Edge Routing with Ordered Bundles

Sergey Pupyrev[1], Lev Nachmanson[2], Sergey Bereg[3], and Alexander E. Holroyd[4]

[1] Ural State University, spupyrev@gmail.com
[2] Microsoft Research, levnach@microsoft.com
[3] University of Texas at Dallas, besp@utdallas.edu
[4] Microsoft Research, holroyd@microsoft.com

Abstract. We propose a new approach to edge bundling. The approach starts by routing the edge paths to minimize a weighted sum of the total length of the paths together with the ink required to draw them. As this problem is NP-hard, we provide an efficient heuristic that finds an approximate solution. Our approach continues by separating edges belonging to the same bundle. To achieve this, we provide a new and efficient algorithm that solves a variant of the metro-line crossing minimization problem. The method creates aesthetically pleasing edge routes that give an overview of the global graph structure, while still drawing each edge separately, without intersecting graph nodes, and with few crossings.

1 Introduction

The core components of most graph drawing algorithms are 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 layouts. The relations between the nodes are difficult to analyze by looking at such layouts. Recently, edge bundling techniques have been developed, in which some edge segments running close are collapsed into bundles to reduce the clutter. 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 allowed edges to overlap nodes, thus obscuring their text or graphics.

We present a novel edge routing algorithm for undirected graphs, which we call **ordered bundles**. This algorithm produces a drawing in a "metro-line" style (see Fig. 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.

Edge routing. In this step the edges are routed along paths, and the overlapping parts are organized into bundles. One approach here has been to minimize the total ink of the paths. However, this often produces excessively long paths. *For this reason we introduce a novel cost function for edge routing, a weighted sum of the ink and total path length.* Minimizing this function forces the paths to share routes, creating bundles,

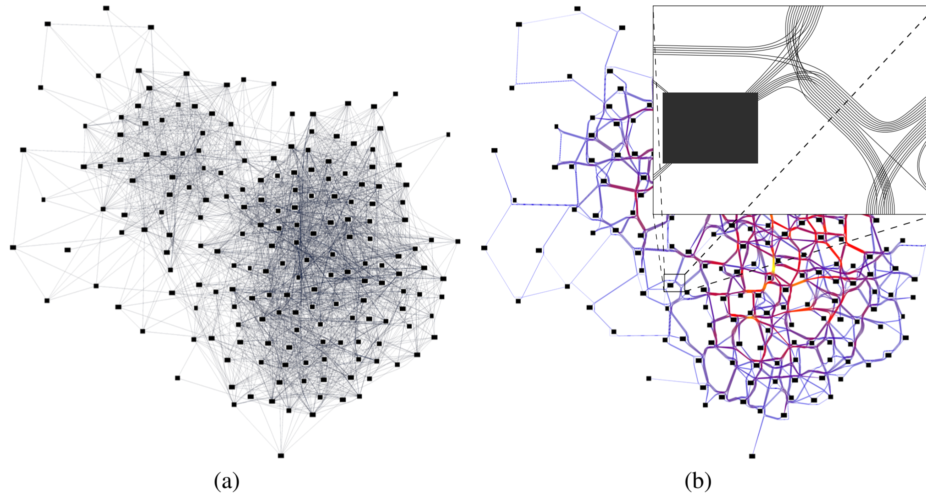

Fig. 1. Edge bundling example (jazz graph)

but at the same time it keeps the paths relatively short. *We furthermore show how to route the bundles outside of the nodes.*

Edge nudging. 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 obtain their 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 might not always exist because of the limited space between the node shapes. We provide a heuristic that finds a drawing with bundles of suitable thickness.

Edge ordering. To route individual edges, an order of the edge segments inside of the bundle needs to be computed. This order minimizes the number of crossings between edges of the same bundle. The problem of finding such an order is related to a variant of the metro-line crossing minimization problem called MLCM-PA [12]. *We provide a new efficient algorithm that solves this problem exactly*.

The next section summarizes related work. In Section 3 we give a detailed explanation of our algorithm. Results of experiments are presented in Section 4. Finally we discuss some additional aspects and future work in Section 5.

2 Related work

We believe that the first use of bundled edges in the graph drawing literature is given in [5]. The authors improve circular layouts by routing edges either on the outer or on the inner face of a circle. Edges in that paper are bundled with 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.

In the hierarchical approach of [8], edges are bundled together based on an additional tree structure. Unfortunately, not every graph comes with a suitable underlying tree, and it is not clear how to extend the method to general layouts. In [13] edge bundles were computed for layered graphs. In contrast, our method applies to general graphs.

Edge bundling methods for general graphs are given in [1, 4, 9, 10]. In the force-directed heuristic [9], edges can attract each other to organize themselves into bundles.

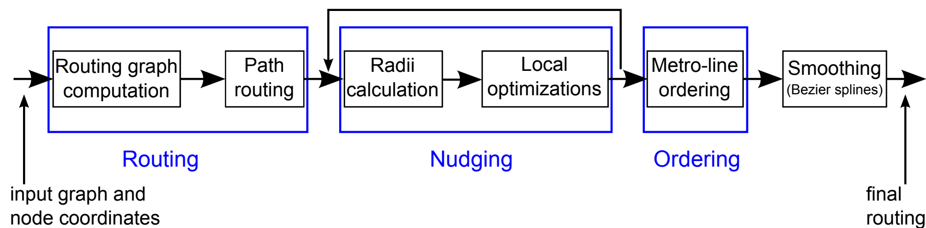

Fig. 2. Algorithm pipeline.

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 edge. The approaches [1, 10] have a common feature: they both create a grid graph for edge routing. Our method also uses a special graph for edge routing, but our approach is different because we modify this graph to obtain better edge bundles. Unlike [1, 4, 9] we avoid edge-node overlaps.

Recently there were several attempts to enhance edge bundled graph visualizations with image-based rendering techniques. Colors and opacity were used to highlight the density of overlapping lines [1, 8]. Shading, luminance, and saturation encode edge types and edge similarity [14]. Our work is focused on the geometry of edge bundles only. Existing rendering techniques can be applied on top of our routing scheme.

The paper [12] inspired us to apply the technique of metro-line routing to minimize edge crossings inside bundles. Related work for orthogonal routings has been done in [15]. The ordering step of [15] tries to orient the paths in a uniform pseudo-direction, and if a path does not confirm to the direction, it is split. This is unnecessary, as our method shows. We build thick edge bundles that avoid the nodes, as the fat edges of [2], but we do not require the graph to be planar.

3 Algorithm

Let us establish some terminology. An undirected graph G is a pair (V, E), where V is the set of nodes and E is the set of edges, i.e. unordered pairs of nodes. A drawing of G is a representation of G in the plane in which each node $v \in V$ is drawn as a convex polygon p_v, and each edge $uv \in E$ is drawn as a simple curve connecting p_u and p_v. We will call the node polygons **obstacles** (since our focus is edge routing), and we assume that they are pairwise disjoint. Following [5, 13], the **ink** of the routing is defined as the sum of the lengths of the line segments drawing the edge paths.

In overview, our algorithm takes the following steps (Fig. 2). We generate a routing graph $\widetilde{G}$ with straight-line edges that avoid the obstacles. We route the edges of G through $\widetilde{G}$. We will refer to an edge of G as a **path** in $\widetilde{G}$. A set of paths sharing the same edge of $\widetilde{G}$ is called a **bundle**. After the initial routing, the paths of the same bundle overlap on its edge. We estimate the space required to draw the paths separately, and we modify the positions of $\widetilde{G}$'s nodes, thus changing the paths' geometry. Then we order each bundle, and draw the paths individually with gaps, according to this order. To complete the drawing, we smooth the paths by fitting Bezier segments into the path corners. Next we give a more detailed description of the steps.

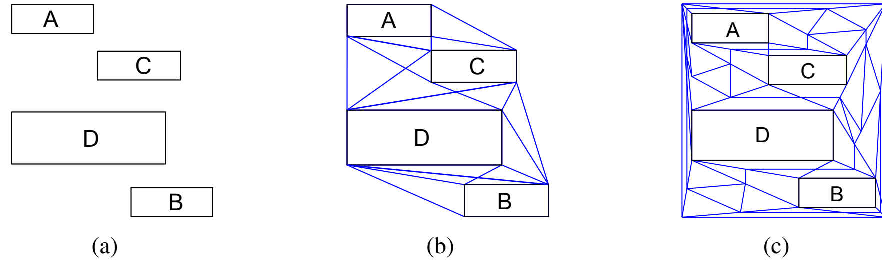

Fig. 3. (a) The obstacles. (b) A visibility graph. (c) A refinement of the Delaunay triangulation.

3.1 Edge Routing

As we mentioned before, minimizing ink only often leads to extremely long paths that are difficult to follow. Therefore, we try to minimize a novel cost function of path routes, which takes both ink and path lengths into account:

$$\textit{routing cost} = \alpha\, ink + \beta \sum_{uv \in E} \ell_{uv}.$$

Here ℓ_{uv} is the length of path uv in the routing. We emphasize that ink is the total length of the union of all the paths: where several paths overlap, their common part is counted only once – in contrast, its contribution to the second term proportional to the number of paths that use it. The non-negative real constants α and β determine the contribution of the components to the function. In our default settings $\alpha = 1$ and $\beta = 500$.

Problem 1 (Edge Routing). Given the graph G with fixed node positions, route the edges of E in the plane so that *routing cost* is minimized.

Problem 1 is NP-hard, because its instance with $\alpha = 1$, $\beta = 0$, and G being a complete graph is equivalent to the Geometric Steiner Tree problem, which is known to be NP-hard [6]. Therefore, we use an approximate solution, which starts with the construction of a routing graph $\widetilde{G}$.

Generation of the Routing Graph We consider two approaches to the routing graph construction. The first one follows [3], which builds a sparse visibility graph (Fig. 3(b)). The second approach is a refinement of the Delaunay triangulation, in which for each two sides of a triangle that are not node polygon sides, we add the segment connecting their midpoints (Fig. 3(c)). This creates routes in the middle of the "channels" between the nodes. Both methods work in $O(n \log n)$ time, and create $\widetilde{G}$ with $O(n)$ edges, where n is the number of node polygon corners. In both cases the edges do not intersect the node interiors. For each node $v \in V$, we add edges connecting the center of polygon p_v to its corners. We denote the set of nodes of $\widetilde{G}$ by W, and the set of edges of $\widetilde{G}$ by U. We write $|W| = n$, and assume that $\widetilde{G}$ contains $O(n)$ edges. We will discuss the effect of the routing graph choice to the final routing later. Next we route the paths on $\widetilde{G}$.

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

Problem 2 (Path Routing). Given the graph $\widetilde{G}$ and a set of pairs of nodes $(a_i, b_i) \in W^2$, find paths between a_i and b_i for all i so that *routing cost* is minimized.

Here a_i and b_i correspond to the centers of obstacles connected by the edges of E. We stress that Problem 2 is different from Problem 1 in that the paths are now constrained to be routed through the edges of $\widetilde{G}$. The Problem 2 is again NP-hard, because its instance with $\beta = 0$ is a Steiner Forest Problem [6]. Therefore 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 the *routing cost* associated to this path. For a path uv we define *additional cost* $= \alpha \, \Delta \, ink + \beta \, \ell_{uv}$. Here $\Delta \, ink$ is the increment in ink, which equals the sum of edge lengths of uv that were not part of any previous path.

Problem 3 (Single Path Routing). Given the graph $\widetilde{G}$ and a set of already routed paths, find a path from a to b so that *additional cost* is minimized.

Let us assign the following weights to edges of $\widetilde{G}$: the weight of edge e is equal to $\alpha \, \delta_e + \beta \, \ell_e$, where δ_e is ℓ_e if e is not taken by a previous path, and 0 otherwise. It can be seen that the minimum *additional cost* is achieved by a shortest path from a to b according to these weights. We can thus apply the Dijkstra algorithm to find a path solving the Problem 3.

To solve Problem 2 approximately, we organize the edges of E in a sequence $(a_1, b_1), \ldots, (a_m, b_m)$, and iteratively solve Problem 3 for already routed paths (a_i, b_i), $i < k$, and $a = a_k$, $b = b_k$ for $k = 1, \ldots, m$. The routing of a single path takes $O(n \log n)$ time with the Dijkstra algorithm in our settings, because the number of edges in $\widetilde{G}$ is $O(n)$. All steps take $O(|E| n \log n)$ time.

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, which will be an improvement in some settings. We define an *additional cost* of a set of paths by analogy with the *additional cost* of a path.

Problem 4 (Multiple Path Routing). Given the graph $\widetilde{G}$ with some paths already routed, find paths for $(a^*, b_1), \ldots, (a^*, b_k)$ so that *additional cost* is minimized.

We can 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 $\{b_1, \ldots, b_k\}$. We need to solve our problem for the state $(a^*, \{b_1, \ldots, b_k\})$. 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 the minimal *additional cost* of a set of paths $\{(v, b), b \in P\}$. A set of paths giving the minimal $f(v, P)$ 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' be the graph generated by Π, and note that it is connected. Let T be a shortest path tree of G', 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. The *additional cost* of Π' is at most that of Π. Indeed, the increment in ink is no greater because T is a subgraph of G'. Each path of Π' is shortest in G' and thus no longer than the corresponding path of Π. Hence, Π' is an optimal set for (v, P). □
Lemma 1 leads us to the following formula.

$$f(v,P) = \min \begin{cases} f(u,P) + \alpha\,\delta(uv) + |P|\,\beta\,\ell_{uv}, & \text{for } u \in W \text{ adjacent to } v, \\ f(v,P') + f(v,P-P'), & \text{for } P' \text{ with } \emptyset \subset P' \subset P \end{cases}$$

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'), where P' is 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 $\alpha\,\delta(e)+|P|\,\beta\,\ell_e$ in H. We add a new node h to H and connect it with all nodes of $\widetilde{G}$. For every new edge hu we assign weight $\min_{P'} f(u, P') + f(u, P - P')$, where P' varies over proper non-empty subsets of P. 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 with the Dijkstra algorithm.

To solve Problem 4 we work bottom-top. We first compute all $f(v, P)$ with $|P| = 1$ and v is a node of $\widetilde{G}$, by the algorithm for Problem 3, where we find a path with the minimal *additional cost*. Then we compute the values $f(v, P)$ for each v and $|P| = 2, \ldots, k$ by creating the corresponding graphs H. Finally, the answer for the problem is $f(a^*, \{b_1, \ldots, b_k\})$.

Running time. The main steps of the algorithm are the construction of graph H and finding a shortest path on it with the Dijkstra 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 $O(2^{|P|}n)$ time. We execute the Dijkstra algorithm only once per P starting from h to compute $f(v, P)$ for all $v \in W$. Thus, finding $f(v, P)$ for a known H and for all $v \in W$ takes $O(n \log n)$ time. Summing over all possible sets P produces

$$O\Big(\sum_{P}(2^{|P|}n + n\log n)\Big) = O(3^k n + 2^k n \log n).$$

To utilize the method solving Problem 4, we organize the paths into a sequence of subsets of paths having a common end. We route the paths of the first element of the sequence with the minimal *additional cost*, solving Problem 4. Then, using this routing we solve Problem 4 for the second subset, using an updated *additional cost* function, and so on. To avoid a long running time we need to keep the path subsets small. We experimented with $k = 5, 10$, and the results are shown in Section 4.

In practice, we set *routing cost* $= \alpha\, ink + \beta \sum_{uv \in E} \frac{\ell_{uv}}{d_{uv}}$, where d_{uv} is the Euclidean distance between the nodes u and v. This way we penalize the relative growth of path lengths to avoid long paths for short edges.

3.2 Local Adjustments and Spline Routing

To save space we omit some details in this section. Routing the paths through $\widetilde{G}$ defines the bundles. In the final drawing we would like to draw the paths of a bundle in a

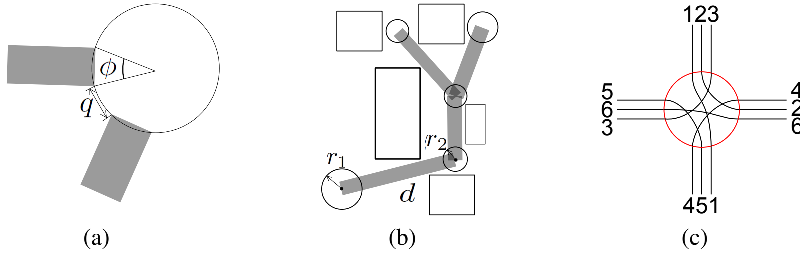

Fig. 4. (a) Desired radius (b) Allowed radius (c) Paths intersecting at a node

particular order, as will be explained in Section 3.3, while keeping them at a predefined distance from each other, and outside the obstacles. For this we need to have some free space around the edges of $\widetilde{G}$. To provide the free space, we surround each node of $\widetilde{G}$ by a circle, called a **hub**, with the center at the node position, and each edge by a rectangle, that are disjoint from the obstacles (Fig. 4(b)). In the final drawing every path is represented as a sequence of line segments and cubic Bezier segments, where each line segment is contained in a rectangle, and each cubic Bezier segment is contained in a hub (Fig. 4(c)). Such a path does not intersect the obstacles. To draw a Bezier segment inside of a hub, we place each control point of the segment inside of the hub; since a circle is convex, and a Bezier segment is contained in the convex hull of its control points, this keeps the segment inside the hub.

Hub Radii Calculation The radius of a hub is defined as the minimum of two radii: a desired one, and an allowed one. The calculation of the desired radius can be explained with Fig. 4(a). We would like a hub to be large enough to accommodate an incoming bundle by keeping angle ϕ at most $\pi/4$. We also would like to keep two bundles separated before entering a hub by having distance q at least the given edge separation. The allowed radius is explained with a help of Fig. 4(b). To keep two connected hubs separated, we require (a) that $r_1 + r_2 \leq \gamma d$, for some $0 < \gamma < 1$, and (b) that each hub does not intersect the obstacles.

Local Optimization To be able to route thick bundles without overlapping the obstacles, we first apply a heuristic preprocessing step before the local optimizations, in which each node of $\widetilde{G}$ participating in a path and belonging to an obstacle is moved away from the obstacles. The *routing cost* usually becomes larger after this step, but we obtain the necessary space around the paths. In order to minimize *routing cost* locally, we next iteratively adjust the position of each node of $\widetilde{G}$ by moving it in a random direction and trying to diminish *routing cost*. We also try to glue some of the nodes of $\widetilde{G}$ together, if it is beneficial. During these transformations, we pay attention to preserve conditions (a) and (b) mentioned above. Let m be the number of the obstacles, and c be the time required to find out if a circle or a rectangle intersects an obstacle. Using an R-tree [7] on the obstacles, one can find out if a circle or a rectangle intersects the obstacles in $O(c \log m)$ time. The number of edges in $\widetilde{G}$ is $O(n)$, therefore, a pass locally optimizing the position of every node of $\widetilde{G}$ can be done in $O(cn \log m)$ time. The Local Optimization is the most expensive stage of the algorithm, since we proceed iterating as long as we diminish *routing cost*, and we do not have a good upper bound for this step.

3.3 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 Fig. 4(c). The order of paths in bundles affects crossings of paths. Let P be the set of paths in $\widetilde{G}$ computed by path routing. We address the following problem.

Problem 5. Given the graph $\widetilde{G}$ and a set of paths P, find an ordering of paths for each edge of $\widetilde{G}$ that minimizes the number of crossings.

In our setting, the paths terminate at the nodes of $\widetilde{G}$ corresponding to the centers of the obstacles, and these nodes cannot be intermediate points of paths. Thus we have:

Path Terminal Property: No node is both an endpoint of some path and an intermediate point of some path.

We call nodes that are endpoints of paths **terminal** nodes. Using an argument similar to the proof of Lemma 1 we can assume that the paths P are simple and satisfy the following property.

Path Intersection Property. The common nodes and edges of any two paths form a path (which may be empty, or a single node).

Ideally, every two paths either do not cross or cross one time if needed (Fig. 5). An ordering of paths is **consistent** if any two paths cross at most one node. Clearly, if two paths cross once in some ordering, they must cross in every ordering. Hence consistent orderings have the minimum number of crossings. However, consistent orders are clearly not necessarily unique, and the choice of a particular one may greatly influence the quality of final drawing. The following property might appear desirable. A consistent order of paths is **nice** if, for any two paths, their order along all their common edges is the same (i.e. they may cross only at endpoint of their common subpath). Unfortunately, we found an example of $(\widetilde{G}, P)$ having no nice consistent order. On the other hand, we prove that a consistent ordering always exists, and moreover we provide an efficient algorithm to construct one. We consider the following problem.

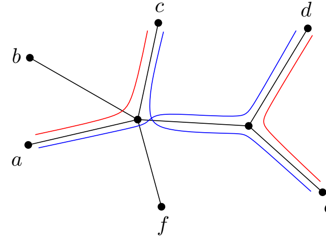

Fig. 5. Paths ac and de do not cross, while paths ad and ce cross once.

Problem 6 (Path Ordering). Given the graph $\widetilde{G}$ and a set of simple paths P satisfying the path terminal and intersection properties, compute a consistent ordering of paths for all edges of $\widetilde{G}$.

Algorithm. *Phase 1.* A basic step of our algorithm is the deletion of a node of $\widetilde{G}$ (Fig. **??**). For every non-terminal node v do the following. Let P_v be the set of paths passing through v in set P_v. Number the edges incident to v as $e_1, e_2, \ldots, e_t$ in clockwise order, and let $v_1, \ldots, v_t$ be the corresponding nodes adjacent to v. For every path $\pi \in P_v$ using edges e_a and e_b, represent it by pair (a, b). For each pair (a, b), add a new edge (v_a, v_b). Assign the paths labeled by (a, b) to this edge. The new edges incident to v_a should be inserted into $v_a's$ clockwise order in the position previously occupied by e_a, in the order determined by the positions of v_b. Finally, delete node v from the graph and the paths.

Phase 2. After all non-terminal nodes have been deleted, we reverse the process and undo the deletions, adding orders to the edges. Consider the deletion of v. The new

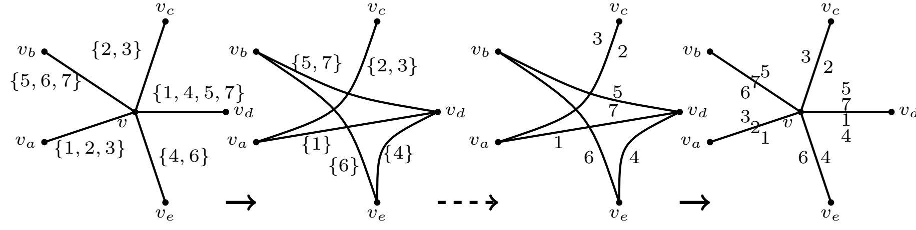

Fig. 6. Removal of node v; intervening steps; re-insertion of v. Paths of P_v are numbered $1, \ldots, 7$, and are shown in braces when unordered, and then with placement indicating their orders.

order of paths along edge e_a is obtained by concatenating the orders of paths along the edges $\{(v_a, v_b) : b \neq a\}$.

Implementation. First, create a new graph H. Initially, H is the subgraph of $\widetilde{G}$ generated by all the paths of P. Every path in P is stored as a list of nodes in H. For every edge e, L_e is a list of paths containing e. We assume that, for every node v of H, the list of edges incident to v are given in the clockwise order. Note that these lists are dynamic since H undergoes deletions of nodes. We keep track of the deletions in a forest F. Initially, F contains 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 children of the node corresponding to e. For instance, $L_{d,v}$ contains paths 1,4,5, and 7 in Fig. **??**. When node v is processed, list $L_{d,v}$ is split into new sublists $L_{d,e}$, $L_{d,a}$, and $L_{d,b}$. The (clockwise) order of sublists is important. Then we replace edge dv by edges de, da, and db in graph H and in order of edges around d. The first phase finishes, when all non-terminals are deleted from H. In the second phase, we process each tree in F in bottom-up order. The list of a node is simply the concatenation of the lists of its children. The leaves of F correspond (one-to-one) to the paths of P.

Theorem 1. *Given the graph $\widetilde{G} = (W, U)$, a set of simple paths P in $\widetilde{G}$ satisfying the path terminal and intersection properties, and a clockwise order of the edges around each node, an ordering of paths along edges of $\widetilde{G}$ minimizing the number of crossings can be computed in $O(|W| + |U| + L)$ time, where L is the total length of paths in P.*

Proof. Correctness. We need to show that the edge (v_a, v_b) added in Phase 1 is new. Indeed, if edge (v_a, v_b) already existed, there would be a path passing from v_a to v_b and a path passing v_a, v, v_b, which contradicts the path intersection property.

The ordering of paths computed by our algorithm is consistent since the split of paths P_v makes only necessary crossings. Two paths π_1 and π_2 will produce a crossing only when the last node of their common subpath is deleted and the clockwise order of the nodes around v is $\cdots v_a \cdots v_b \cdots v_{a'} \cdots v_{b'} \cdots$, where $\pi_1 = \cdots v_a v v_{a'} \cdots$ and $\pi_2 = \cdots v_b v v_{b'} \cdots$.

Running time. The time for processing node v (the deletion of v) is $O(1+d_{v,H}+s_v)$, where $d_{v,H}$ is the degree of v in H at the current step and s_v is the number of paths passing through v. The theorem follows since $d_{v,H} \leq d_{v,\widetilde{G}} + s_v$. $\square$

Overall, the complexity of the Ordering step is $O(|E|n + n \log D)$, where $|E|$ is the number of edges of the original graph G (the number of paths), n is the number of nodes in $\widetilde{G}$, and D is the maximum degree of $\widetilde{G}$.

Graph	Visibility graph					Delaunay triangulation				
	$\|W\|$	$\|U\|$	$k = 1$	$k = 5$	$k = 10$	$\|W\|$	$\|U\|$	$k = 1$	$k = 5$	$k = 10$
tail	105	348	8.4	16.9	18.8	257	638	13.3	16.0	18.5
notepad	198	776	16.4	16.8	16.8	445	1122	16.9	17.0	17.0
airlines	1175	5297	61.3	62.4	63.3	2825	7076	61.3	62.1	62.7
jazz	955	4478	32.2	33.2	33.6	2297	5798	32.2	34.3	36.0
protein	7290	32585	16.2	16.2	17.3	17501	43676	15.5	16.6	17.1
power grid	24705	109779	1.0	1.5	–	59297	148280	0.1	0.1	–
Java	7690	32712	32.0	34.9	–	18461	46350	30.0	34.1	–
migrations	8575	41451	74.5	75.3	–	20585	51510	74.4	75.9	–

Table 1. The percentage of ink gain, $1-($*routing cost* of bundled graph$)/($*routing cost* of straight edges$)$, of the proposed algorithm. For the cells with "–" running time exceeds 10 hours.

Graph	$\|V\|$	$\|E\|$	source	Visibility	Routing	Radii	Optimizations	Ordering	Overall
tail	21	68	[13]	0.11	0.02	0.04	0.13	0.01	0.34
notepad	22	113	[13]	0.00	0.03	0.05	0.24	0.01	0.35
airlines	235	1297	[1]	0.16	0.31	0.17	2.50	0.15	3.32
jazz	191	2732	[16]	0.14	0.42	0.23	3.02	0.18	4.04
protein	1458	1948	[16]	0.57	0.88	0.48	11.45	0.10	13.52
power grid	4941	6594	[16]	1.86	6.11	1.15	16.85	0.18	26.31
Java	1538	7817	GD'06 Contest	0.59	3.49	1.37	28.80	0.96	35.35
migrations	1715	6529	[1]	0.50	3.39	1.38	29.19	1.16	35.75

Table 2. Performance of ordered edge bundling algorithm (in seconds).

4 Experimental Results

We implemented our algorithm in MSAGL tool [11]. Edge bundling was applied for several real-world graphs (see Tables 1 and 2 for quick statistics). Unless node coordinates are available, we used the tool to position the nodes. All our experiments were run on a 3.1 GHz quad-core machine with 4 GB of RAM.

The quality analysis of ink minimization heuristics is given in Table 1. We compare the ink gain of algorithms with two methods of routing graph construction. An iterative approach with routing paths one by one corresponds to a $k = 1$ case. The results of routing multiple paths at a time are shown for groups of size 5 and 10.

The variant of the algorithm that routes multiple paths with the same endpoint simultaneously produces routings with smaller *routing cost*, while its running time is much longer (e.g. 4 seconds with $k = 1$, 2 minutes with $k = 5$, and 1.5 hours with $k = 10$ for airlines graph). The approaches with different routing graphs are quite similar in both *routing cost* minimization, and running time. Moreover, we could not identify significant differences in the quality of final drawings. We believe it is a result of Local Optimization step of our algorithm in which edge routes are shortened and smoothed. Overall, we chose $k = 1$ with sparse visibility graph as a default settings for our routing.

Table 2 shows the CPU times of algorithm steps. As can be seen, ordered bundles can be constructed for graphs with several thousand of nodes and edges in less than a minute. The most expensive steps are Local Optimization and Edge Routing.

We now demonstrate ordered edge bundling algorithm on real-world examples. A migration graph used for comparison of edge bundling algorithms is shown in Fig. 7.

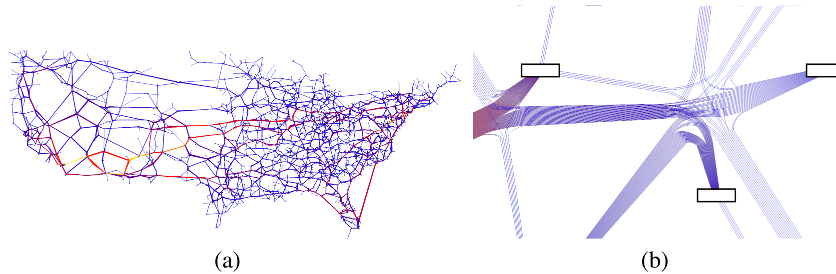

Fig. 7. Migration graph. (a) Overview. (b) Detail.

In our opinion, on a global scale ordered edge bundles are aesthetically as pleasant as other drawings of the graph (see e.g. [1, 4, 9, 10]). On a local scale, our result outperforms previous approaches by arranging edge intersections. A smaller example of edge bundling is given in Fig. 8. It shows another advantage of our routing scheme. Multiple edges are visualized separately making them easier to discover (compare the edge between nodes DrawingEditor and DrawApplication on original and bundled drawings).

5 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.

In our opinion, the novel cost function can be considered as a quality measure for different bundling heuristics. In a future, it would be interesting to verify if layouts with smaller *routing cost* correspond to subjectively better images. We are also exploring a possible extensions of the function to control the curvature of the resulting edges.

An important contribution of the paper is an efficient algorithm that finds an order of edges inside of bundles with minimal number of crossings. As mentioned above, this order is not unique. We left the question of choosing the best order as future research.

Our method splits the overlapped edge segments. The resulting edges has some thickness, so the nudging step can be treated as a heuristic for drawing fat edges. The main limitation of our technique is that routing and nudging steps are performed independently. A minimal *routing cost* might correspond to a routing, where edges can not be drawn with ideal thickness. In contrast, nudging step moves bundles, thus, increasing ink and edge lengths. We plan next to combine these two steps.

Another possible direction for future work concerns dynamic issues of edge bundling algorithm. First, a user may want to interactively change a bundled graph or change node positions. In that case, a system should not completely rebuild a drawing, but recalculate affected parts only. Second, a small deviation of algorithm parameters (e.g. ink importance α) may theoretically involve a full reconstruction of the routing, while a smooth transformation is preferable.

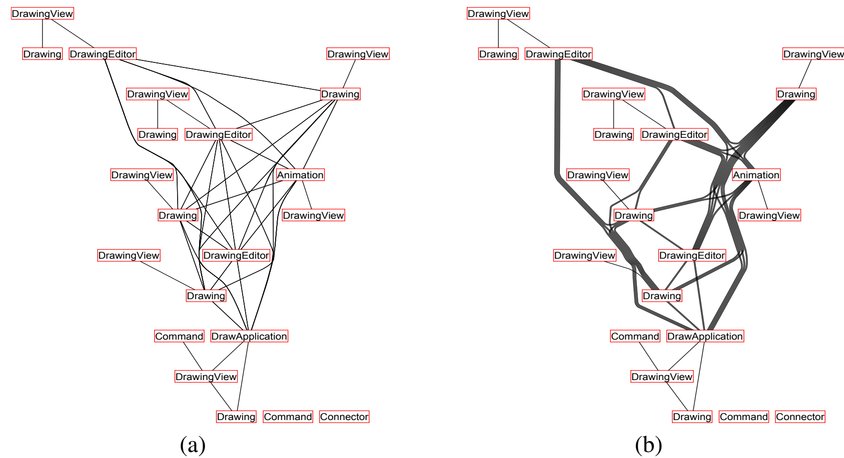

Fig. 8. Tail graph. (a) Original. (b) Bundled.

References

1. W. Cui, H. Zhou, H. Qu, P. C. Wong, and X. Li. Geometry-based edge clustering for graph visualization. *IEEE Trans. on Visualization and Computer Graphics*, 14(6):1277–1284, 2008.
2. C. A. Duncan, A. Efrat, S. G. Kobourov, and C. Wenk. Drawing with fat edges. In *Proc. 9th Int. Symp. on Graph Drawing*, pages 162–177, 2002.
3. T. Dwyer and L. Nachmanson. Fast edge-routing for large graphs. In *Proc. 17th Int. Symp. on Graph Drawing*, pages 147–158, 2009.
4. E. Gansner, Y. Hu, S. North, and C. Scheidegger. Multilevel agglomerative edge bundling for visualizing large graphs. In *Proc. IEEE Pacific Visualization Symposium*, 2011. to appear.
5. E. R. Gansner and Y. Koren. Improved circular layouts. In *Proc. 14th Int. Symp. on Graph Drawing*, pages 386–398, 2006.
6. 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.
7. A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. Int. Conf. on Management of Data*, pages 47–57, 1984.
8. D. Holten. Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):741–748, 2006.
9. D. Holten and J. J. van Wijk. Force-directed edge bundling for graph visualization. *Computer Graphics Forum*, 28(3):983–990, 2009.
10. A. Lambert, R. Bourqui, and D. Auber. Winding Roads: Routing edges into bundles. *Computer Graphics Forum*, 29(3):853–862, 2010.
11. L. Nachmanson, G. Robertson, and B. Lee. Drawing graphs with GLEE. In *Proc. 15th Int. Symp. on Graph Drawing*, pages 389–394, 2007.
12. M. Nöllenburg. An improved algorithm for the metro-line crossing minimization problem. In *Proc. 17th Int. Symp. on Graph Drawing*, pages 381–392, 2009.
13. S. Pupyrev, L. Nachmanson, and M. Kaufmann. Improving layered graph layouts with edge bundling. In *Proc. 18th Int. Symp. on Graph Drawing*, pages 465–479, 2010.
14. A. Telea and O. Ersoy. Image-based edge bundles: Simplified visualization of large graphs. *Computer Graphics Forum*, 29(3):843–852, 2010.
15. M. Wybrow, K. Marriott, and P. Stuckey. Orthogonal connector routing. In *Proc. 17th Int. Symp. on Graph Drawing*, pages 219–231, 2009.
16. Gephi dataset. `http://wiki.gephi.org/index.php?title=Datasets`.

A Non-existence of Nice Order of Paths

We show an example of $\widetilde{G}$ and P such that *any* consistent ordering of P is not nice. Consider a gadget with four paths shown in Fig. 9. Any consistent ordering satisfies the property that the first (left) blue path must be above the black path or the second (right) blue path must be below the black path. If the blue paths are drawn below (the first path) and above (the second one) the black path, then red-black do not cross properly.

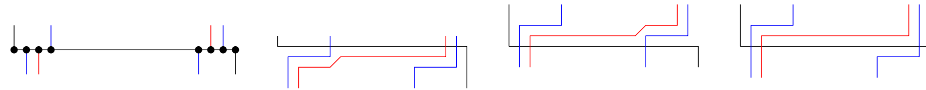

Fig. 9. The gadget and 3 ways to draw it.

Consider graph $\widetilde{G}$ with 7 gadgets in Fig. 10, where only black paths are shown. The gadgets are independent except (a) paths 1 and 4 share the blue path, and (b) paths 1 and 7 share the blue path. Either the first blue path for path 1 is above it or the second one is below it. In the first case, the blue paths of gadget 4 do not satisfy the above property, see Fig. 11 (a). Contradiction. In the second case, the blue paths of gadget 7 do not satisfy the above property, see Fig. 11 (b). Contradiction.

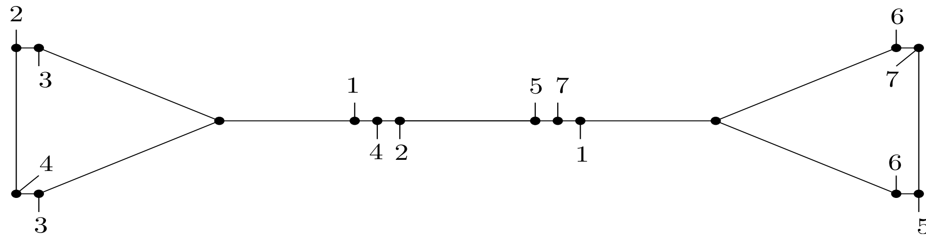

Fig. 10. Graph $\widetilde{G}$.

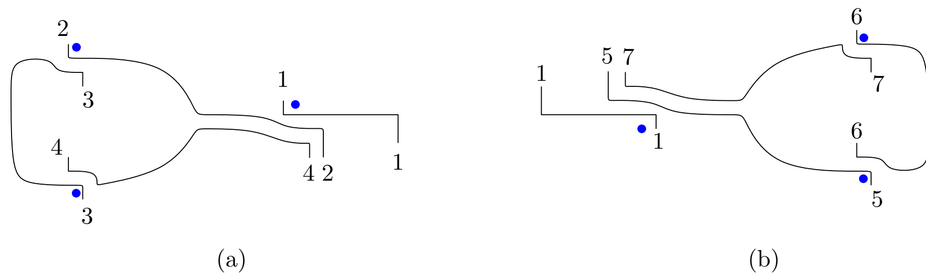

Fig. 11. Two cases.

B Further Examples

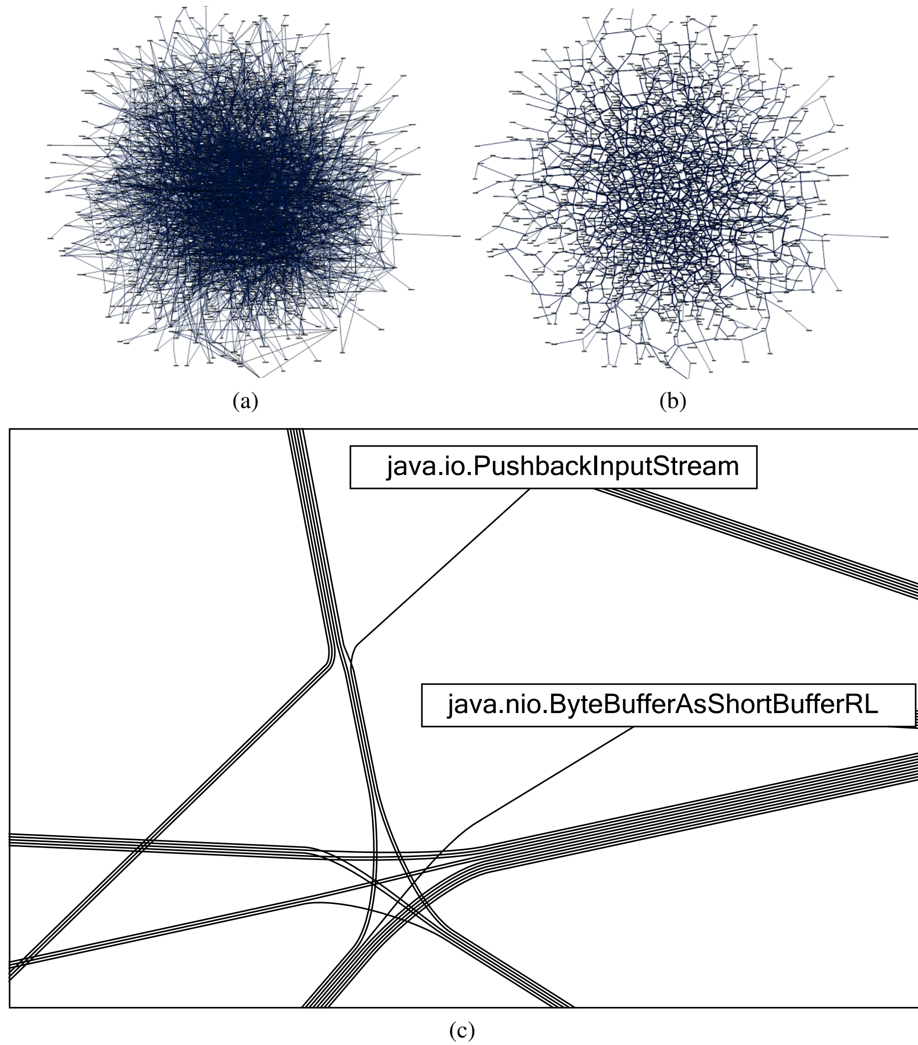

Fig. 12. Java graph. (a) Original. (b) Bundled. (c) Detail.