Multilevel Agglomerative Edge Bundling for Visualizing Large Graphs

Emden R. Gansner* Yifan Hu* Stephen North* Carlos Scheidegger*

AT&T Labs - Research, 180 Park Ave, Florham Park, NJ 07932.

ABSTRACT

Graphs are often used to encapsulate relationships between objects. Node-link diagrams, commonly used to visualize graphs, suffer from visual clutter on large graphs. Edge bundling is an effective technique for alleviating clutter and revealing high-level edge patterns. Previous methods for general graph layouts either require a control mesh to guide the bundling process, which can introduce high variation in curvature along the bundles, or all-to-all force and compatibility calculations, which is not scalable. We propose a multilevel agglomerative edge bundling method based on a principled approach of minimizing ink needed to represent edges, with additional constraints on the curvature of the resulting splines. The proposed method is much faster than previous ones, able to bundle hundreds of thousands of edges in seconds, and one million edges in a few minutes.[1]

Keywords: Edge bundling, multilevel, clustering, graph drawing.

Index Terms: I.3.3 [Computer Graphics]: Picture/Image Generation—Line and Curve Generation; I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Hierarchy and Geometric Transformations

1 INTRODUCTION

Graphs are often used to model relationships between objects arising in many application areas, such as social networks, biology, computer science and transportation. Node-link diagrams, in which nodes are drawn as points and edges as straight lines, are commonly employed to visualize graphs. These visualizations are mostly determined by the position we choose to associate with each vertex of the graph. This mapping of vertices to position is the *layout* of a graph. It is a truism that our ability to generate large data exceeds our ability to visualize it, and this gap motivates the work we present here. We are driven in part by the popularity of online social networks, as well as automated data acquisition techniques in the sciences and social sciences. Scalable and high-quality algorithms exist to lay out very large graphs; these include multilevel force-directed algorithms [6, 11, 15] and scalable multidimensional scaling algorithms [3]. On many graphs, the node-link diagrams produced by these algorithms are often aesthetically pleasing and reveal intrinsic structure of the graphs. However, on certain classes of graphs, a node-link diagram using straight edges is almost always difficult to comprehend, with edges obstructing nodes and each other. Often, the layout algorithms themselves are working correctly, in the sense that removing edges from the visualization can reveal meaningful node clusters. It is the sheer number of edges and their heterogeneous arrangement that clutter the visualization, hiding potential high-level patterns.

A number of approaches help reduce visual clutter. First, graphs can be simplified by coalescing clusters of nodes in a hierarchical fashion [25]. This can be combined with fisheye-style distortion to balance high-level structures and local details [8]. Second, interactive visualization systems [19] can be used to explore the graph, filtering out unnecessary visual artifacts.

A third useful tool is edge bundling. In this approach, edges are represented by deformable curves, typically cubic splines. Edges which are in some sense close to each other are referred to as *compatible*, and these compatible edges are then combined in a single bundle, sharing part of their routes. This is analogous to electrical wires fanning into a bundle, and fanning out at the other end. Bundling reduces visual clutter, and helps reveal high-level node and edge patterns. Initially, edge bundling was proposed for circular and hierarchical layout [7, 12, 21, 22], and was later extended to general graph layouts [5, 13, 18].

In this paper, we consider edge bundling for general undirected layouts. We propose a principled, efficient, and conceptually simple edge bundling algorithm. The algorithm is similar to recent fast agglomerative clustering techniques [1, 23], except here we cluster compatible edges to save ink, as we will shortly explain. We were also inspired by agglomerative bundling algorithms for circular layouts [7]. The major advance is that our algorithm works for general layouts.

The guiding principle of our algorithm is saving ink [7]. Drawing a bundled group of edges should take less ink than drawing each edge separately. Conceptually, the algorithm mimics the behavior of a human faced with the task of bundling a mass of electrical wires: identifying wires that have similar start and end points; merging them; checking whether additional wires can join existing bundles or need to start new bundles; and repeating this process.

To avoid the cost of computing all-to-all edge interactions, as in the force-directed edge bundling algorithm FDEB [13], we first construct a proximity graph of edges. Guided by this graph, we then check whether bundling each edge with its neighbors saves ink. When all possible bundlings are identified, we form a coarse edge proximity graph based on the edge grouping, and repeat the bundling process on this graph. After no more ink saving is possible, we fix the fan-in and fan-out parts of the edges (as shown in Figure 1), and consider the bundled parts of the edges to see if the aforementioned edge bundling procedure can be applied to the bundled parts recursively. Curvature is controlled by restricting the turning angles when an edge joins a bundle. Applying this algorithm to many real-world graphs results in fast edge bundling that reveals high-level edge patterns.

The remainder of this paper is organized as follows. In Section 2, we review related work. Section 3 introduces the multilevel agglomerative edge bundling algorithm, followed by Section 4, where we introduce GPU-based rendering methods for faster interactive manipulation. The section provides example visualizations, including some for very large graphs. We conclude the paper in Section 5 with topics for further research.

2 RELATED WORK

Early work on edge bundling focused on special classes of graph layouts. Newbery [22] proposed a method for handling layered layouts of directed graphs. The method identifies edges that form a complete bipartite graph, and makes these edges pass through a common dummy node, thus eliminating many crossings. A modi-

*e-mail: {erg,yifanhu,north,cscheid}@research.att.com

[1]Additional images can be found at `http://www2.research.att.com/~yifanhu/edge_bundling/`.

fication of this was implemented in `Graphviz` [9]. Holten's Hierarchical Edge Bundling (HEB) [12] works with graphs that have a defined hierarchy. It bundles edges using B-splines, following the control points defined by the hierarchy. Gansner et al. [7] presented an algorithm that reduces clutter in circular layouts by merging edges so that the resulting splines share some control points. In our work, we adopt their model of minimizing the total amount of ink needed to draw the edges. This objective is incorporated in our algorithm. Finally, Nachmanson et al. [21] consider edge bundling in layered drawings with edges already routed as polylines or splines. The goal of uncluttering the drawing is balanced with preserving the topology of the original drawing and disambiguating edges.

Cui et al. [5] proposed one of the first methods suitable for general undirected layouts. In the Geometry Based Edge Bundling (GBEB) method, a control mesh guides the edge-clustering process; edge bundles are formed by forcing all edges to pass through the same control points on the mesh. The algorithm was reported to be fast, although the resulting visualization was observed to exhibit a "webbing" effect [13], with edges having high curvature variations.

Holten and van Wijk proposed a Force-Directed Edge Bundling (FDEB) algorithm [13]. The algorithm is conceptually simple, utilizing edges modeled as flexible springs that can attract each other. The attractive force is proportional to the inverse (or inverse square) distance of the springs, as well as to the compatibility of the edges. It was found to result in smoother bundles that are easy to read. A weakness of this algorithm is its high computational complexity. The authors suggested that a Barnes-Hut like subdivision-based approach may be used to speed up the algorithm, though no details on how this can be done were given.

Lambert et al. [18] proposed an edge bundling algorithm that is also based on the use of a mesh. Graph edges are routed along mesh edges using a shortest path algorithm. Mesh edge weights are updated to encourage more graph edges to share common mesh edges. The algorithm was found to be faster than force-directed edge bundling [13]. Further optimization in the use of shortest path algorithm and in parallelization made its speed close to that of the geometry-based bundling algorithm [5]. The method was subsequently extended to 3D [17].

3 Multilevel Agglomerative Edge Bundling

Throughout this paper we assume that we are working with a graph $G = \{V, E\}$, with $|V|$ vertices and $|E|$ edges. We assume that we are making 2D drawings, and that the positions of vertices are given. The proposed algorithm is readily extended to 3D.

As discussed in Section 1, our intuition is to mimic what a human operator would do when faced with the task of bundling a mass of electrical wires: identify wires that have similar start and end points; merge them; check if additional wires can join existing bundles, or need to start their own; and repeat this process. To achieve this, we must first identify edges that are "similar." Holten and van Wijk [13] introduced four edge compatibility measures. For each edge, a naive way to find similar edges is to check every other edge for compatibility, an $|E|^2$ operation.

However, we are interested in an edge bundling algorithm that can scale to very large graphs, so we must avoid the quadratic complexity involved in making such an all-to-all similarity computation. Our solution is to use a simple compatibility measure where each edge is treated as a point in 4-dimensional space, and edge-edge similarity is given by a metric in that space. Within this setting, we can form an edge proximity graph efficiently. Edges that are neighbors in the proximity graph can then be checked for possible bundling. Edges that are not immediate neighbors can still be bundling in the multilevel process described in Section 3.3. The overall algorithm is illustrated in Figure 1, and described in detail in the following.

3.1 Edge proximity graph

Each vertex u has a position in 2D denoted as x_u. We represent each edge (u,v) as a 4-dimensional vector (x_u,x_v). Two edges are "close" if their Euclidean distance in the 4-dimensional space is small. A space decomposition, e.g., a *kd*-tree, can be constructed using all $|E|$ 4-dimensional vectors in time $|E|log(|E|)$. This data structure then allows us to find the k-nearest ($k << |E|$) neighbors in time $k\ log(|E|)$ per edge. Thus we construct an edge proximity graph Γ in time $k|E|log(|E|)$. We note that ordering of the points in the 4-dimensional vector (x_u,x_v) affects the distance in the embedding. This effect can be avoided by ordering the 4-dimensional vector based on the $x-$ or $y-$ coordinates, or simply by inserting both (x_u,x_v) and (x_v,x_u) in the data structure, without affecting the $k|E|log(|E|)$ complexity. We stress here that vertices in Γ and G are not the same: vertices of Γ are edges of G.

An edge proximity graph Γ so constructed may not treat all compatible edges as neighbors, both because of the limited number k of neighbors considered, and because we measure proximity using the Euclidean norm in 4D, instead of using more detailed compatibility measures [13] that take into account length, angle, position, and visibility. This, however, does not cause significant problems, because we ultimately measure the compatibility of edges by whether bundling results in saving ink. In addition, our multilevel bundling process can bundle not only neighbors, but a neighbor's neighbors, etc. Finally, a recursive process further increases the opportunity to bundle edges that were not initially considered as being close. Figure 1 (b) shows the edge proximity graph corresponding to the edges in Figure 1 (a).

3.2 Agglomerative bundling

Once Γ is constructed, it guides the bundling decisions. For each neighbor v of a vertex u in Γ, we calculate the ink saving that may result if the edge represented by v is bundled with the edge represented by u. We then choose among all u's neighbors one that gives the maximal ink saving. Note that a neighbor v may already be bundled with other edges, in which case v represents a bundle. Let $e(u)$ denote the edge or edges represented by a node u in Γ. Let $\text{ink}(e(u))$ denote the ink needed to draw the edge (or bundle of edges) represented by u, and $e(u) \cup e(v)$ an edge bundle formed by merging the edges represented by u and v. Then the amount of ink saved by bundling the edges is defined as

$$\text{ink}(e(u)) + \text{ink}(e(v)) - \text{ink}\,(e(u) \cup e(v))$$

We calculate the (approximate) minimal amount of ink needed to draw a set of edges using the one dimensional optimization procedure of Gansner and Koren [7]. Given the set of edges $e(u) \cup e(v) = \{e_1 = (x_1^S, x_1^T), e_2 = (x_2^S, x_2^T), \ldots, e_k = (x_k^S, x_k^T)\}$. Assume that the ends of the edges are properly ordered so that they form two equal sized sets, the source set $S = \{x_1^S, x_2^S, \ldots, x_k^S\}$ and the target set $T = \{x_1^T, x_2^T, \ldots, x_k^T\}$. The aim of the ink optimization procedure is to find two meeting points M_1 and M_2, such that edges leaving nodes in the source set S fan-in to M_1, travel along a straight line to M_2, then fan-out to nodes in the target set T (Figure 2). The total ink used to draw these edges is

$$f(S,T,M_1,M_2) = \sum_{x \in S} ||x - M_1|| + ||M_1 - M_2|| + \sum_{x \in T} ||M2 - x||. \quad (1)$$

The meeting points M_1 and M_2 are chosen to minimize the total ink,

$$\text{ink}\,(e(u) \cup e(v)) = \min_{M1,M2} f(S,T,M_1,M_2).$$

The following heuristic is used to find the approximate optimal meeting points. First, the centroids of the sets S and T are computed. Then, we minimize (1) by the procedure known as golden

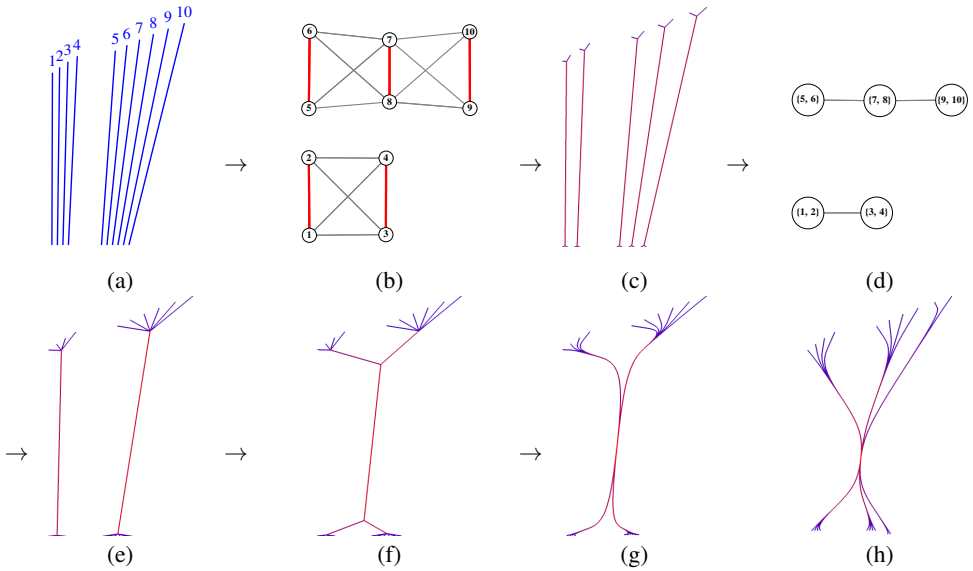

Figure 1: How multilevel agglomerative bundling works: (a) Original edges. Ink = 35.82. (b) Constructing an edge proximity graph ($k = 3$). (c) After one level of agglomerative bundling. Ink = 19.31. (d) Constructing a coarsened proximity graph. (e) After multilevel agglomerative bundling. Ink = 12.00. We call the tree-like branching portions of the bundles at their endpoints the *fan-in* and *fan-out*. (f) After recursive application of bundling. Ink = 10.94. (g) Rendered using splines. (h) With turning angle $<= 40^o$ and spline rendering. Ink = 15.24.

section search [24], finding two points M_1 and M_2 along the line linking the centroids. Figure 2 illustrates the process.

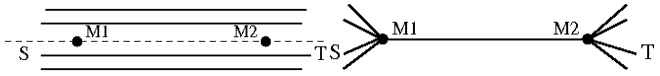

Figure 2: Illustration of the ink minimization process. The dotted line pass through the centroids of S and T.

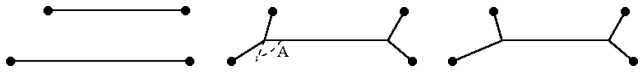

Figure 3: Left: edges with far away end points. Middle: bundling results in a large turning angle. Right: limiting the turning angle.

When the end points of edges are far away, sometimes bundling these edges results in large turning angles (Figure 3, middle), and the corresponding splines will have a large curvature. We avoid this in two ways. First, we allow a user-specified maximum turning angle, and constrain the two meeting points so that this angle is not exceeded. Second, we compromise between saving ink and smaller turning angle by selecting the meeting points M_1 and M_2 to minimize

$$f(S,T,M_1,M_2)\left(1+\frac{\cos(A_{max})}{p}\right),$$

instead of (1), with parameter $p > 1$, subject to the maximum turning angle constraint. Here A_{max} is the maximum turning angle given by the specific meeting points. A larger value of p makes ink saving more important, while a smaller value avoids sharp turning angles.

3.3 Multilevel agglomerative bundling

After edges are bundled by the agglomerative bundling process, we coarsen the edge proximity graph by coalescing nodes (which represent edges of the original graph) that are bundled. Now each node of the coarsened graph may represent a bundle of edges (Figure 1 (d)). We then repeat the bundling process described in the previous section. This multilevel process terminates when no more ink saving can be identified. The multilevel process allows edges that may be far away in the original edge proximity graph to have a chance to form a bundle, provided that doing so results in saving ink.

Figures 1 (c) and 1 (e) show the results after one and two levels of agglomeration. The first level reduces ink from 35.82 to 19.31. The second level reduces it further to 12.00. No further ink reduction can be achieved by going one level further.

3.4 Recursion

After one step of this bundling process, each edge is represented by a polyline with at most three segments. Figure 4 (middle) shows the result at the end of the multilevel process when applied to the `airlines` graph (see Section 4). While a single step of the algorithm reduces clutter when compared with the original graph, there are still many similar splines that can be merged further. Therefore we take the bundled straight sections as a new set of edges, and recursively apply the multilevel process to bundle the edge bundles. To promote straight lines, weights are used to represent the number of edges a bundle represents, and the ink to draw a bundled line is proportional to its weight. This has the effect of avoiding excessive bending of heavier bundles when merging bundles together. Figure 1 (f) illustrates the result of recursion. This recursion ends when no ink saving can be achieved. At this point each edge is a polyline. We use the turning points as control points and render the edges as splines (Figure 1 (g)). Figure 4 (bottom) shows the result of recursively applying the multilevel process to the `airlines` graph.

We call this multilevel agglomerative edge bundling procedure *MINGLE*. Algorithm 1 gives the pseudo-code for MINGLE. In the algorithm, $e(v)$ denotes the set of edges represented by node v. If v has been assigned a group, then this set also include edges represented by those nodes in the same group as v.

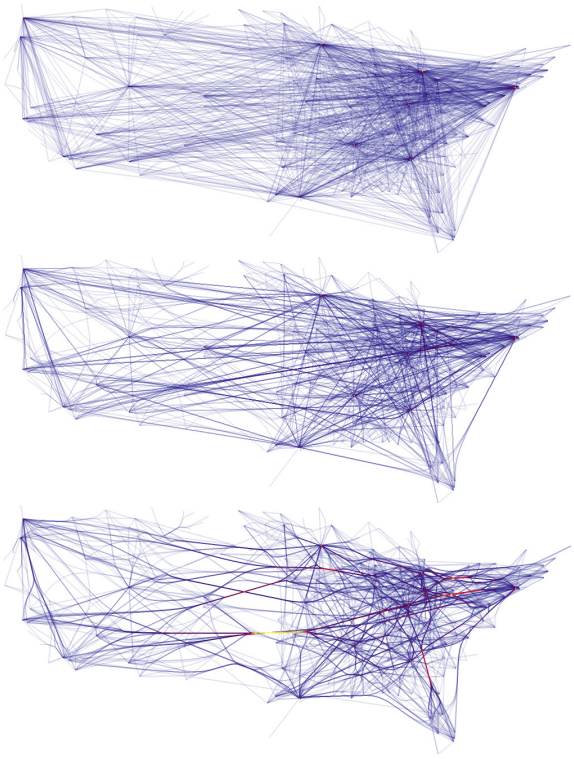

Figure 4: (a) Graph `airlines` (b) Bundled without recursion (c) With recursion.

Algorithm 1 Multilevel agglomerative edge bundling algorithm (MINGLE)

input: a set E of edges with position of the end points given.
set $totalgain = 0$; UNGROUPED $= -1$;
form an edge proximity graph $\Gamma = \{V_E, E_E\}$ of E;
repeat
  set $gain = 0$; $k = 0$;
  set $group(u) =$ UNGROUPED for all $u \in V_E$;
  for each node u of Γ **do**
    if $group(u) =$ UNGROUPED **then**
      find among all neighbors of u a node v that gives the most ink saving if $e(u)$ and $e(v)$ are bundled;
      $gain(u,v) = \text{ink}(e(u) \cup e(v)) - (\text{ink}(e(u)) + \text{ink}(e(v)))$;
      if $gain(u,v) > 0$ **then**
        bundle $e(u)$ and $e(v)$;
        $gain = gain + gain(u,v)$;
        if $group(v) \neq$ UNGROUPED **then**
          $group(u) = group(v)$;
        else
          $group(u) = k$; $group(v) = k$;
        end if
      else
        $group(u) = k$;
      end if
      $k = k + 1$;
    end if
  end for
  $\Gamma = \Gamma$ with nodes that are in the same group coalesced;
  $totalgain = totalgain + gain$;
until $gain \leq 0$
return $group$, $totalgain$.

The time complexity of the MINGLE algorithm can be analyzed as follows. At each level of the multilevel algorithm, every edge has to be checked against k of its neighbors to see if merging them saves ink. In the best case, when each edge only merges with a few other edges, this takes $O(k|E|)$ time (assuming that finding the optimal meeting point of two edges can be done in constant time). Combined with the time needed to compute the proximity graphs, the best case complexity is $O(k|E|\log(|E|))$. The worst case occurs when each edge bundles with all previously processed edges, i.e., edge i merges with a bundle consisting of edges $1, 2, \ldots, i-1$. In this case, the total time will be proportional to $O(|E|^2)$. In practice, on real-world graphs, this pathological complexity has not been observed, and the agglomerative bundling process runs very efficiently, as explained further in the next section.

4 Results

We implemented MINGLE in C, and used OpenGL for rendering.

For the initial edge proximity graph, instead of constructing a strict proximity graph [16], such as a relative neighborhood graph or a Gabriel graph, for our purposes it is sufficient to find an approximate proximity graph, which can be constructed quickly. We generate a k-nearest neighbor graph using the ANN library (Version 1.1.2, [20]). In our experiments, unless otherwise specified, we fix k at 10. Further discussion of the choice of k is given below.

In the examples and experiments in this paper, we set the maximal turning angle to 40°, except for Figure 7 (right), where we wanted to be aggressive and did not set a limit on turning angle. Unless node positions are given, all graphs are first drawn using `sfdp` from `Graphviz` [9], an implementation of a force-directed algorithm [15].

All CPU timings of the algorithm were measured on an Intel Xeon E5506 2.13GHz processor running Ubuntu 9.10 with 12 GB of RAM and a GeForce GTX260 graphics card. We used `gcc -O3` for compilation. Table 1 shows the CPU times given by MINGLE. For comparison, we also show the CPU times for FDEB [13] and GBEB [5] reported[2] by Holten et al. [13]. For these two algorithms, we only have CPU data for the two small graphs `airlines` and `migration`, which depict US airline routes and migration information, respectively. The timing data were from an Intel Core2 Duo 2.66 GHz processor. According to `cpubenchmark.net`, this class of processors is 3 to 4 times slower than the Intel Xeon processor we used.

We can see from the table that even factoring in processor speed, the MINGLE algorithm is substantially faster than FDEB and GBEB.

We also tested MINGLE on nine larger graphs. These graphs, except `ixpas`, were taken from the University of Florida Sparse Matrix Collection [26]. Further information concerning them can be found there. The graphs are chosen with two objectives.

Our first objective is to have a diverse sample of graphs. For example, the graph `yeast` comes from a biological application. The graph `ixpas` is bipartite with edges connecting autonomous system (AS) nodes and Internet exchange points (IXP) where they have a presence. AS nodes are assigned estimated (averaged) locations; IXP nodes have known, exact geographic locations. The graph `amazon0302` depicts co-purchase relations between items on amazon.com. The graph `amazon0302_b` uses the `amazon0302`

[2] We did run FDEB on our machine. It was able to bundle the first three graphs in Table 1. The CPU time we observed (15, 56 and 102 seconds, respectively) is less favorable than these reported in Holten et al. [13], when factor in the difference in processor speed. This could be due to many reasons, one of which could be that we only have an excutable program which may be optimized for a platform different from ours. Therefore we thought it would be fairer to quote the timing as reported in Holten et al. [13]. We have no access to GBEB.

name	$\|E\|$	MINGLE	FDEB*	GBEB*
airlines	1297	0.1	19	2.5
yeast	6646	0.9	-	-
migration	9660	1.0	80	18.8
wiki-Vote	100762	18.4	-	-
ixpas	149661	32.3	-	-
net50	464440	87.1	-	-
amazon0302	899792	277.	-	-
net100	1001640	204.	-	-
amazon0302_b	1233364	267.	-	-
net150	1538840	355.	-	-
Stanford	1992636	404.	-	-
pattern1	4652095	1049.	-	-

Table 1: CPU time (in seconds) taken by different edge bundling algorithms. Columns marked with an asterisk (*) show CPU times from a processor that is 3-4 times slower than the one used for MINGLE.

name	source	$\|V\|$	% ink saving	avg. work
airlines	[5, 13]	235	59.2	126.4
yeast	[26]	2361	45.1	206.6
migration*	[5, 13]	6517	74.5	137.8
wiki-Vote	[26]	8297	66.8	173.5
ixpas	AT&T	28546	90.5	230.4
net50	[26]	16320	81.5	164.3
amazon0302	[26]	262111	54.8	240.7
net100	[26]	29920	85.3	164.0
amazon0302_b	[26]	519010	55.8	207.9
net150	[26]	43520	83.9	192.5
Stanford	[26]	281903	69.7	196.1
pattern1	[26]	19242	84.5	176.2

Table 2: Additional measurements of MINGLE on test cases. "Ink saving" is percentage of ink saved when edges are bundled, instead of drawn as straight edges. "Work" is average number of ink optimizations each edge, or bundle, is involved in. *The migration graph contains two-way edges that represent two-way migrations, and the duplication amount to 21.9% of total ink. Subtracting this, the ink saving for the migration graph is 52.6.

matrix, but treats the rows and columns as two separate sets of vertices, thus yielding a bipartite graph.

The second objective is to have a wide range of number of edges so as to test the scalability of MINGLE. To that end we selected net50, net100, net150, three graphs of the same kind but different sizes.

As can be seen, MINGLE is able to bundle graphs with around 100,000 edges in about 20 seconds, and graphs with around one million edges in around 4 minutes. The largest graph, pattern1, has 4.6 million edges, and MINGLE processes it in 14 minutes. Overall, the algorithm scales well.

Table 2 gives some additional details about the graphs, and about MINGLE. It shows the percentage of ink saved, defined as,

$$\frac{\text{ink to draw straight edges} - \text{ink to draw bundled edges}}{\text{ink to draw straight edges}},$$

in percentage. It also shows "average work", a measure of the average number of times an edge, or a bundle, is involved in ink optimization. Every time the ink optimization routine is invoked to find optimal bundle for m edges or bundles, we increment "work" by m. The total "work" is divided by the number of edges in the original graph to get the "average work". We can see that this quantity remains relatively stable as number of edges increases (rows in both tables 1-2 are ordered by increasing number of edges). Given that the majority of CPU time is spent in the ink optimization procedure, the relatively stable "average work" shows that algorithm is not exhibiting the worst case quadratic time complexity.

While all above results are based on $k = 10$, we experimented with values of k between 2 to 1000. Table 3 shows the amount of ink saving and CPU time for six graphs of varying sizes. Clearly, a relatively small k is sufficient to achieve good ink saving. Surprisingly, increasing k beyond a small value actually reduces ink saving, and increases CPU time. We do not know the exact reason why ink saving actually deteriorate as k increases beyond 3. But we believe that a smaller k prevents bundling of multiple edges at an early stage of the multilevel agglomerative bundling process. This create a more balanced tree and promotes consideration of merging candidates that are not consider very close in the 4D space.

We choose a conservative $k = 10$ for results in Table 1-2 as we are aware that for artificial data, e.g., two sets of m parallel lines each, with a small separating distance between the sets, a smaller $k < m$ will yield a disconnected proximity graph. Although MINGLE will run fine on this example, the disconnected proximity graph does prevent edges between the two sets to have a chance to merge. But in practice, for the graphs tested in Table 3, $k = 3$ seems to be a good choice. Overall, this experiment confirms that a small k does not hurt performance because of the local nature of k-nearest neighbor graph, rather, the multilevel agglomerative process is robust and does provide the global reach needed.

k	2	3	5	10	50	100	1000
airl.	57.1	62.3	61.4	59.2	55.6	53.5	48.75
	0.1	0.05	0.08	0.14	0.5	0.86	3.9
migra.	72.3	77.1	75.7	74.5	72.5	71.6	68.1
	0.7	0.5	0.64	1.0	3.9	6.4	51.5
wiki.	60.5	68.8	67.5	66.8	64.7	63.5	63.5
	9.5	5.7	11.4	18.4	70.3	127	391
amaz.	48.5	57.4	56.0	54.8	-	-	-
	262	157	177	277	-	-	-
net15.	86.2	84.5	83.9	83.0	-	-	-
	85	112	355	1257	-	-	-
patt.	70.6	87.4	85.2	84.5	-	-	-
	516	407	520	1049	-	-	-

Table 3: Effect of k on the ink saving and CPU time. In each cell, the top number is the percentage of ink saving; the bottom number is CPU time (seconds). Cells marked with "-" mean that CPU time exceeds 1500 seconds. It is seen that a relatively small k (e.g., $k = 3$) is better than very large k, and gives a larger ink saving and smaller CPU time. (The names of graphs are abbreviated to save space.)

4.1 Rendering

Similar to Holten and van Wijk [13], we use a GPU-based, OpenGL rendering technique to highlight edge bundles. We use the standard technique of additive alpha blending with a floating-point framebuffer object to accurately count the number of edges incident on each pixel, typically called the *overdraw*, which we will model as a scalar field $\omega : D \subset R^2 \to R$ (D denotes the screen rectangle). To determine the color of each pixel, we first find the maximum overdraw $M = \max_{x \in D} \omega(x)$ using the standard procedure of *reduction* on the GPU [4]. We then use a linear colormap that encodes the overdraw as a function from $[0, M] \to RGB$. We use a blue-red-yellow-white palette for light backgrounds and a light-cyan-red-yellow-white palette for dark backgrounds.

Holten and van Wijk [13] suggest a continuous interpolation between straight edges and bundled edges as a way for user to understand the bundle structure. We adopt that approach. Specifically,

let a bundled edge be represented as a polyline $\{x_0, x_1, \ldots, x_k\}$. The projection of point x_i onto the straight line $\{x_0, x_k\}$ is

$$\bar{x_i} = x_0 + \frac{(x_i - x_0)^T (x_k - x_0)}{||x_k - x_0||^2}(x_k - x_0).$$

Each step of the animation uses control points $\{s\bar{x_i} + (1-s)x_i | i = 0, 1, \ldots, k\}$ to form splines, with parameter s varying from 0 (straight line) to 1 (bundled edge). To speed up rendering, we use a relatively recent feature in GPUs known as geometry shaders. The geometry shading step appears in the graphics pipeline between vertex and fragment processing. Its most important feature is to create entirely new graphics primitives (triangles, polylines, etc.) directly on the GPU. In our case, the advantage of such an approach is that the control points of the splines are only sent to the GPU once. The actual interpolation along the s parameter and the tessellation of the spline into line segments is performed by the geometry shader, greatly reducing the workload on the graphics bus in comparison with what it would be if we were to create this geometry on the GPU for every frame.

Interactively varying s gives a continuous deformation from the original graph to the bundled graph, making the edge structure much easier to understand. (See `http://www2.research.att.com/~yifanhu/edge_bundling/`.) Larger values for s, approaching 1, also give thick bundles, allowing one to see more clearly how many edges go into particular sections of a bundle.

In the case where relative bundle sizes are critically important, we provide an alternative rendering based on *hill shading* [14]. The added illumination cues help determine which bundles carry more edges, in particular where they split and merge. Our hillshading rendering procedure is straightforward. Starting with the raw overdraw count image, we first blur it by some amount (currently we use a separable Gaussian filter with 3σ decay at 5 pixels). Hill shading works by approximating diffuse illumination of a mountain range modeled as a height field. The illumination b at each pixel is given by $b = \max(0, -\langle l, \hat{n} \rangle)$, where l is the direction of the incoming light vector (assumed constant) and $\hat{n}$ is the normal vector at each point. The normal vector is given by

$$n = \left(-s\frac{\partial \omega}{\partial x}, -s\frac{\partial \omega}{\partial y}, 1.0 \right), \; \hat{n} = n/||n||.$$

The constant s is a parameter which essentially controls the slope of the hills. In traditional hill shading, the units of height are the same as the units which measure length in the field itself, which gives a single sensible choice of $s = 1$. However, when using hill shading in abstract settings such as ours, s can be an arbitrary positive value. The choice of s can influence the perception of the final rendering, and while we could leave s to be interactively determined, we believe it is important to provide good defaults. We have found in our experiments that picking the value of s which maximizes the entropy of the resulting normal distribution tends to yield aesthetically pleasing results. The idea of maximizing entropy of a distribution related to the viewing parameters is well-known [2].

4.2 Further Examples

Figure 5 (top) shows a layout of the Internet IXP peering graph `ixpas`, with 149661 edges. It is difficult to see what is going on in parts of the graph where the edge density is high. Using edge bundling, Figure 5 (bottom) shows the flow structure more clearly. For example, there is a strong connection between a site near Honolulu, Hawaii and sites in Europe and Asia. Also, there is a site in Nassau, Bahamas that connects with many sites in the US and Europe, a pattern difficult to discern in the original drawing.

Figure 6 shows the largest component of graph `net50`, comparing the original drawing and bundled drawing, with and without hill shading. With bundling, it is easier to see the small groups of vertices between the rightmost large group of vertices, and the long vertical group vertices. This is because there are fewer edges in the way to obstruct the view. Hill shading adds additional visual cues to determine which bundles carry more edges, and where they split and merge.

Finally, Figure 7 shows the graph `amazon0302` with close to a million edges. This directed graph was collected by crawling the Amazon.com website. Each node is an item, and an edge exists from item i to item j if, according to Amazon.com, customers who bought item i also frequently bought item j. Without edge bundling, the drawing is a hairball cluttered with edges. With bundling, there is a significant increase in discernible details. However this graph also illustrates a limitation of edge bundling: while more details can be seen with bundling, the result is still a hairball. We believe no edge bundling algorithm can reveal high-level edge patterns that do not exist in the layout.

5 Conclusions and Future Work

We proposed a multilevel agglomerative edge bundling method for general graph layouts. Our method is based on a principled approach of minimizing the ink needed to represent edges, with additional constraints on the curvature of the resulting splines. The method is significantly more efficient than previously published ones, able to bundle hundreds of thousands of edges in seconds, and one million edges in a few minutes. The algorithm is not difficult to implement, requiring no underlining meshes. The resulting edge-bundled graphs show significantly reduced cluttering, and can reveal some high-level edge patterns.

While we combined the edge proximity graph with an ink-saving based edge bundling technique, we could replace the ink-saving algorithm with, say, a force-directed edge bundling algorithm [13]. More specifically, in Algorithm 1, the gain in bundling edges could be calculated using a force-directed procedure. Alternatively, we could modify the force-directed edge bundling of Holten and van Wijk [13] by limiting interaction of edges only to those that are neighbors in the edge proximity graph. Either approach has the potential to speed up the force-directed algorithms. We are exploring these possibilities and initial results look promising.

We note that the one-dimensional ink saving algorithm we used is simplistic and assumes that all edges will merge at one of the two control points. In practice, it may be more aesthetic for edges to join at different points along the bundle. This is another reason why a force-directed algorithm could be an interesting alternative to the one-dimensional ink saving algorithm. The problem of finding the shortest length of lines linking a set of end points is related to finding a minimum Steiner tree, which is known to be NP-Hard [10].

Finally, we note that while edge bundling is a helpful tool in reducing clutter, it cannot find structures that are not present in the layout. How to steer a layout algorithm with a edge bundling tool to find hidden structures could be an interesting topic of research.

Acknowledgments

We would like to thank Danny Holten for providing the FDEB code, and the `airlines` and `migration` test data. We would also like to thank Arif Bilgin for helping with gathering data for FDEB.

References

[1] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008.

[2] U. Bordoloi and H.-W. Shen. View selection for volume rendering. In *Proceedings of IEEE Visualization*. IEEE, 2005.

[3] U. Brandes and C. Pich. An experimental study on distance based graph drawing. In *Proc. 16th Intl. Symp. Graph Drawing (GD '08)*, volume 5417 of *LNCS*, pages 218–229. Springer-Verlag, 2009.

[4] I. Buck and T. Purcell. *GPU Gems*, chapter 37. Addison-Wesley, 2005.

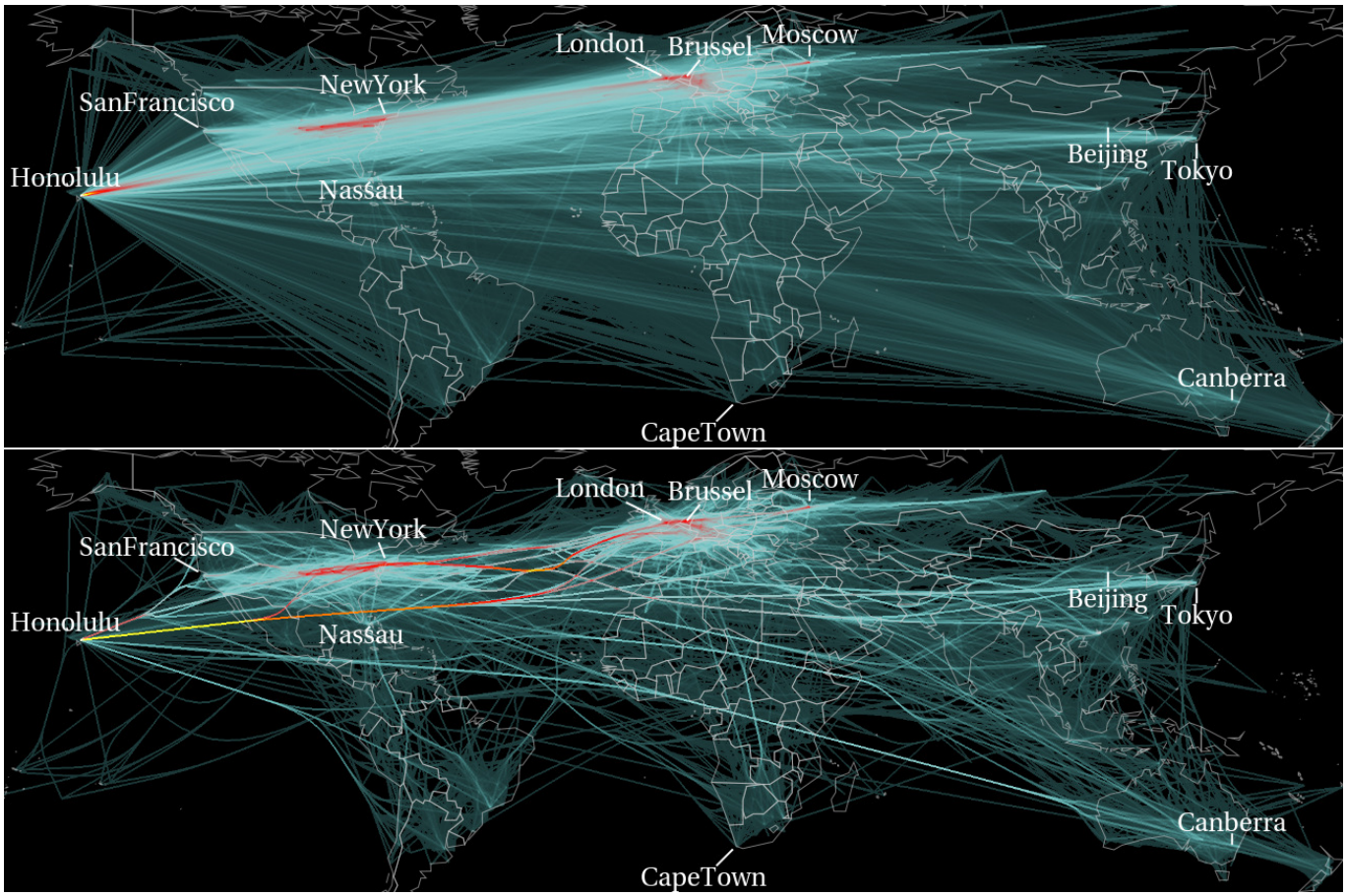

Figure 5: ixpas graph ($|E| = 149661$) before and after edge bundling

[5] 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*, 14:1277–1284, 2008.

[6] P. Gajer, M. T. Goodrich, and S. G. Kobourov. A fast multi-dimensional algorithm for drawing large graphs. *LNCS*, 1984:211 – 221, 2000.

[7] E. R. Gansner and Y. Koren. Improved circular layouts. In *Proc. 14th Intl. Symp. Graph Drawing (GD '06)*, LNCS, pages 386–398, 2006.

[8] E. R. Gansner, Y. Koren, and S. North. Topological fisheye views for visualizing large graphs. *IEEE Transactions on Visualization and Computer Graphics*, 11:457–468, 2005.

[9] E. R. Gansner and S. North. An open graph visualization system and its applications to software engineering. *Software - Practice & Experience*, 30:1203–1233, 2000.

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

[11] S. Hachul and M. Jünger. Drawing large graphs with a potential field based multilevel algorithm. In *Proc. 12th Intl. Symp. Graph Drawing (GD '04)*, volume 3383 of *LNCS*, pages 285–295. Springer, 2004.

[12] D. Holten. Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. *IEEE Transactions on Visualization and Computer Graphics*, 12:2006, 2006.

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

[14] B. K. P. Horn. Hill shading and the reflectance map. *Proc. of the IEEE*, 69(1), 1981.

[15] Y. F. Hu. Efficient and high quality force-directed graph drawing. *Mathematica Journal*, 10:37–71, 2005.

[16] J. W. Jaromczyk and G. T. Toussaint. Relative neighborhood graphs and their relatives. *Proc. IEEE*, 80:1502–1517, 1992.

[17] A. Lambert, R. Bourqui, and D. Auber. 3D edge bundling for geographical data visualization. In *IV '10: Proceedings of the 14 International Conference on Information Visualisation (IV'09)*, Washington, DC, USA, 2010. IEEE Computer Society.

[18] A. Lambert, R. Bourqui, and D. Auber. Winding Roads: Routing edges into bundles. *Computer Graphics Forum*, 29:853–862, 2010.

[19] T. Moscovich, F. Chevalier, N. Henry, E. Pietriga, and J.-D. Fekete. Topology-aware navigation in large networks. In *CHI'09: Proceedings of the 27th international conference on Human factors in computing systems*, pages 2319–2328, New York, NY, USA, 2009. ACM.

[20] D. M. Mount and S. Arya. ANN: A library for approximate nearest neighbor searching. `http://www.cs.umd.edu/~mount/ANN/`.

[21] L. Nachmanson, S. Pupyrev, and M. Kaufmann. Improving layered graph layouts with edge bundling. In *Proc. 18th Intl. Symp. Graph Drawing (GD '10)*. Springer, to appear.

[22] F. J. Newbery. Edge concentration: a method for clustering directed graphs. *ACM SIGSOFT Softw. Eng. Notes*, 14(7):76–85, 1989.

[23] M. E. J. Newman. Fast algorithm for detecting community structure in networks. *Phys. Rev. E*, 69:066133, 2004.

[24] W. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes: The Art of Scientific Computing*, chapter 10. Cambridge University Press, 2007.

[25] A. Quigley and P. Eades. Fade: Graph drawing, clustering, and visual abstraction. *LNCS*, 1984:183–196, 2000.

[26] University of Florida Sparse Matrix Collection. `http://www.cise.ufl.edu/research/sparse/matrices/`.

[27] D. Watts and S. Strogate. Collective dynamics of "small-world" networks. *Nature*, 393:440–442, 1998.

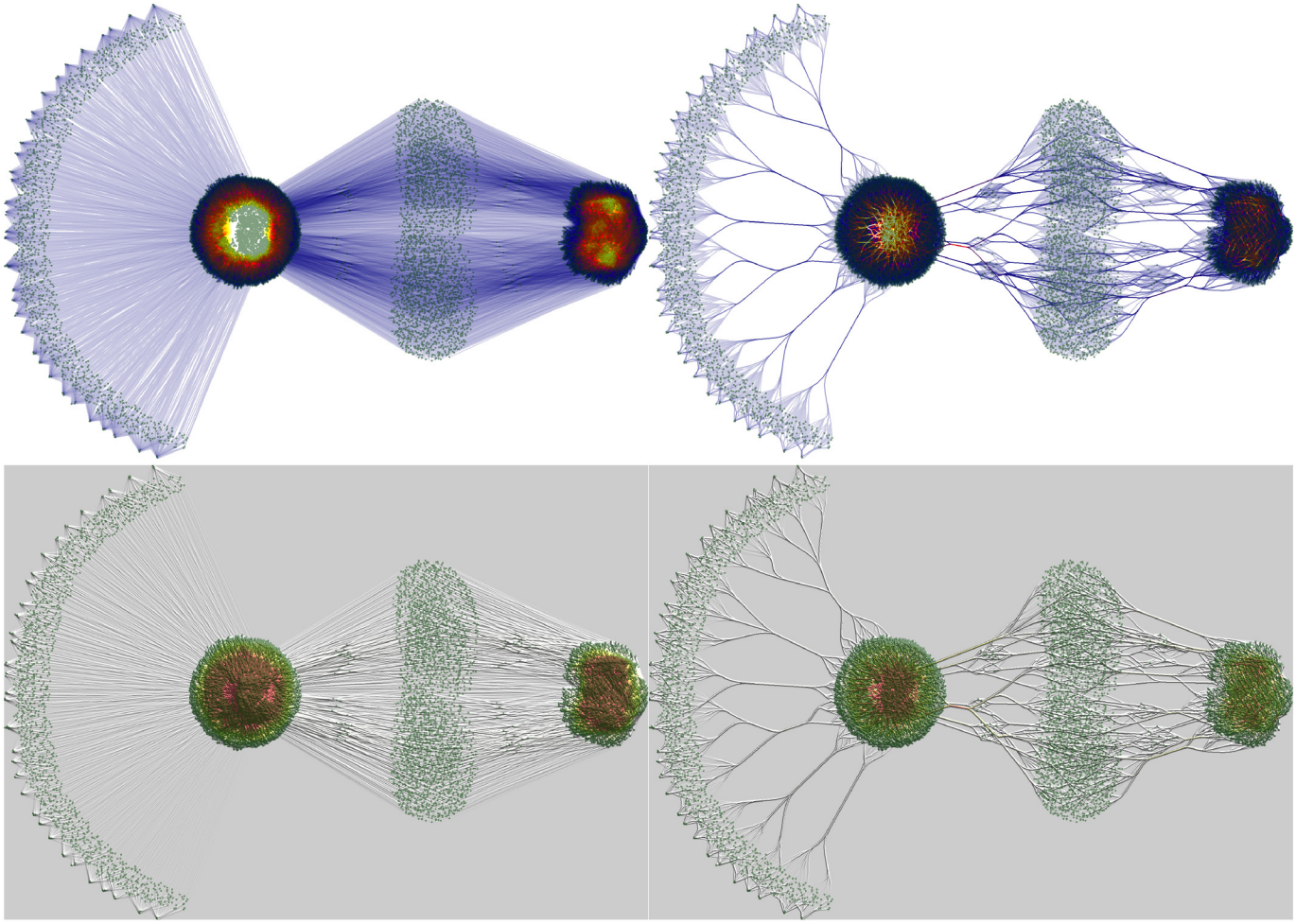

Figure 6: Comparing hillshaded rendering vs standard accumulation-buffer colormap on the largest component of `net50` ($|E| = 429760$). Left: before edge bundling; Right: after edge bundling.

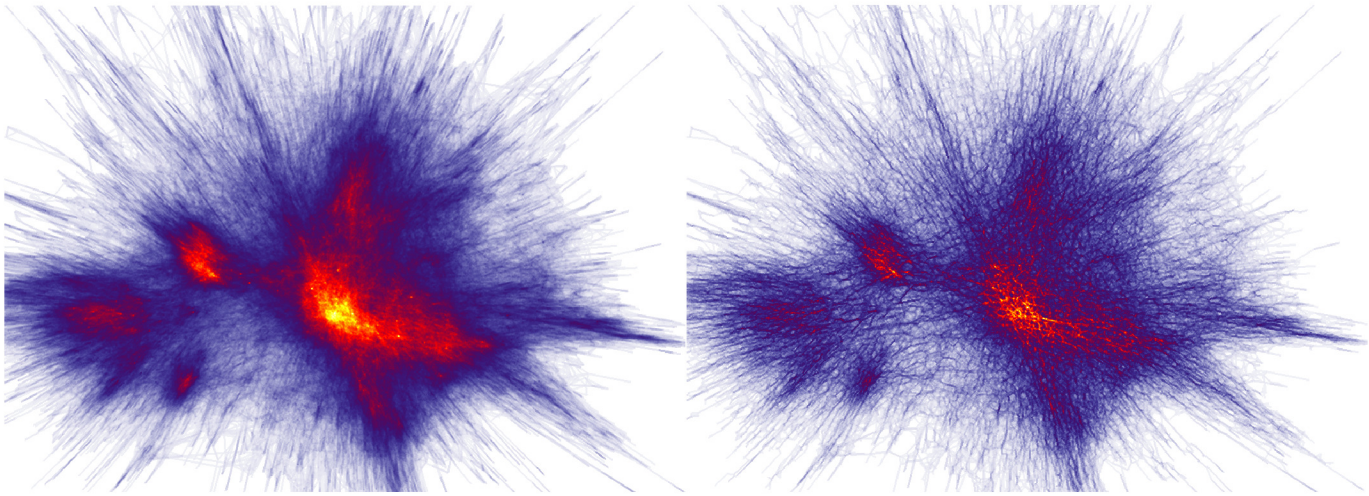

Figure 7: amazon0302 graph ($|E| = 899792$) before and after edge bundling