Edge Routing with Ordered Bundles[☆]

Sergey Bereg[a], Alexander E. Holroyd[b], Lev Nachmanson[b], Sergey Pupyrev[c,d]

[a]*Department of Computer Science, University of Texas at Dallas, USA*
[b]*Microsoft Research, USA*
[c]*Department of Computer Science, University of Arizona, USA*
[d]*Institute of Mathematics and Computer Science, Ural Federal University, Russia*

Abstract

Edge bundling reduces the visual clutter in a drawing of a graph by uniting the edges into bundles. We propose a method of edge bundling that draws each edge of a bundle separately as in metro-maps and call our method ordered bundles. To produce aesthetically looking edge routes, it minimizes a cost function on the edges. The cost function depends on the ink, required to draw the edges, the edge lengths, widths and separations. The cost also penalizes for too many edges passing through narrow channels by using the constrained Delaunay triangulation. The method avoids unnecessary edge-node and edge-edge crossings. To draw edges with the minimal number of crossings and separately within the same bundle, we develop an efficient algorithm solving a variant of the metro-line crossing minimization problem. In general, the method creates clear and smooth edge routes giving an overview of the global graph structure, while still drawing each edge separately and thus enabling local analysis.

Keywords: graph drawing, edge routing, edge bundling, crossing minimization

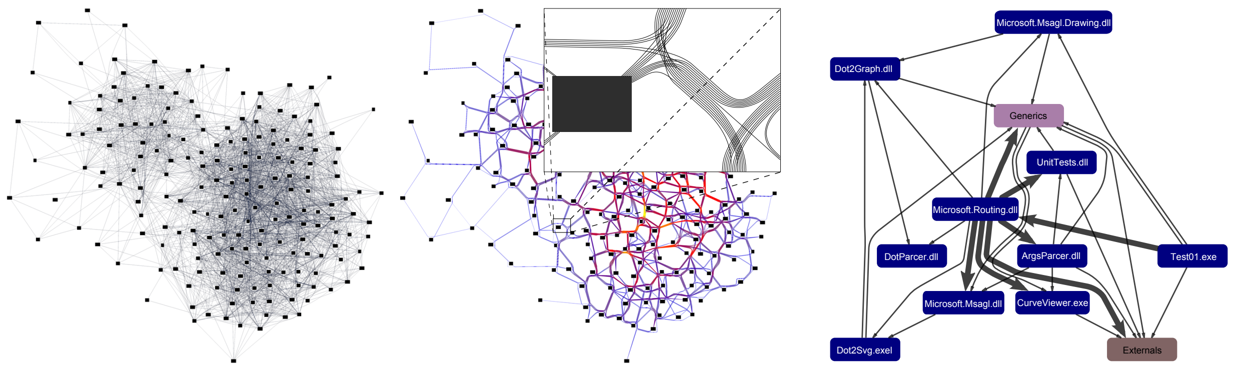
1. Introduction

Graph drawing is often used in information visualization for exploring and analyzing network data. The core components of most graph drawing algorithms are the computation of positions of the nodes and edge routing. In this paper we concentrate on the latter problem.

For many real-world graphs with substantial numbers of edges traditional algorithms produce visually cluttered edge routes. The relations between the nodes are difficult to analyze by looking at such drawings. Recently, edge bundling techniques have been developed in which some edge segments running close to each other are collapsed into bundles to reduce the clutter [23, 28, 29]. While these methods create an overview drawing, they typically allow the edges within a bundle to cross and overlap each other arbitrarily, making individual edges hard to follow. In addition, previous approaches allow the edges to overlap the nodes and obscure their text or graphics [11, 19, 23, 29, 30].

[☆]A part of the results of this paper was presented at the *19th International Symposium on Graph Drawing* [38].

Email addresses: `besp@utdallas.edu` (Sergey Bereg), `holroyd@microsoft.com` (Alexander E. Holroyd), `levnach@microsoft.com` (Lev Nachmanson), `spupyrev@gmail.com` (Sergey Pupyrev)



(a) (b)

Figure 1: (a) Visualizations of `jazz` graph with straight-line edges (left) and ordered edge bundles (right). (c) An example of ordered edge bundles where edges walk around node labels.

We present a novel edge routing algorithm for undirected graphs that produces drawings in a "metro map" style; see Figure 1. The graphs for which our algorithm is best applicable are of medium size with a large number of edges, although it can process larger graphs efficiently too.

The input for our algorithm is an undirected graph with given node positions. These positions can be generated by a graph layout algorithm, or in some applications (for instance, geographical ones) they are fixed in advance. During the algorithm the node positions are not changed. The main steps of our algorithm are similar to existing approaches, but with several innovations, which we indicate with *italic text* in the following description.

Path routing. The edges are routed along the edges of some underlying graph and the overlapping parts are organized into bundles. One approach here has been to minimize the total "ink" of the bundles. However, this often produces excessively long paths. *For this reason we introduce a novel cost function for edge routing.* Minimizing this function forces the paths to share routes, creating bundles, but at the same time it keeps the paths relatively short. Additionally, the cost function penalizes routing too many edges through narrow gaps between the nodes. *We furthermore show how to route the bundles around the nodes.*

Path construction. In this step the paths belonging to the same bundle are "nudged" away from each other. The effect of this action is that individual edges become visible and the bundles acquire thickness. *Contrary to previous approaches, we try to draw the edges of a bundle as parallel as possible with a given gap.* However, such a routing may not always exist because of the limited space between the nodes. We provide a heuristic that finds a drawing with bundles of a suitable thickness.

Path ordering. To route individual edges, an order of the edge segments within each bundle needs to be computed. This order minimizes the number of crossings between the edges of the same bundle. The problem of finding such an order is a variant of the metro-line crossing minimization problem. *We provide a new efficient algorithm that solves this problem exactly.*

The next section summarizes related work. In Section 3 we introduce the ordered edge bundling algorithm, followed by the detailed explanation of its steps in Sections 4-6. Results of experiments are presented in Section 7. We discuss some additional issues and future work in Section 8.

2. Related work

Edge routing. The problem of drawing edges as curves has been considered in several visualization papers. Such edges, with their smoother and organic shape, have an aesthetic appeal to humans, who are known to prefer round shapes [3]. Curves can improve performance in some typical graph reading tasks when compared to straight-line edges [20, 40, 42]. In some scenarios, graph nodes have associated text labels, and curved edges are used to avoid node interiors; see Figure 1(b). Dobkin et al. [14] initiated a visibility-based approach in which the shortest obstacle-avoiding route for an edge is chosen. The method requires the computation of the visibility graph, which can be too slow for graphs with hundreds of nodes. An improvement was suggested by Dwyer and Nachmanson [17], achieving faster edge routing using sparse visibility graphs. The latter method constructs approximate shortest paths for the edges. We use the method as the starting point of our algorithm.

Dwyer et al. [16] proposed a force-directed method, which improves a given edge routing by straightening and shortening edges that do not overlap node labels. However, a feasible initial routing is needed. Moreover, the method moves the nodes, which is prohibited in our scenario.

The problem of finding obstacle-avoiding paths in the plane arises in several other contexts. In robotics, the motion-planning problem is to find a collision-free path among polygonal obstacles. This problem is usually solved with a visibility graph, a Voronoi diagram, or a combination thereof [41]. Though these methods can be used to route a single edge in a graph, they do not apply directly to edge bundling. Many path routing problems were studied in the very-large-scale integration (VLSI) community. We use some ideas of [10, 12] as will be explained later.

Edge bundling. The idea of improving the quality of layouts via edge bundling is related to edge concentration [35], in which a layered graph is covered by bicliques in order to reduce the number of edges. Eppstein et al. [13, 18] proposed a representation of a graph called a confluent drawing. In a confluent drawing a non-planar graph is visualized in a planar way by merging together groups of edges, in the manner of railway tracks.

We believe that the first discussion of bundled edges in the graph drawing literature appeared in [24]. There the authors improve circular layouts by routing edges either on the outer or on the inner face of a circle. In that paper the edges are bundled by an algorithm that tries to minimize the total ink of the drawing. Here we follow a similar strategy, but in addition we try to keep the edges themselves short (among other criteria).

In the hierarchical approach of [28] the edges are bundled together based on an additional tree structure. Unfortunately, not every graph comes with a suitable underlying tree, so the method does not extend directly to general layouts. In [39] edge bundles were computed for layered graphs. In contrast, our method applies to general graph layouts.

Edge bundling methods for general graphs are given in [7, 11, 19, 23, 29, 30, 31]. In the force-directed approach [29] the edges can attract each other to organize themselves into bundles. The method is not efficient, and while it produces visually appealing drawings, they are often ambiguous in a sense that it is hard to follow an individual edge. The approaches [11, 31] have a common feature: they create a grid graph for edge routing. Our method also uses a special graph for edge routing, but our approach is different; we modify this graph to obtain better edge bundles. Unlike the methods [11, 19, 23, 29, 30], we avoid edge-node overlaps.

Recently there were several attempts to enhance edge bundled graph visualizations via image-based rendering techniques. Colors and opacity were used to highlight the density of overlapping lines [11, 28]. Shading, luminance, and saturation encode edge types and edge similarity [19]. We focus on the geometry of the edge bundles only. The existing rendering techniques may be applied on top of our routing scheme.

Path ordering. The problem of ordering paths along the edges of an embedded graph is known as the metro-line crossing minimization (MLCM) problem. The problem was introduced by Benkert et al. [5], and was studied in several variants in [1, 2, 4, 21, 22, 36]. We mention two variants of MLCM that are related to our path ordering problem.

Asquith et al. [2] defined the so-called MLCM-FixedSE problem (also called MLCM-PA in [36]), in which it is specified whether a path terminates at the top or bottom of its terminal node. For a graph $G = (V, E)$ and a set of paths P they give the algorithm with $\mathcal{O}(|E|^{5/2}|P|^3)$ time complexity. A closely related problem called MLCM-T1, in which all paths connect degree-1 terminal nodes in G, was considered by Argyriou et al. [1]. Their algorithm computes an optimal order of paths and has a running time of $\mathcal{O}((|E| + |P|^2)|E|)$. Recently, Nöllenburg [36] presented an $\mathcal{O}(|V||P|^2)$-time algorithm for these variants of the MLCM problem. Our algorithm can be used to solve both the MLCM-FixedSE and MLCM-T1 problems with complexity $\mathcal{O}(|V||P| + |V|\log|V| + |E|)$. To the best of our knowledge, this is the fastest solution to date of these variants of MLCM.

Crossing minimization problems have also been researched in the circuit design community [8, 26, 32]. Given a global routing of nets (physical wires) among the circuit modules, the goal is to adjust the routing so as to minimize crossings between the pairs of the nets. In this context, a problem similar to the path ordering problem was studied by Groeneveld [26], who suggested an $\mathcal{O}(|V|^2|P|)$-time algorithm. Another method, which works for graphs of maximum degree 4, is given in [8]. Marek-Sadowska et al. [32] considered the problem of distributing the set of crossings among circuit regions. They introduced constraints for the number of allowed crossings in each region, and provided an algorithm to build an order with these constraints. Their algorithm has $\mathcal{O}(|V|^2|P| + |V|\xi^{3/2})$ complexity, where ξ is the number of path crossings.

3. Algorithm

In this section we introduce some terminology and give an overview of our algorithm. The input to the edge bundling algorithm comprises an undirected embedded graph $G = (V, E)$, where V is the set of nodes and E is the set of edges. Each node of the graph comes with its **boundary curve**, which is a simple closed curve. The boundary curves of different nodes are disjoint, and the interiors they surround are disjoint too. The output of the algorithm is a set of paths $P = \{\pi_{st} : (s, t) \in E\}$, where for an edge $(s, t) \in E$, its path is referenced as π_{st}; it is a simple curve in the plane connecting the boundary curve of s with the boundary curve of t. The input also includes a required path width for each edge and a desired path separation.

Step 1 of the edge bundling algorithm produces an initial routing of the paths. This routing is done on an auxiliary graph $\widetilde{G}$ called the **routing graph**. The edges of $\widetilde{G}$ are straight-line segments in the plane.

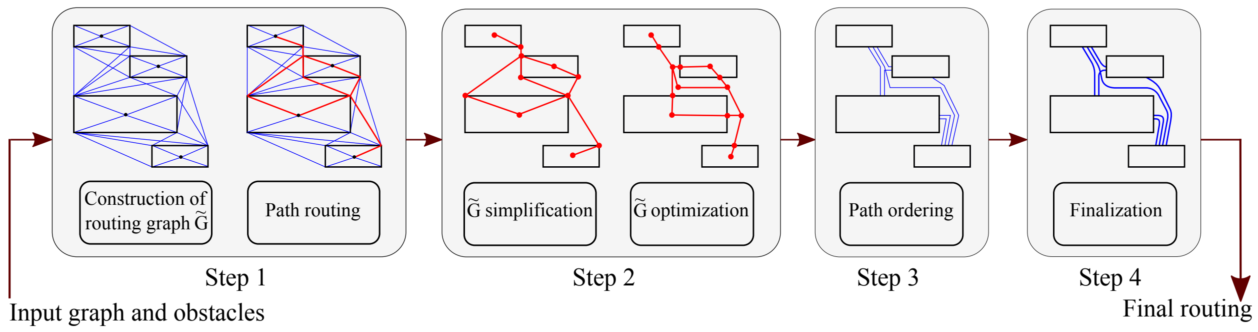

Construction of routing graph $\widetilde{G}$
Path routing
Step 1
$\widetilde{G}$ simplification
$\widetilde{G}$ optimization
Step 2
Path ordering
Step 3
Finalization
Step 4
Input graph and obstacles
Final routing

Figure 2: Algorithm pipeline.

Step 2 improves the routing. Firstly, $\widetilde{G}$ is cleaned up by removal of all its nodes and edges that are not part of any path. Secondly, $\widetilde{G}$ is modified to accommodate the paths together with the path widths and separations, and to improve the paths.

In steps 1 and 2 we use a notion of **ink** similar to [24, 39]. Given a graph embedded in the plane with straight-line edges, the ink I of a set of paths in the graph is the sum of the lengths of the edges of the graph used in these paths; if an edge is used by several paths its length is counted only once. A set of paths sharing an edge of the graph is called a **bundle**. It is important to note that in the final drawing the paths in a bundle will be drawn as separate curves, so I does not represent the amount of ink used in the final drawing.

Step 3. Consider a bundle with the shared edge (u, v) in $\widetilde{G}$. We would like to draw the segments of the paths that run along edge (u, v) as distinct curves parallel to (u, v). This requires ordering of the paths of each bundle. Step 3 produces such an order for all bundles in a way that minimizes crossings (which occur at the nodes).

Step 4 draws each path as a smooth curve using straight-line and biarc segments.

The overview of the algorithm is illustrated in Figure 2. Sections 4, 5, and 6 describe in detail Steps 1,2, and 3, respectively. The final step (the construction of smooth curves) is performed as the last operation of the process, although it is described in Section 5.1. This is necessary as the structure of the final paths is important for understanding Step 2 of the algorithm.

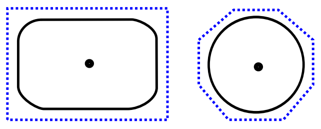
4. Path Routing

In this section, we explain the first step of our algorithm – generation of the routing graph and initial path construction.

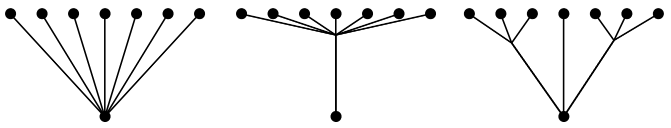
4.1. Generation of the Routing Graph

As a preprocessing step, for each node $s \in V$, we create a convex polygon O_s containing the boundary curve of s in its interior. We keep the number of corners in O_s bounded from above by some constant. We call O_s an **obstacle**, and build the obstacles in such a way that O_s and O_t are disjoint for different s and t. For each $s \in V$, we choose a point inside of the boundary curve of s, which of course is inside of O_s too, and call this point a **center** of O_s; see Figure 3(a).

The algorithm starts with the construction of the routing graph $\widetilde{G}$. Traditionally edge routing is performed on the visibility graph [14] containing edges between every pair of visible obstacles



(a)



(b)

Figure 3: (a) Boundary curves (solid black) of two nodes with its obstacles (dotted blue) and centers. (b) (left) The routing without bundling; (center) edge routing with small ink; (right) edge routing with larger ink but shorter paths.

corners; the routing is performed using shortest paths on the graph between corresponding obstacles. Such graph may have quadratic number of edges and require quadratic time for construction, which can be too slow, especially for interactive applications. We achieve faster routing using a sparse visibility spanner, that is, a subgraph of the visibility graph approximately preserving shortest paths. In our implementation, we utilize the so called Yao graphs [43], as they contain linear number of edges and can be efficiently constructed [9, 17]. The vertices of $\widetilde{G}$ comprise the obstacle centers, the obstacle corners, and some additional vertices from the boundaries of O_s. These additional points are added to make $\widetilde{G}$ rich enough for path generation. The points added to O_s have the property that each cone with the apex at the center of O_s and with an angle given in advance contains at least one corner of O_s or at least one new added point; see Figure 4(a). To build the visibility edges of $\widetilde{G}$ adjacent to node u, the node is surrounded by a family of cones covering the plane with the apexes at u, and in each cone at most one edge is created by connecting u with another node of $\widetilde{G}$ which is visible from u, belongs to the cone, and is closest to u in the cone. The angle of the cones is $\alpha = 30°$ and the cone axes are represented by vectors $(\cos(i\alpha), \sin(i\alpha))$ for $0 \le i \le 11$ in our default settings. The method works in $\mathcal{O}(r \log r)$ time and creates $\widetilde{G}$ with $\mathcal{O}(r)$ edges, where r is the number of vertices in $\widetilde{G}$ [9, 17].

There are two types of edges in $\widetilde{G}$: the visibility edges and the edges from the obstacle centers. A visibility edge has both its ends belonging to the obstacle boundaries and it does not intersect the interior of any obstacle. An edge of the second type is adjacent to an obstacle center. Such an edge does not intersect the interior of any obstacle except of the one containing the center; see Figure 4(a). We denote the set of nodes of $\widetilde{G}$ by $\widetilde{V}$ and the set of edges of $\widetilde{G}$ by $\widetilde{E}$. By construction $r = \mathcal{O}(|V|)$, and therefore $\widetilde{G}$ has $\mathcal{O}(|V|)$ nodes and $\mathcal{O}(|V|)$ edges.

Further on, it is necessary for our method to keep the visibility edges of $\widetilde{G}$ disjoint from the obstacles. For this purpose after computing $\widetilde{G}$, we shrink the obstacles slightly so that each obstacle still contains its boundary curve, but the visibility edges do not touch or intersect the obstacles anymore; see Figure 4(b).

Next we route the paths on $\widetilde{G}$.

4.2. Routing Criteria and Routing Cost

In order to create bundles, we follow the idea of ink minimization [23, 24, 39]. However, minimizing the ink alone does not always produce satisfactory results; some paths in a routing with a small ink may have sharp angles or long detours as shown in Figure 3(b). We improve the paths by introducing additional criteria.

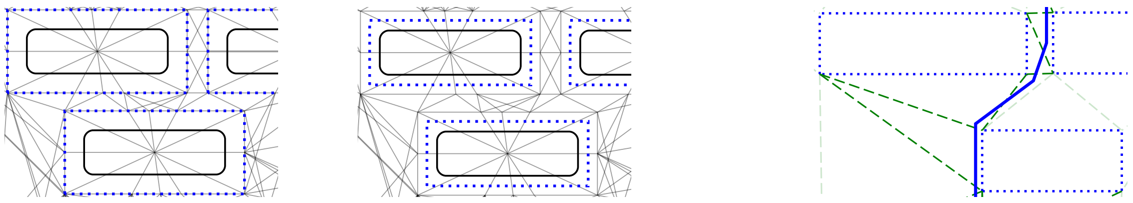

(a) (b) (c)

Figure 4: (a-b) A fragment of routing graph $\widetilde{G}$: (a) the boundary curves are solid black, the obstacles are dotted blue, the edges of $\widetilde{G}$ are solid gray; (b) the obstacles are shrunk, so the visibility edges of $\widetilde{G}$ do not cross or touch the obstacles. (c) The solid curve represents a path, the long dashed opaque line segments are the capacity segments crossed by the path, dotted line segments are the shrunk obstacles.

- *Short Path Lengths*. The length of a path π_{st} is denoted by ℓ_{st}. One approach would be to minimize the sum of ℓ_{st}. However, this gives insufficient weight to paths that connect relatively close pairs of nodes. For this reason we instead consider the normalized lengths $\ell_{st}/|st|$, where $|st|$ is the distance between the centers of O_s and O_t. For a set of paths P, we define $L = \sum_{\pi_{st} \in P} \ell_{st}/|st|$.

- *Capacity*. We would like to avoid routing too many paths through a narrow gap. For this purpose we adapt an idea of [10, 12], where capacities are associated with segments connecting obstacles. To construct these segments, we compute the constrained Delaunay triangulation [15] with the constrained edges being the obstacle sides. A Delaunay edge connecting two different obstacles is called a **capacity segment**. We penalize for too many paths passing through a capacity segment.
  Let us define **capacity** c_σ of a capacity segment σ. Suppose σ connects points $a \in A$ and $b \in B$ for obstacles A and B. Then $c_\sigma = (|a, B| + |b, A|)/2$, where $|a, B|$ ($|b, A|$) is the distance from a to B (from b to A). While routing a path, we assign it to the capacity segments intersected by the path; see Figure 4(c). For each capacity segment σ, we select all paths assigned to σ and calculate **routing width** w_σ as the total path widths plus the total path separation. For example, if a segment σ is intersected by paths a, b, and c, then w_σ is equal to the path separation taken twice plus the sum of widths of a, b, and c. **Capacity overflow penalty** of σ is computed as $p_\sigma = \max(w_\sigma - c_\sigma, 0)$. The total capacity overflow penalty is defined as $C = \sum_\sigma p_\sigma$, where the sum is taken over all capacity segments.

Minimizing the ink, the paths lengths, and the capacity overflow penalty often is impossible at the same time. For example, smaller ink usually leads to longer paths. The capacity penalty can force some overlapping paths to go through different channels thus possibly increasing the ink. Trying to satisfy all the criteria, we consider a multi-objective optimization problem: minimize the **routing cost** of a set of paths P defined as

$$\textit{routing cost} = k_{ink} I + k_{len} L + k_{cap} C.$$

We stress here that *routing cost* is used only in steps 1 and 2 of our algorithm. Steps 3 and 4 producing the final routing preserve the overflow penalty and try to preserve the path lengths. The notion of ink, as we define it, is not applicable to the final routing since the resulting paths are drawn as separate curves. The choice of values of k_{ink}, k_{len}, and k_{cap} are discussed in Section 7.2.

4.3. Routing of Paths

We try to route the paths on $\widetilde{G}$ with minimal *routing cost*. The problem is formulated as follows.

Problem 1 (Path Routing). *Given graph $\widetilde{G} = (\widetilde{V}, \widetilde{E})$ and a set of pairs of nodes $\{s_i, t_i\} \in \widetilde{V}^2$, find paths between s_i and t_i for all i so that* routing cost *is minimized.*

Here s_i and t_i correspond to the centers of obstacles connected by the edges of G. Problem 1 is NP-hard because its variant with $k_{ink} = 1$ and $k_{len} = k_{cap} = 0$ is the Steiner Forest problem [25]. The problem remains NP-hard in the case when $k_{ink} = k_{len} = 0$ and $k_{cap} = 1$. It is easy to see that the latter variant is a generalization of the Edge Disjoint Paths problem [33], as one can assign unit capacities for every edge of $\widetilde{G}$ (so that only one path passes through an edge without overflow penalty). Therefore, we suggest a heuristic to optimize *routing cost*. In this heuristic we solve an easier task, where some paths are already known and we need to route the next path. We will route it by minimizing an *additional cost*, which is the increment of *routing cost* associated with this path. For a path π_{st}, we define *additional cost* $= k_{ink}\, \Delta I + k_{len}\, \ell_{st}/|st| + k_{cap}\, \Delta C$. Here, ΔI is the increment in the ink, which is the sum of edge lengths of π_{st} that were not part of any previous path. Similarly, ΔC is the growth of $\sum_\sigma p_\sigma$, where the sum is taken over all capacity segments σ assigned to path π_{st}; note that the non-zero terms in the sum correspond only to the segments whose capacity is strictly less than the corresponding routing width.

Problem 2 (Single Path Routing). *Given graph $\widetilde{G} = (\widetilde{V}, \widetilde{E})$ and a set of already routed paths on $\widetilde{G}$, find a path from $s \in \widetilde{V}$ to $t \in \widetilde{V}$ so that* additional cost *is minimized.*

To solve the problem, for each edge e of $\widetilde{G}$, we assign weight $k_{ink}\, \delta_e + k_{len}\, \ell_e/|st| + k_{cap}\, \Delta C_e$ to e. Here δ_e is ℓ_e if e is not taken by a previous path, and 0 otherwise. The value of ΔC_e is the growth of $\sum_\sigma p_\sigma$ for the case the path is passing through e, where the sum is taken over all capacity segments σ crossed by e. It can be seen that the minimum of *additional cost* is achieved by a shortest path from s to t when we use these weights. We apply the Dijkstra's algorithm to find the path solving Problem 2.

To find a solution for Problem 1 (not necessarily the optimal one), we organize paths in a sequence $\{s_1, t_1\}, \ldots, \{s_m, t_m\}$, and iteratively solve Problem 2 for $\{s_k, t_k\}$, with the paths $\{s_i, t_i\}$, $i < k$ already routed, for $k = 1, \ldots, m$. In our implementation, the paths in the sequence are taken in the increasing order of the distances between the corresponding obstacle centers. Such an order helps to connect close pairs of nodes with relatively straight curves. The routing of a single path takes $\mathcal{O}(|\widetilde{E}| \log |\widetilde{V}|)$ time with the Dijkstra's algorithm. Hence, routing of all paths amounts to $\mathcal{O}(|E||\widetilde{E}| \log |\widetilde{V}|)$ steps.

Can we do better in this iterative approach than routing one path at a time? It turns out that we can route optimally a set of paths with a common end, in the case when capacity component of the routing cost is not taken into account (or equivalently, when the graph nodes are positioned far enough from each other). The result is interesting from theoretical point of view, as it shows that the problem of routing edges with the minimum ink can be solved more efficiently (though the variant is still NP-hard as noticed earlier). In the remaining of the section, we assume that all the paths share a common end $s^* \in \widetilde{V}$, having degree d and neighbors $t_1, \dots, t_d$, and that $k_{cap} = 0$. We define an additional cost of a set of paths by analogy with the additional cost of a path.

Problem 3 (Multiple Path Routing). *Given graph $\widetilde{G}$ with some paths already routed, find paths for $\{s^*, t_1\}, \dots, \{s^*, t_d\}$ so that* additional cost *is minimized.*

We solve this problem by a dynamic programming approach. We first fix a set of pre-existing paths in $\widetilde{G}$; *additional cost* will always be with respect to these paths. Let us call a **state** a pair (v, P), where v is a node of $\widetilde{G}$ and P is a subset of $\{t_1, \dots, t_d\}$. We need to solve our problem for the state $(s^*, \{t_1, \dots, t_d\})$. We reduce the problem to solving it for "smaller" states, that are the states with fewer elements in P. For a state (v, P), we define its cost $f(v, P)$ as minimal *additional cost* of a set of paths $\{\pi_{vt} : t \in P\}$. A set of paths giving the minimal value of $f(v, P)$ is called an **optimal** set for state (v, P). Let us clarify the structure of an optimal set of paths.

By the subgraph **generated** by a set of paths in $\widetilde{G}$, we mean the subgraph of $\widetilde{G}$ comprising all edges and nodes in the paths.

Lemma 1. *For each state, there exists an optimal set of paths that generates a tree.*

Proof: Let Π be any optimal set of paths for state (v, P) and G^Π be the graph generated by Π; note that G^Π is connected. Let T be a shortest path tree of G^Π, rooted at v, with respect to ordinary edge lengths. Let Π' be the set of paths connecting v to the points of P in T. Note that *additional cost* of Π' is at most that of Π. Indeed, the increment in I is no greater because T is a subgraph of G^Π. Each path of Π' is shortest in G^Π and thus no longer than the corresponding path of Π. Hence, Π' is an optimal set for (v, P). $\square$

Lemma 1 leads us to the following formula.

$$f(v,P) = \min \begin{cases} f(u,P) + k_{ink}\,\delta_{vu} + k_{len}\,\sum_{t\in P} \ell_{vu}/|s^*t|, & \text{for all } u \in \widetilde{V} \text{ adjacent to } v, \\ f(v,P') + f(v, P - P'), & \text{for all } P' \text{ with } \emptyset \subset P' \subset P \end{cases}$$

The minimum is taken over both expressions on the right as u and P' vary. To verify this, we consider some optimal set of paths for (v, P) that form a tree, and split into two cases. The first line corresponds to the case where u is the only neighbor of v in the tree. The second line is the case where v has at least two neighbors, thus the paths can be partitioned into two proper subsets with no common edges.

Now we describe how to compute $f(v, P)$. Let us assume that f is known for all states (u, P') for P' a proper subset of P. To compute $f(v, P)$, a new graph H is constructed with $\widetilde{G}$ as a subgraph. An edge e of $\widetilde{G}$ has weight $k_{ink}\,\delta_e + k_{len}\,\sum_{t\in P} \ell_e/|s^*t|$ in H. We add a new node h to H and connect it with all nodes of $\widetilde{G}$. For every new edge (h, u), we assign weight

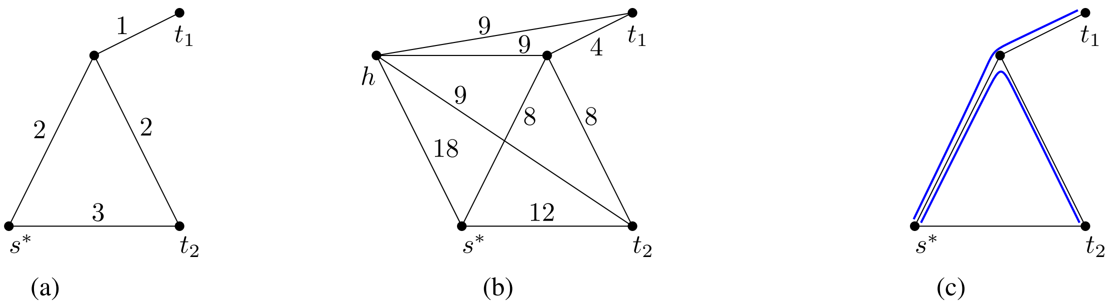


Figure 5: An example for Problem 3 with $k_{ink} = 2, k_{len} = 1, k_{cap} = 0$, and $|s^*t_1| = |s^*t_2| = 1$. The goal is to find paths $\pi_{s^*t_1}$ and $\pi_{s^*t_2}$ with minimal *additional cost*. (a) Graph $\widetilde{G}$, weights ℓ_e for each edge e are shown. (b) Constructed graph H with edge weights. (c) Optimal paths (blue) with *additional cost* = 17.

$\min_{P'} f(u, P') + f(u, P - P')$, where P' varies over proper non-empty subsets of P; see Figure 5. One can see that the required value $f(v, P)$ is the length of a shortest path from v to h in graph H. We can compute it using the Dijkstra's algorithm.

To solve Problem 3 we work bottom-up. We first compute all $f(v, P)$ with $|P| = 1$ and v is a node of $\widetilde{G}$, by the algorithm for Problem 2, where we find a path with minimal *additional cost*. Then we compute the values $f(v, P)$ for each v and $|P| = 2, \ldots, d$ by creating the corresponding graphs H. Finally, the answer for the problem is $f(s^*, \{t_1, \ldots, t_d\})$.

Let us discuss the running time. The main steps of the above algorithm are the construction of graph H, and finding a shortest path on it with the Dijkstra's algorithm for each state (v, P). Luckily, graph H depends only on the P component of a state. The construction of graph H for a fixed set P takes $\mathcal{O}(2^{|P|}|\widetilde{V}|)$ time. We execute the Dijkstra's algorithm only once per P starting from h to compute $f(v, P)$ for all $v \in \widetilde{V}$. Thus, finding $f(v, P)$ for a known H and for all $v \in \widetilde{V}$ takes $\mathcal{O}(|\widetilde{V}| \log |\widetilde{V}|)$ time. Summing over all possible sets P gives

$$\mathcal{O}\Big(\sum_{P}\big[2^{|P|}|\widetilde{V}|+|\widetilde{V}|\log|\widetilde{V}|\big]\Big) = \mathcal{O}\Big(\sum_{i=0}^{d}\binom{d}{i}\big[2^{i}|\widetilde{V}|+|\widetilde{V}|\log|\widetilde{V}|\big]\Big) = \mathcal{O}\big(3^{d}|\widetilde{V}|+2^{d}|\widetilde{V}|\log|\widetilde{V}|\big).$$

5. Path Optimization

In this section, we describe an optimization of $\widetilde{G}$, which starts with removing all its nodes and edges that are not part of any path. Then we modify $\widetilde{G}$ and move its nodes to accommodate space needed for drawing paths. First we provide details on how we draw the paths and define some additional constructions on $\widetilde{G}$.

5.1. Structure of the Paths

We call a node $v \in \widetilde{V}$ an **intermediate** node if it is not an obstacle center. Recall that a bundle is a set of paths sharing the same edge of $\widetilde{E}$. That means that as soon as the paths are routed on $\widetilde{G}$, the bundles become defined. In the final drawing we would like to draw the paths of a bundle respecting the width and path separation and outside of the obstacles. As described in Section 4.1,

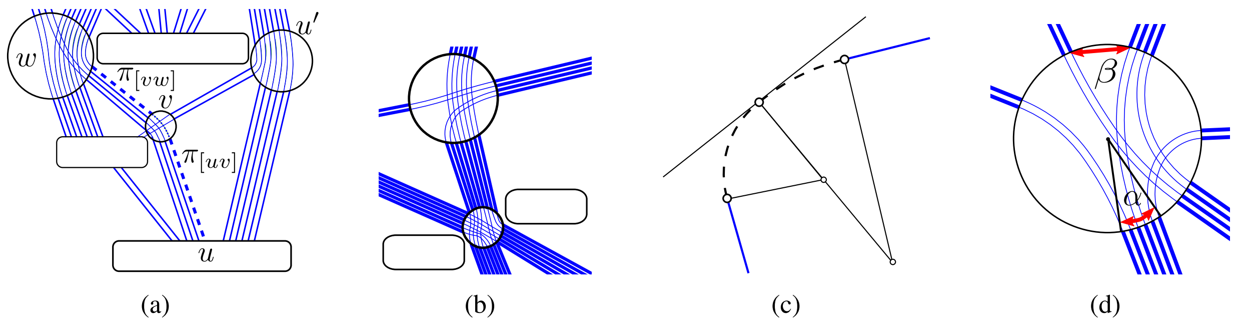


Figure 6: (a) Structure of the paths in the final drawing. The circles represent the hubs of the intermediate nodes and the rounded rectangles represent the hubs of the obstacle centers. The dashed path represents segments $\pi_{[uv]}$, $\pi_{[vw]}$, and the hub segment. (b) A hub with not enough free space. (c) A biarc consisting of two circular arcs with an identical tangent at the connecting point. (d) The hub radius should be large enough so that $\alpha \leq 90°$ and $\beta \geq$ *path separation*.

we shrink the obstacles after constructing $\widetilde{G}$, therefore each intermediate node of $\widetilde{V}$ lies outside the obstacles; an obstacle intersects an edge of $\widetilde{E}$ only if the edge is adjacent to the obstacle center. To give a general idea of the final path structure, we refer to Figure 6(a).

We surround each node of $\widetilde{G}$ by a simple closed curve. In the case of an obstacle center it is the boundary curve of the node, and in the case of an intermediate node it is a circle disjoint from the obstacles. We call these curves **hubs** and denote the hub of node v by h_v. Each path π from the bundle of edge (u, v) is represented by a straight-line segment connecting h_u and h_v, which is called a **bundle segment** and denoted by $\pi_{[uv]}$. The bundle segments of different paths are drawn in a particular order, as discussed in Section 6 below. If path π passes through consecutive nodes u, v, w, then we connect the segments $\pi_{[uv]}$ and $\pi_{[vw]}$ by a smooth curve called a **hub segment**. The hub segment is constructed in a way that it is tangent to $\pi_{[uv]}$ and $\pi_{[vw]}$ and is contained in h_v. For the the hub segment, we use a biarc, that is, a smooth curve formed by two circular arcs with a common point; see Figure 6(c). We follow the method of [37] to construct a biarc. Therefore, in the final drawing each path is represented by a sequence of straight-line bundle segments and biarc hub segments.

Next we give a heuristic that moves intermediate nodes of $\widetilde{G}$ to find space for hubs and bundles. Ideally, we would like to keep the hubs large enough to accommodate all the bundle segments with their widths and separations. However, this is not always succeeded; see Figure 6(b).

5.2. *Optimization Goals*

We change positions of intermediate nodes of $\widetilde{G}$ to provide enough space for drawing paths. To this end, we compute how much free space we need around bundles and hubs. The ideal bundle width for edge e, denoted by e_w, is the sum of all path widths of the bundle, together with the total of the separations between them. For example, for a bundle with three paths of widths 1, 2 and 2, and a path separation of 1, the bundle width is 7.

On one hand, the radius of a hub for an intermediate node u should be large enough to accommodate all paths. We define a **desired** radius of a hub, which is the maximum over two values. The first one is $\max_e \mu e_w$, where the maximum is taken over all edges $e \in \widetilde{E}$ adjacent to u and μ is a positive constant, which is $\frac{1}{\sqrt{2}}$ in our implementation. This guarantees that the angle subtended by the arc of the hub corresponding to the bundle is at most 90 degrees; see Figure 6(d). In addition, the desired radius is chosen so that any two bundles are separated before entering the hub by at least the given edge separation; see Figure 6(d).

On the other hand, the radius of a hub should be small enough that: (a) the hub interior does not intersect other hubs and (b) the hub is disjoint from the obstacles. We say that the hub of an intermediate node u is **valid** if conditions (a) and (b) are satisfied, and in addition, for each edge (u, v), the straight-line segment from the center of h_u to the center of h_v does not intersect any obstacle (if v is an intermediate node) and intersects only obstacle O_v (otherwise). The routing graph is **valid** if all intermediate hubs are valid.

Our optimization procedure finds a valid graph $\widetilde{G}$ while pursuing two objectives. First, the radii of all hubs should be as close to the desired radii as possible. Second, we try to keep *routing cost* of $\widetilde{G}$ low.

5.3. Routing Graph Optimization

Let us denote by p_v the position of node v of $\widetilde{G}$. Before the optimization step we have $\widetilde{G}$ in a valid state. We find hub radii (perhaps very small) keeping $\widetilde{G}$ valid, but we may not have enough space for the hubs and the bundles. Equivalently, we may have hubs for which the desired radius is larger than the actual one. The optimization starts by trying to move each such hub away from obstacles into another valid position, where its radius is closer to the desired radius than before. This is done in a loop, for which we initially set r to the desired radius of node v. A move is attempted along the direction $\sum_s u_s$, where $u_s = \frac{\overrightarrow{O_s p_v}}{|\overrightarrow{O_s p_v}|}$ and the sum is taken over all obstacles s intersecting the circle of radius r with p_v as the center. The length of the move is θr for some fixed $\theta > 1$, which is 1.2 in our implementation. If the new position of v is invalid, we reduce r (dividing it by θ, in our implementation) and repeat the procedure. We stop processing a node after 10 unsuccessful moves.

After this step we obtain more space for the paths, but *routing cost* is typically increased. Therefore we again iteratively adjust the positions of intermediate nodes to diminish the cost. Here we do no change the radii of the hubs. In one iteration we consider an intermediate node u of $\widetilde{G}$ and try to find a valid position p_u, which maximally decreases *routing cost.* We assume that our optimizations do not change the capacity segments crossed by each path; hence, we only try to find the local minimum of ink and path lengths components of the cost. Consider the function representing the contribution of node u to *routing cost*: $f(p_u) = k_{ink} \sum_v |p_u p_v| + k_{len} \sum_{\pi_{st}} (|p_u p_v| + |p_u p_w|)/|st|$, where the first sum is taken over all neighbors of u and the second sum is taken over all paths passing through nodes v, u, w. To minimize $f(p_u)$, we use the gradient descent method by moving u along the direction $k_{ink} \sum_v \frac{\overrightarrow{p_u p_v}}{|p_u p_v|} + k_{len} \sum_{\pi_{st}} (\frac{\overrightarrow{p_u p_v}}{|p_u p_v|} + \frac{\overrightarrow{p_u p_w}}{|p_u p_w|})/|st|$ using a small fixed step size. If *routing cost* is reduced and the new position is still valid, we repeat the procedure; otherwise, the move is discarded and we proceed with the next node. We pass a predefined number of times over all nodes of the graph; this number is 10 in our implementation.

After this optimization procedure we also try to modify the structure of graph $\widetilde{G}$, if it is beneficial. We try the following modifications: (1) shortcut an intermediate node of degree two;

(2) glue adjacent intermediate nodes together. These transformations are applied only if they reduce *routing cost* while keeping the graph valid.

What is the complexity of the above procedure? Let c be the time required to find out if a circle or a straight-line segment intersects an obstacle. Using an R-tree [27] on the obstacles, one can find out if a circle or a rectangle intersects the obstacles in $\mathcal{O}(c \log |\widetilde{V}|)$ time. The number of edges in $\widetilde{G}$ is $|\widetilde{E}|$; therefore, an iteration optimizing the position of every node of $\widetilde{G}$ can be done in $\mathcal{O}(c|\widetilde{E}| \log |\widetilde{V}|)$ time. Since we apply a constant number of iterations, this is also a bound for the whole procedure.

6. Ordering Paths

At this point the routing is completed and the bundles have been defined. We draw the paths of a given bundle parallel to the corresponding edge, therefore two paths may need to cross at a node as shown in Figure 7. The order of paths in bundles affects crossings of paths. Some path crossings cannot be avoided since they are induced by the edge crossings of $\widetilde{G}$, while others depend on the relative order of paths along their common edges, and thus may be avoided. We would like to find the orders such that only unavoidable crossings remain. Let P be the set of simple paths in $\widetilde{G}$ computed by path routing. We address the following problem.

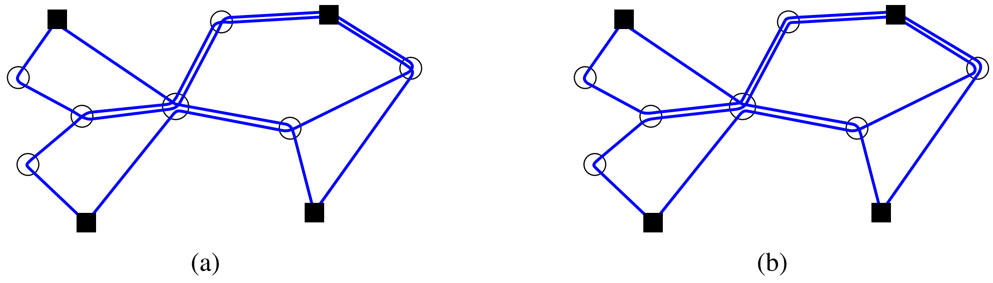


Figure 7: Graph $\widetilde{G}$ with 4 terminal (squares) and 7 intermediate (circles) nodes. (a) An order with 6 crossings. (b) An optimal order with 3 crossings.

Problem 4 (MLCM). *Given the embedded graph $\widetilde{G}$ and a set of simple paths P, find an order of paths for each edge of $\widetilde{G}$ that minimizes the total number of crossings among all pairs of paths.*

Problem 4 has been recently shown to be NP-hard [21]. However, in our setting paths satisfy an additional property that makes it possible to solve the ordering problem efficiently. Let us describe this property.

For a path $(v_1, \ldots, v_k) \in P$, the nodes v_1 and v_k are called **terminal** and the nodes $v_2, \ldots, v_{k-1}$ are called **intermediate**. In our setting, the paths terminate at the nodes of $\widetilde{G}$ corresponding to the centers of the obstacles. These are the only nodes that can be terminal for paths. The rest of the nodes are intermediate. Thus we have:
Path Terminal Property: No node of $\widetilde{G}$ is both a terminal of some path and an intermediate of some path.

We consider the following variant of the crossing minimization problem.

Problem 5 (Path Ordering). *Given an embedded graph $\widetilde{G}$ and a set of simple paths P satisfying the path terminal property, compute an order of paths for all edges of $\widetilde{G}$ so that the number of crossings between pairs of paths is minimized.*

The problem is computationally equivalent to the existing variants of the metro-line crossing minimization problems, MLCM-FixedSE and MLCM-T1. That is, an algorithm for Problem 5 can be used to efficiently construct solutions for the two variants [36].

We say that two paths having a common subpath have an **unavoidable crossing** if they cross for every order of paths. An order of paths is **consistent** if only the pairs of paths with unavoidable crossings cross (and these only once). Clearly, a consistent order has the minimum possible number of crossings. We will prove that a consistent order always exists, and we provide two algorithms to construct one, solving Problem 5. The first algorithm is a modification of a method for wire routing in VLSI design proposed in [26]. We call it the "simple algorithm" since it utilizes a simple idea and it is straightforward to implement. The second is a new algorithm that has a better computational complexity.

6.1. Simple Algorithm

To find an order of paths, we iterate over the edges of $\widetilde{G}$ in any order and build the order for each edge. Let P_{uv} (P_v) be the set of paths passing through edge $(u, v) \in \widetilde{E}$ (node $v \in \widetilde{V}$). We fix a direction of the edge and construct the order on it by sorting P_{uv}. To define the order between two paths $\pi_1, \pi_2 \in P_{uv}$, we walk along their common edges starting from u until the paths end or one of the following cases happen. (a) If we find an edge that was previously processed by the algorithm, we reuse the order of π_1 and π_2 generated for this edge. (b) If we find a fork node f (that is, the end of the common subpath of the two paths), then the order between π_1 and π_2 is determined according to the positions of the next nodes of the paths following f, the path turning to the left precedes the other one in the resulting order. (c) If such a fork node is not found (which means that the paths end at the same node), we walk along the common subpath in the reverse direction starting from node v, and again looking for a previously processed edge or a fork. Note that the case when one path ends and the other one continues cannot happen due to the path terminal property. The above procedure determines a consistent order for all pairs except of the pairs of coincident paths; such paths are ordered arbitrarily on the edge.

It is easy to see that this ordering creates unavoidable crossings only. As mentioned earlier, the above algorithm is a minor variant of one in [26]. The proof of correctness follows the same lines as in [26]. Here we discuss the main differences between the two variants. First, for the sake of simplicity, we do not modify the graph and the paths during computations. Second, in our setting the nodes of $\widetilde{G}$ may have an arbitrary degree; in particular, several paths may share an endpoint, and thus the degree of the terminals is not bounded. That is why to compare two paths, we traverse their common subpath in both directions. The running time of our algorithm is also different from that in [26]. Our algorithm iterates over the edges of $\widetilde{G} = (\widetilde{V}, \widetilde{E})$, which takes $\mathcal{O}(|\widetilde{E}|)$ time. For each edge, it sorts paths passing through it. A comparison of two paths requires at most $\mathcal{O}(|\widetilde{V}|)$ steps (maximal length of a common subpath). Let $|P_e|$ be the number of paths passing through edge $e \in \widetilde{E}$. The sorting on edge e takes $\mathcal{O}(|\widetilde{V}||P_e| \log |P_e|)$ time. Since $\sum_{e \in \widetilde{E}} |P_e| = L$, where L is the total length of paths in P and $|P_e| \leq |P|$, the algorithm runs in $\mathcal{O}(|\widetilde{V}|L \log |P| + |\widetilde{E}|)$ time.

However, this is a pessimistic bound. Normally, the length of a common subpath of two paths is much smaller than $|\widetilde{V}|$, and the number of paths passing through an edge is smaller than $|P|$. That makes the algorithm quite fast in practice.

The above algorithm is straightforward to implement. The following slightly more complicated algorithm has a better asymptotic computational complexity.

6.2. Linear-Time Algorithm

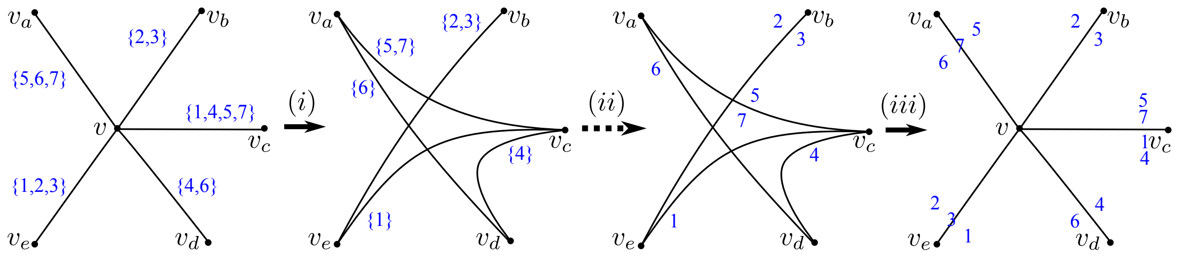


Figure 8: *(i)* Removal of intermediate node v; *(ii)* intervening steps; *(iii)* re-insertion of v. Paths of P_v are numbered $1, \dots, 7$ and are shown in braces when unordered, and then with placement indicating their orders. The removal of v defines the order between paths 4 and 6 on edge (v, v_d). The orders between paths 2 and 3 and between paths 5 and 7 were calculated in the intervening steps.

We assume that the clockwise order of edges around nodes of $\widetilde{G}$ is given. The algorithm consists of two phases. Each step of *Phase 1* involves the deletion of a node of $\widetilde{G}$; see Figure 8. For each intermediate node v in turn, we do the following, which modifies graph $\widetilde{G}$, the clockwise order of edges around the nodes of $\widetilde{G}$, and the paths. Enumerate the edges incident to v as $e_1, e_2, \dots, e_d$ in clockwise order, and let $v_1, \dots, v_d$ be the corresponding nodes adjacent to v. Represent each path $\pi \in P_v$ using edges e_a and e_b by an unordered pair $\{a, b\}$. For each pair $\{a, b\}$, add a new edge (v_a, v_b). Modify the paths represented by $\{a, b\}$ so that it goes directly from v_a to v_b using this new edge. The new edges incident to v_a, that is $\{(v_a, v_b) : b \neq a\}$, should be inserted into the clockwise order of v_a in the position previously occupied by e_a. Finally, delete node v and all adjacent edges from the graph.

In *Phase 2* the process is inverted. Initially, all intermediate nodes have been deleted, and we re-insert them in the reverse order to which they were deleted, adding orders of paths to the edges in the process. Consider the re-insertion of node v. The new order of paths along edge e_a is obtained by concatenating the orders of paths along the edges $\{(v_a, v_b) : b \neq a\}$.

Theorem 1. *Given graph $\widetilde{G} = (\widetilde{V}, \widetilde{E})$, a set of simple paths P satisfying the path terminal property, and a clockwise order of the edges around each node, the above algorithm computes a consistent order of paths in $\mathcal{O}(|\widetilde{V}| + |\widetilde{E}| + L)$ time, where L is the total length of paths in P.*

Proof: *Correctness*. Due to the path terminal property, removing an intermediate node from $\widetilde{G}$ yields another graph satisfying the property. Hence, every step of the algorithm can be performed. Note that at the end of the first phase, the graph consists of terminal nodes connected by (possibly multi-) edges; all the paths consist of two nodes.

Let us show that the order of paths computed by our algorithm is consistent. Indeed, the deletion of node v may add a crossing between two paths π_1 and π_2 only when v is the last deleted node of their common subpath. Suppose that $\pi_1 = (\dots, v_a, v, v_{a'}, \dots)$ and $\pi_2 = (\dots, v_b, v, v_{b'}, \dots)$. If the clockwise order of nodes around v is $\cdots v_a \cdots v_b \cdots v_{a'} \cdots v_{b'} \cdots$, then the removal of v creates a crossing between π_1 and π_2; the crossing is clearly unavoidable. Otherwise (if the clockwise order around v is $\cdots v_a \cdots v_{a'} \cdots v_b \cdots v_{b'} \cdots$), the crossing is not created. The second phase of our algorithm is straightforward and only concatenates the orders of paths; thus, it does not affect the number of crossings.

Running time. In order to prove the computational complexity of our algorithm, we provide details of the implementation. For the first phase, create a new graph H. Initially, H is the subgraph of $\widetilde{G}$ induced by the union of all the paths of P. Each path in P is stored as a list of nodes in H. For an edge e of H, let P_e denote the set of paths containing e. We assume that for every node v of H, the list of edges incident to v in clockwise order is given. Note that these lists are dynamic since H undergoes deletions of nodes. We keep track of the deletions in a directed acyclic graph F; the graph is ordered, that is, the order of outgoing edges for every node of F is important. Initially, F consists of isolated nodes corresponding to the edges of H. When a node is deleted and an edge e is replaced by new edges, we add them in F as (direct) successors of the node corresponding to e. Note that every new edge is a successor of exactly two nodes in F. For instance, consider $P_{v_c v}$ containing paths 1,4,5, and 7 in Figure 8. When node v is processed, set $P_{v_c v}$ is split into new subsets $P_{v_c v_d}$, $P_{v_c v_e}$, and $P_{v_c v_a}$. The clockwise order of the subsets is important. Then we replace edge (v_c, v) by edges (v_c, v_d), (v_c, v_e), and (v_c, v_a) in graph H, in all paths of $P_{v_c v}$, and in the clockwise order of edges around v_c. In graph F, the node corresponding to edge (v_c, v) gets three successors (in this order) corresponding to (v_c, v_d), (v_c, v_e), and (v_c, v_a). We stress that graph H may contain multiple edges after the operation, since there could be another path from v_c to any of v_d, v_e, or v_a; it is important to keep multiple edges and the corresponding sets of paths separately. The first phase finishes when all non-terminals are deleted from H.

The time for treating node v (the deletion of v) is $\mathcal{O}(d_{v,H} + |P_v|)$, where $d_{v,H}$ is the degree of v in H at the current step and $|P_v|$ is the number of paths passing through v. The bound can be achieved utilizing radix sort with buckets corresponding to the neighbors of v. The sorting is performed for all paths of P_v simultaneously, producing the clockwise orders of the created edges around neighbors of v. Since $d_{v,H} \le d_{v,\widetilde{G}} + |P_v|$, the complexity of *Phase 1* is $\mathcal{O}(|\widetilde{V}| + |\widetilde{E}| + L)$.

In the second phase, we process nodes of F in bottom-up order from successors to predecessors. The list of paths for a node of F is simply the concatenation of the lists of its children. The leaves of F correspond to the original paths of P. A concatenation of two lists is done in constant time, as we do not copy the lists; hence, the total complexity of *Phase 2* is $\mathcal{O}(|\widetilde{E}| + L)$. $\square$

We note that the size of the input for Problem 5 is $\Omega(|\widetilde{V}| + |\widetilde{E}| + L)$, which is the same as the running time of our algorithm if the clockwise order of edges around nodes is a part of the input. If this order is not given in advance, the time complexity of the algorithm is $\mathcal{O}(|\widetilde{V}| \log |\widetilde{V}| + |\widetilde{E}| + L)$. Since the length of a path does not exceed $|\widetilde{V}|$, we have $L = |\widetilde{V}||P|$. Therefore, another estimate for the running time is $\mathcal{O}(|\widetilde{V}| \log |\widetilde{V}| + |\widetilde{E}| + |\widetilde{V}||P|)$.

6.3. Distribution of Crossings

Consistent orders are not necessarily unique. The choice of a particular one may greatly influence the quality of the final drawing [6, 22]. Arguably, the following property may appear desirable. An order of paths is **nice** if it is consistent, and for any two paths, their relative order along all their common edges is the same, that is, they may cross only at an endpoint of their common subpath. We next analyze the existence of nice orders.

Lemma 2. *If $\widetilde{G}$ is a tree, then there is a nice order of paths P.*

Proof: To build a nice order of paths on a tree, we use the algorithm described in Section 6.1 processing the edges in the following order. Choose any terminal node to be the root of the tree, and then consider the edges starting at the leaves towards the root. The first time we encounter a common edge of two paths will be at the end of their common subpath. Thus, the crossing between them (if needed) will take place on an endpoint of that edge. To ensure that the crossing goes to the endpoint of the subpath, we direct the considered edge towards the root of the tree. □

Unfortunately, for general graphs, there is an example having no nice order.

Lemma 3. *There exists a graph $\widetilde{G}$ with paths P that admits no nice order.*

Proof: Consider a gadget with four paths shown in Figure 9. It consists of a black path, two blue paths (called left and right blue path respectively) and a central red path. For every nice order, one of the following conditions holds. Either the left blue path must be above the black path, or the right blue path must be below the black path. Otherwise, if the left blue path is drawn below and the right blue path is drawn above the black path, then red and black paths do not intersect nicely, that is, their order along common edges is not the same. We use the property of the gadget to prove the lemma.

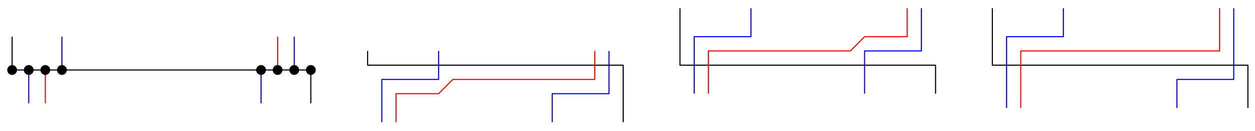


Figure 9: The gadget and 3 ways to draw it.

Consider graph $\widetilde{G}$ with 7 gadgets shown in Figure 10(a), where only black paths are shown. Every black path starts and ends at leafs with the same number. The gadgets with a common subpath share a blue path. For example, gadgets 1, 2, and 4 share a blue path, which is left for gadget 1 and right for gadgets 2 and 4. Similarly, gadgets 2 and 3 share a blue path; the blue path is left for gadget 2 and right for gadget 3. In total, graph $\widetilde{G}$ contains 7 blue paths, 7 black paths, and 7 red paths (one per gadget).

As noticed above, in a nice order either (a) the left blue path of gadget 1 is above its black path, or (b) its right blue path is below the black path. In case (a), the right blue path of gadget 2 is above its black path; see Figures 10(b), where we mark the position of blue paths relative to black paths by a blue dot. Then the left blue path of gadget 2 must be above its black path, using the property of the gadgets. Since the blue path is also the right one for gadget 3, the left blue path of gadget 3 must be above its black path. Again, the blue path is the left one for gadget 4, and it must below its black path. This contradicts to the property of the gadget for the right blue path of gadget 4. Hence, case (a) is impossible.

In case (b), we similarly "propagate" the states of the gadgets and get a contradiction with the left blue path of gadget 5; see Figures 10(c). This proves that there is no nice order of considered paths on $\widetilde{G}$. □

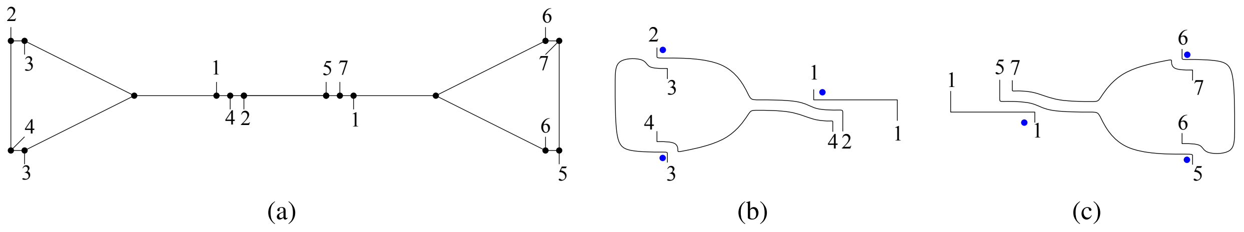


Figure 10: (a) Graph $\widetilde{G}$ consisting of 7 gadgets; each black path starts at a leaf marked by a number and ends at a leaf with the same number. (b-c) Two possible cases: the blue dots mark the location of the gadget's blue paths.

Since in general a nice order may not exist, what is the best consistent order? We suggest to distribute path crossings evenly among intermediate nodes of $\widetilde{G}$. In other words, we propose to avoid dense regions in the drawing with many crossings. To this end, we utilize the following heuristic. The edges of $\widetilde{G}$ in the simple algorithm are processed in the increasing order of the number of paths passing through an edge. Note that the algorithm tries to realize all unavoidable crossings on an endpoint of the considered edge; hence, processing first the edges with few paths (and thus, few possible crossings) has the desired effect. Similarly, we delete the nodes of $\widetilde{G}$ in the linear-time algorithm in the decreasing order of the number of paths passing through a node.

7. Results

We implemented our algorithm in the MSAGL package [34], which is a part of Visual Studio Ultimate 2012 and higher versions; see a real example produced in the programming environment in Figure 1(b). Unfortunately, this version of MSAGL currently is not available for public. The algorithm is implemented in `C#`; we run all experiments on a 3.1 GHz quad-core machine with 4GB of RAM. Edge bundling was applied for several real-world and synthetic graphs; see [39] for a detailed description of our dataset. Unless node coordinates are available as a part of the input, we used the MSAGL package to position the nodes. Table 1 provides statistics and some measurements of the method on test graphs.

For the path routing step of our algorithm, we sort the graph edges based on the Euclidean distance between the corresponding obstacle centers, and then use the Dijkstra's algorithm to route the edges one by one, as described in Section 4.3. The second step follows the description in Section 5. For the path ordering, we implemented both methods presented in Sections 6.1 and 6.2. By default, the simple algorithm is utilized, as we have not found any consistent visual difference in the drawings produces by these two methods. We do not apply any crossing distribution strategy, but we process the edges in the simple algorithm in the increasing order of the number of paths in

Graph	Source	$\|V\|$	$\|E\|$	$\|\widetilde{V}\|$	$\|\widetilde{E}\|$	Routing	Optimization	Ordering	Total
`tail`	[39]	21	68	105	348	0.13	0.17	0.01	0.34
`notepad`	[39]	22	113	198	776	0.03	0.29	0.01	0.35
`airlines`	[11]	235	1297	1175	5297	0.47	2.67	0.15	3.32
`jazz`	[44]	191	2732	955	4478	0.56	3.25	0.18	4.04
`protein`	[44]	1458	1948	7290	32585	1.45	11.92	0.10	13.52
`power grid`	[44]	4941	6594	24705	109779	7.97	18.01	0.18	26.31
`Java`	GD'06 Contest	1538	7817	7690	32712	4.08	30.17	0.96	35.35
`migrations`	[11]	1715	6529	8575	41451	3.89	30.57	1.16	35.75

Table 1: Statistics on test graphs and performance of the algorithm steps (in seconds).

a bundle. The final step, the construction of smooth curves, uses a straightforward implementation described in Section 5.1 and follows [37] to construct biarcs.

7.1. Performance

Table 1 shows the CPU times of the main algorithm steps. Recall that the computational complexity of the *Routing*, *Optimization*, and *Ordering* steps in our implementation is $\mathcal{O}(|E||\widetilde{E}|\log|\widetilde{V}|)$, $\mathcal{O}(|\widetilde{E}|\log|\widetilde{V}|)$, and $\mathcal{O}(|\widetilde{V}|^2|E|\log|E|+|\widetilde{E}|)$, respectively. Since by construction $\widetilde{V}=\mathcal{O}(|V|)$ and $\widetilde{E}=\mathcal{O}(|V|)$, it amounts to $\mathcal{O}(|V||E|\log|V|+|V|\log|V|+|V|^2|E|\log|E|)=\mathcal{O}(|V|^2|E|\log|E|)$ in terms of the original input graph $G=(V,E)$. However, this bound is too pessimistic.

As can be seen from the table, ordered bundles can be constructed for graphs with several thousand of nodes and edges in less than a minute. For graphs of medium size with hundreds of nodes, the algorithm can be applied in an interactive environment. Overall, this is slower than the standard visibility-based approach [17] and some of the edge bundling techniques [19, 23], but sufficient for drawing small and medium sized graphs improving the quality of single edge routes when compared to existing methods.

The first step, *Path Routing*, is usually quite fast; it is less efficient for densely embedded graphs in which the paths make long detours to avoid narrow channels between the obstacles. Although *Path Ordering* has the worst time complexity among all the steps, it is the fastest step in practice. The reason is that the theoretical worst case is rather unrealistic; in the real-world examples, two paths share only a short subpath and the number of path in a bundle is low. The most time expensive step is *Routing Graph Optimization*. The explanation for its long running time is twofold. Firstly, it continues to modify $\widetilde{G}$ as long as it reduces *routing cost*, and we do not have a good upper bound for the number of iterations. Secondly, the step utilizes a lot of geometric operations such as checking an intersection between circles, polygons and line segments. These operations are theoretically constant time but computationally expensive in practice. The optimization of this step is the primary direction for the running time improvement of our algorithm. For example, utilizing a specialized geometric library will likely improve the performance. The last step of our algorithm, the construction of smooth curves for the paths, takes few milliseconds for the graphs and are not included in Table 1.

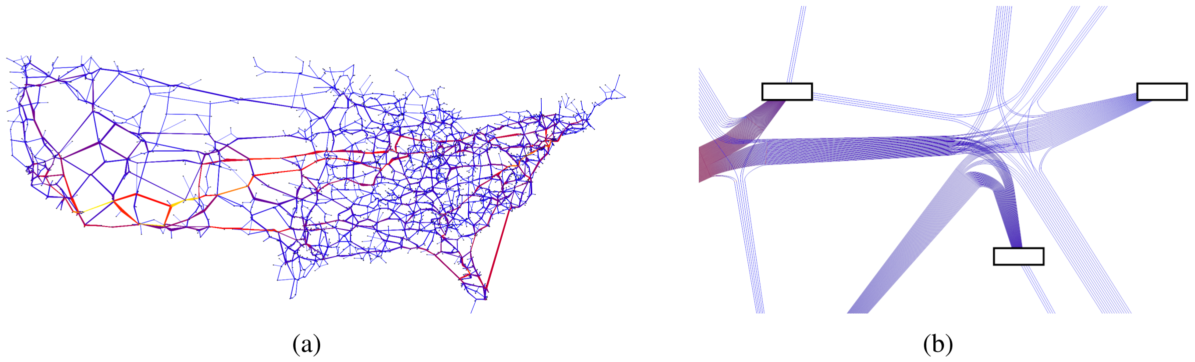


Figure 11: `Migration` graph: (a) overview, (b) detail.

7.2. *Selection of parameters*

We describe the influence of the parameters in *routing cost* on the final drawing. The weight of the ink component, k_{ink}, can be considered as the "bundling strength", since larger values of k_{ink} encourage more bundling. The path length component is complimentary and has the opposite effect. The capacity component plays an important role for graphs with large and closely located node labels (see for example the narrow channel between nodes $S2$ and $S5$ in Figure 13(b)). Based on our experiments, we recommend selecting k_{len} significantly larger than k_{ink}, otherwise some paths can be too long. The capacity weight, k_{cap}, is set larger by order of magnitude than the other coefficients. In our default settings $k_{ink} = 1$, $k_{len} = 500$, and $k_{cap} = 10(k_{ink} + k_{len})$.

We may also adjust the widths of individual paths and path separation; see Figures 1(b) and 13. In the extreme case when the path separation and path widths are set to zero, we obtain a drawing with overlapping edges.

7.3. *Examples*

We now demonstrate our ordered edge bundling algorithm on some real-world examples. The *migration* graph used for comparison of edge bundling algorithms is shown in Figure 11. In our opinion, on a global scale ordered edge bundles are aesthetically as pleasant as other drawings of the graph (see e.g., [11, 23, 29, 31]). On a local scale, our method outperforms previous approaches by arranging edges and crossings between them. Arguably, the results of ordered edge bundling are easier to analyze when one is interested in individual connections between graph nodes; in that sense, our results are less ambiguous. A smaller example of edge bundling is given in Figure 12. It shows another important advantage of our routing scheme. Multiple edges are visualized separately making them easier to follow; compare the edge between nodes *Editor* and *Application* on the original and bundled drawings. Note also that the edges never obscure node labels, which is crucial in some applications.

7.4. *Limitations*

The main limitation of our approach is the computationally expensive routing graph optimization step. We are not aware of a good method for choosing radii and positions for hubs. As the path routing and the path optimization is done independently, sometime we do not utilize the available

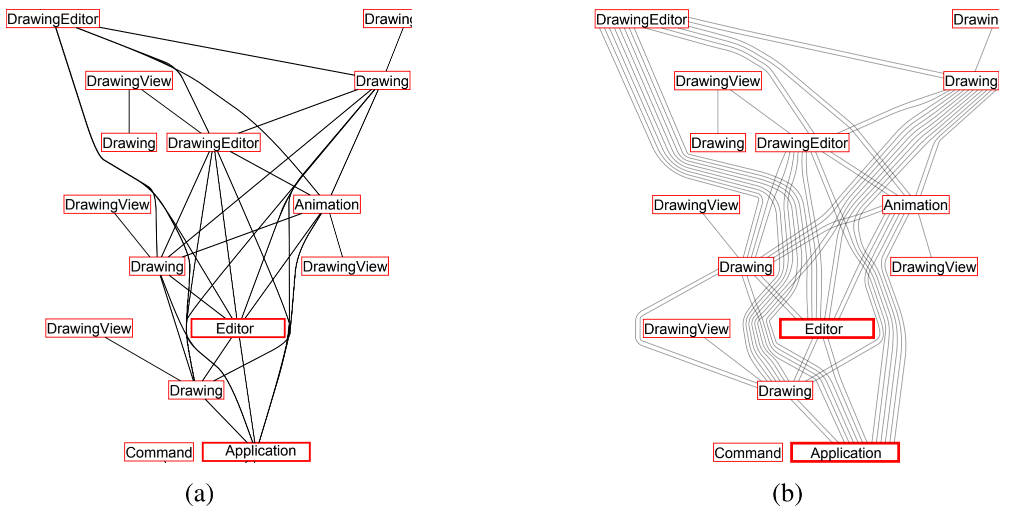


Figure 12: `Tail` graph: (a) standard routing computed with the visibility graph approach, (b) edge bundling routing.

space in the best way. As a future work, we consider two directions for improving the heuristic. One approach is to integrate the routing and optimization steps in order to get lower routing cost and better space utilization. One requirement here is to keep the computational complexity of the procedure low, as the algorithm is used in an interactive environment. Another approach concerns the choice of the routing graph. For example, the greedy visibility spanner or a suitable refinement of the Delaunay triangulation have certain geometric properties (e.g., large angles between crossing/incident edges) that may simplify (or even make unnecessary) the path optimization step.

8. Conclusions and Future Work

We have presented a new edge routing algorithm based on ordered bundles that improves the quality of single edge routes when compared to existing methods. Our technique differs from classical edge bundling, in that the edges are not allowed to actually overlap, but are run in parallel channels. The algorithm ensures that the nodes do not overlap with the bundles and that the resulting edge paths are relatively short. The resulting layout highlights the edge routing patterns and shows significant clutter reduction.

An important contribution of the paper is an efficient algorithm that finds a consistent order of edges inside of bundles with the minimal number of crossings. As the consistent order with the minimum number of crossings is not unique and a nice order may not exist, the question of computing the "best" order remains open. For example, an edge is easy to follow if it has few bends. Hence, an interesting question is how to visualize a bundled graph using the minimum number of edge bends.

Another possible future direction concerns dynamic issues of edge bundling algorithm. First, a user may want to interactively change a bundled graph or modify node positions. In that case, a system should not completely rebuild a drawing, but recalculate affected parts only. Second, a

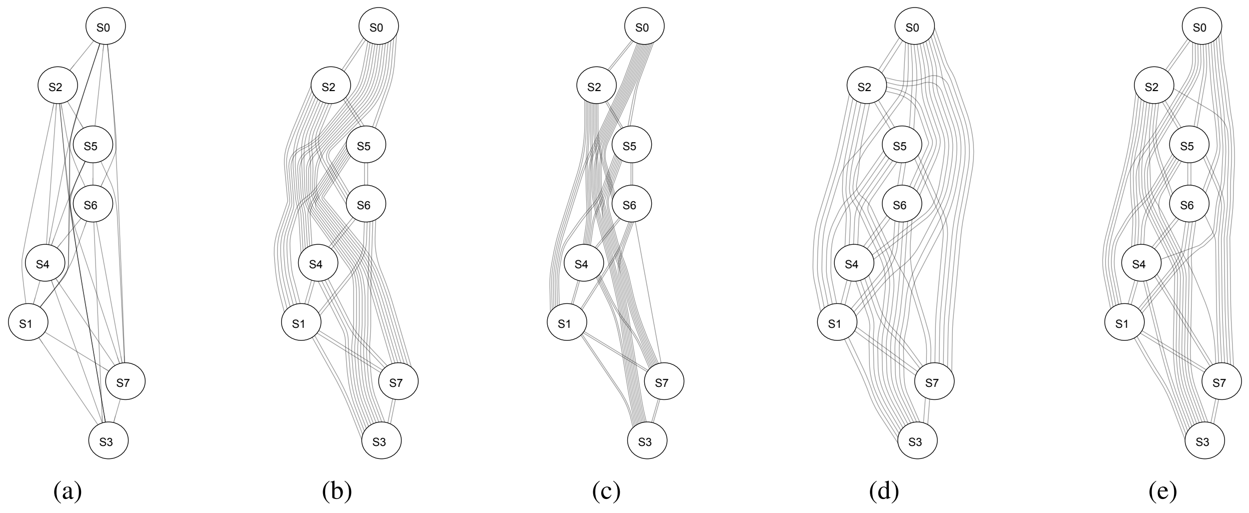


Figure 13: (a) Standard edge routing produced with the visibility graph approach with many edge overlaps. (b) No capacity overflow penalty, $k_{cap} = 0$. (c) Small edge separation. (d) Large edge separation. (e) Ordered edge bundling with the default settings.

small deviation of the algorithm parameters (e.g., ink importance k_{ink}) may theoretically involve a full reconstruction of the routing, while a smooth transformation is preferable. Finally, in the cases when the node positions are not naturally defined, it may be possible to allow small adjustments of the given embedding in order to improve the routing.

From theoretical point of view, it would be interesting to further explore Problem 1. An approximation algorithm for the problem or one of its variants is a natural goal.

References

[1] E. Argyriou, M. A. Bekos, M. Kaufmann, and A. Symvonis. Two polynomial time algorithms for the metro-line crossing minimization problem. In I. G. Tollis and M. Patrignani, editors, *International Symposium on Graph Drawing*, volume 5417 of *Lecture Notes in Computer Science*, pages 336–347. Springer, 2009.

[2] M. Asquith, J. Gudmundsson, and D. Merrick. An ILP for the metro-line crossing problem. In J. Harland and P. Manyem, editors, *Symposium on Computing: the Australasian Theory*, volume 77 of *CRPIT*, pages 49–56. Australian Computer Society, 2008.

[3] M. Bar and M. Neta. Humans prefer curved visual objects. *Psychological science*, 17(8):645–648, 2006.

[4] M. A. Bekos, M. Kaufmann, K. Potika, and A. Symvonis. Line crossing minimization on metro maps. In S.-H. Hong, T. Nishizeki, and W. Quan, editors, *International Symposium on Graph Drawing*, volume 4875 of *Lecture Notes in Computer Science*, pages 231–242. Springer, 2008.

[5] M. Benkert, M. Nöllenburg, T. Uno, and A. Wolff. Minimizing intra-edge crossings in wiring diagrams and public transportation maps. In M. Kaufmann and D. Wagner, editors, *International Symposium on Graph Drawing*, volume 4372 of *Lecture Notes in Computer Science*, pages 270–281. Springer, 2007.

[6] S. Bereg, A. E. Holroyd, L. Nachmanson, and S. Pupyrev. Drawing permutations with few corners. In S. K. Wismath and A. Wolff, editors, *International Symposium on Graph Drawing*, volume 8242 of *Lecture Notes in Computer Science*, pages 484–495. Springer, 2013.

[7] M. Cermák, J. Dokulil, and J. Katreniaková. Edge routing and bundling for graphs with fixed node positions. In *International Conference on Information Visualisation*, pages 475–481. IEEE Computer Society, 2011.

[8] H.-F. S. Chen and D. T. Lee. On crossing minimization problem. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(5):406–418, 1998.

[9] K. Clarkson. Approximation algorithms for shortest path motion planning (Extended Abstract). In A. V. Aho, editor, *Symposium on Theory of Computing*, pages 56–65. ACM, ACM, 1987.

[10] R. Cole and A. Siegel. River routing every which way, but loose (Extended Abstract). In *Symposium on Foundations of Computer Science*, pages 65–73. IEEE Computer Society, 1984.

[11] W. Cui, H. Zhou, H. Qu, P. C. Wong, and X. Li. Geometry-based edge clustering for graph visualization. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1277–1284, 2008.

[12] W. W.-M. Dai, T. Dayan, and D. Staepelaere. Topological routing in SURF: Generating a rubber-band sketch. In *ACM/IEEE Design Automation Conference*, pages 39–44, New York, NY, USA, 1991. ACM.

[13] M. Dickerson, D. Eppstein, M. T. Goodrich, and J. Y. Meng. Confluent drawings: Visualizing non-planar diagrams in a planar way. *Journal of Graph Algorithms and Applications*, 9(1):31–52, 2005.

[14] D. Dobkin, E. Gansner, E. Koutsofios, and S. North. Implementing a general-purpose edge router. In G. D. Battista, editor, *International Symposium on Graph Drawing*, volume 1353 of *Lecture Notes in Computer Science*, pages 262–271. Springer, 1997.

[15] V. Domiter and B. Zalik. Sweep-line algorithm for constrained Delaunay triangulation. *International Journal of Geographical Information Science*, 22(4):449–462, 2008.

[16] T. Dwyer, K. Marriott, and M. Wybrow. Integrating edge routing into force-directed layout. In M. Kaufmann and D. Wagner, editors, *International Symposium on Graph Drawing*, volume 4372 of *Lecture Notes in Computer Science*, pages 8–19. Springer, 2007.

[17] T. Dwyer and L. Nachmanson. Fast edge-routing for large graphs. In D. Eppstein and E. R. Gansner, editors, *International Symposium on Graph Drawing*, volume 5849 of *Lecture Notes in Computer Science*, pages 147–158. Springer, 2009.

[18] D. Eppstein, M. T. Goodrich, and J. Y. Meng. Confluent layered drawings. *Algorithmica*, 47(4):439–452, 2007.

[19] O. Ersoy, C. Hurter, F. V. Paulovich, G. Cantareiro, and A. Telea. Skeleton-based edge bundling for graph visualization. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2364–2373, 2011.

[20] M. Fink, H. Haverkort, M. Nöllenburg, M. Roberts, J. Schuhmann, and A. Wolff. Drawing metro maps using bézier curves. In W. Didimo and M. Patrignani, editors, *International Symposium on Graph Drawing*, volume 7704 of *Lecture Notes in Computer Science*, pages 463–474. Springer, 2013.

[21] M. Fink and S. Pupyrev. Metro-line crossing minimization: Hardness, approximations, and tractable cases. In S. K. Wismath and A. Wolff, editors, *International Symposium on Graph Drawing*, volume 8242 of *Lecture Notes in Computer Science*, pages 328–339. Springer, 2013.

[22] M. Fink and S. Pupyrev. Ordering metro lines by block crossings. In K. Chatterjee and J. Sgall, editors, *Mathematical Foundations of Computer Science*, volume 8087 of *Lecture Notes in Computer Science*, pages 397–408. Springer, 2013.

[23] E. Gansner, Y. Hu, S. North, and C. Scheidegger. Multilevel agglomerative edge bundling for visualizing large graphs. In G. D. Battista, J.-D. Fekete, and H. Qu, editors, *Pacific Visualization Symposium*, pages 187–194. IEEE, IEEE, 2011.

[24] E. R. Gansner and Y. Koren. Improved circular layouts. In M. Kaufmann and D. Wagner, editors, *International Symposium on Graph Drawing*, volume 4372 of *Lecture Notes in Computer Science*, pages 386–398. Springer, 2006.

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

[26] P. Groeneveld. Wire ordering for detailed routing. *Design & Test of Computers*, 6:6–17, 1989.

[27] A. Guttman. R-trees: A dynamic index structure for spatial searching. In B. Yormark, editor, *International Conference on Management of Data*, pages 47–57. ACM Press, 1984. SIGMOD Record 14(2).

[28] D. Holten. Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):741–748, 2006.

[29] D. Holten and J. J. van Wijk. Force-directed edge bundling for graph visualization. *Computer Graphics Forum*, 28(3):983–990, 2009.

[30] C. Hurter, O. Ersoy, and A. Telea. Graph bundling by kernel density estimation. In *Computer Graphics Forum*,

volume 31, pages 865–874. Wiley Online Library, 2012.

[31] A. Lambert, R. Bourqui, and D. Auber. Winding roads: Routing edges into bundles. *Computer Graphics Forum*, 29(3):853–862, 2010.

[32] M. Marek-Sadowska and M. Sarrafzadeh. The crossing distribution problem [IC layout]. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 14(4):423–433, 1995.

[33] M. Middendorf and F. Pfeiffer. On the complexity of the disjoint paths problems. *Combinatorica*, 13(1):97–107, 1993.

[34] L. Nachmanson, G. Robertson, and B. Lee. Drawing graphs with GLEE. In S.-H. Hong, T. Nishizeki, and W. Quan, editors, *International Symposium on Graph Drawing*, volume 4875 of *Lecture Notes in Computer Science*, pages 389–394. Springer, 2007.

[35] F. J. Newbery. Edge concentration: A method for clustering directed graphs. In *International Workshop on Software Configuration Management*, pages 76–85, 1989.

[36] M. Nöllenburg. An improved algorithm for the metro-line crossing minimization problem. In D. Eppstein and E. R. Gansner, editors, *International Symposium on Graph Drawing*, volume 5849 of *Lecture Notes in Computer Science*, pages 381–392. Springer, 2009.

[37] L. A. Piegl and W. Tiller. Biarc approximation of NURBS curves. *Computer-Aided Design*, 34(11):807–814, 2002.

[38] S. Pupyrev, L. Nachmanson, S. Bereg, and A. E. Holroyd. Edge routing with ordered bundles. In M. J. van Kreveld and B. Speckmann, editors, *International Symposium on Graph Drawing*, volume 7034 of *Lecture Notes in Computer Science*, pages 136–147. Springer, 2012.

[39] S. Pupyrev, L. Nachmanson, and M. Kaufmann. Improving layered graph layouts with edge bundling. In U. Brandes and S. Cornelsen, editors, *International Symposium on Graph Drawing*, volume 6502 of *Lecture Notes in Computer Science*, pages 465–479. Springer, 2010.

[40] H. C. Purchase, J. Hamer, M. Nöllenburg, and S. G. Kobourov. On the usability of Lombardi graph drawings. In W. Didimo and M. Patrignani, editors, *International Symposium on Graph Drawing*, volume 7704 of *Lecture Notes in Computer Science*, pages 451–462. Springer, 2013.

[41] R. Wein, J. van den Berg, and D. Halperin. The visibility-Voronoi complex and its applications. *Computational Geometry: Theory and Applications*, 36(1):66–87, 2007.

[42] K. Xu, C. Rooney, P. Passmore, D.-H. Ham, and P. H. Nguyen. A user study on curved edges in graph visualization. *IEEE Transactions on Visualization and Computer Graphics*, 18(12):2449–2456, 2012.

[43] A. C.-C. Yao. On constructing minimum spanning trees in k-dimensional spaces and related problems. *SIAM Journal on Computing*, 11(4):721–736, 1982.

[44] Gephi dataset. `http://wiki.gephi.org/index.php?title=Datasets`.