 17(3), 203–224 (2006)
7. Nöllenburg, M., Wolff, A.: Drawing and labeling high-quality metro maps by mixed-integer programming. IEEE Trans. Visual. Comput. Graphics 17(5), 626–641 (2011)
8. Ovenden, M.: Metro maps of the world. Harrow Weald: Capital Transport Publishing, 2nd edition (2003)
9. Prautzsch, H., Boehm, W., Paluszny, M.: Bézier and B-Spline Techniques. Springer, Heidelberg (2002)
10. Roberts, M.J.: Underground maps unravelled: Explorations in information design. Published by the author, Wivenhoe (2012), http://privatewww.essex.ac.uk/~mjr.
11. Roberts, M.J., Newton, E.J., Lagattolla, F.D., Hughes, S., Hasler, M.C: Objective versus subjective measures of Paris metro map usability: Investigating traditional octolinear versus all-curves schematics. Int. J. Human-Comput. Studies 71, 363–386 (2013)
12. Stott, J., Rodgers, P., Martínez-Ovando, J.C., Walker, S.G.: Automatic metro map layout using multicriteria optimization. IEEE Trans. Visual. Comput. Graphics 17(1), 101–114 (2011)
13. Wang, Y.S., Chi, M.T.: Focus+context metro maps. IEEE Trans. Visual. Comput. Graphics 17(12), 2528–2535 (2011)