Geometry-Based Edge Clustering for Graph Visualization

Weiwei Cui, Hong Zhou, *Student Member, IEEE*, Huamin Qu, *Member, IEEE*,
Pak Chung Wong, and Xiaoming Li

Abstract—Graphs have been widely used to model relationships among data. For large graphs, excessive edge crossings make the display visually cluttered and thus difficult to explore. In this paper, we propose a novel geometry-based edge-clustering framework that can group edges into bundles to reduce the overall edge crossings. Our method uses a control mesh to guide the edge-clustering process; edge bundles can be formed by forcing all edges to pass through some control points on the mesh. The control mesh can be generated at different levels of detail either manually or automatically based on underlying graph patterns. Users can further interact with the edge-clustering results through several advanced visualization techniques such as color and opacity enhancement. Compared with other edge-clustering methods, our approach is intuitive, flexible, and efficient. The experiments on some large graphs demonstrate the effectiveness of our method.

Index Terms—Graph visualization, visual clutter, mesh, edge clustering

✦

1 INTRODUCTION

Graphs have been widely used to model many problems such as citations in scientific papers, traffic between telecommunication switches, and airline routes among cities. The scale of these problems keeps increasing and the associated graphs can easily contain tens of thousands of nodes and edges. Visual clutter caused by excessive edge crossings has made traditional layouts no longer effective to convey information. Thus, reducing visual clutter in graphs is a very important research problem. An informative and clear graph layout is critical for clutter reduction.

Many methods have been proposed to improve graph layout. These methods can be classified into two major categories: adjust node positions and improve edge layout. Rearranging the nodes can decrease edge crossings in graphs and thus reduce edge clutter. Node layout methods, such as force-based model algorithm [17], can generate visually pleasing results for small or medium sized graphs according to some aesthetic criteria. However, for dense graphs with a substantial number of edges, rearranging the nodes usually cannot reduce the edge crossings to a satisfactory level. In addition, nodes in some applications such as airline routes have semantic meanings and it may not be appropriate to move their positions. Another promising approach to reduce visual clutter is to bundle edges. For example, a flow map layout [18] is proposed for single-source graphs while Edge Bundles [12] are designed for visualizing datasets containing both hierarchical structures and adjacency relations. Their results demonstrate the high potential of using edge clustering to improve the graph layout and reduce visual clutter. However, these previous solutions are all designed for special graphs such as source-sink style graphs and graphs with known hierarchical structures. An efficient edge-clustering solution for general graphs is still missing.

In this paper, we follow the same line of research by bundling edges to reduce visual clutter. Our goal is to design an edge-clustering framework for general graphs. Our method is inspired by road maps, which are visually pleasing and relatively uncluttered. There are some good features of road maps. First, in road maps, the connection between two nodes is no longer a straight line; it is turned into segments that consist of cities and highways. Second, the road maps can be viewed at different levels of details (i.e., country level, city level, and county level). Third, by studying the road map, some high-level patterns can be detected. For example, the major highways usually indicate heavy traffic along the highway direction. Therefore, we believe that turning straight line graphs into road-map-style graphs may effectively reduce clutter and help detect the underlying patterns in the data.

It is not easy to generate informative road-map-style graphs for general straight line graphs. The major components of road maps are cities and roads. We can consider cities as control points and roads as segments connecting cities. Then all paths must pass through certain cities and roads. One major challenge is how to choose control points (i.e., cities in road maps) for general graphs. We find that a good control point should be close to the point with high line density, which means heavy traffic, and the edge connecting control points should be aligned with the primary line direction, which means the major traffic direction. In addition, the influence of the control points should be localized (i.e., only edges within a certain distance can pass through a control point). Based on these intuitive observations, we design a geometry-based edge-clustering framework for general graphs. The basic idea is to select control points based on a control mesh that reflects the underlying graph patterns. We first analyze the link distributions and detect a primary direction for each local area. Then, we generate a control mesh with edges piercing through the cluster of lines. The control points will then be positioned on the mesh edges. By forcing all links to pass through these control points, edge bundles can be naturally formed. To further improve the layout, we introduce a local-smoothing scheme to smooth all the zigzag curves. We then provide some advanced visualization techniques to enhance the patterns after edge clustering. Compared with previous methods, our method can work on general graphs, and it is geometry-based so expensive optimization is avoided. It is intuitive, allowing users to easily control the final layout by adjusting the control mesh and the control points. The control meshes can be easily constructed in a hierarchical way, so users can examine the graphs at different levels of detail.

The major contributions of this paper are as follows:

- We propose a general edge-clustering framework based on control meshes to reduce visual clutter and enhance patterns in graphs. Our framework is intuitive, flexible, and efficient.

- We present several schemes to generate control meshes that can capture the underlying edge distribution patterns and lead to informative graph layouts. A local-smoothing scheme is proposed to further improve the layout quality.

- We introduce three advanced visualization and interaction techniques (i.e., color and opacity enhancement, mesh adjustment, and animation) which can significantly increase the effectiveness of edge-clustered graphs.

- *W. Cui, H. Zhou, and H. Qu are with the Hong Kong University of Science and Technology, E-mail: {weiwei|zhouhong|huamin}@cse.ust.hk.*
- *P. C. Wong is with Pacific Northwest National Laboratory. E-mail: pak.wong@pnl.gov.*
- *X. Li is with Peking University. E-mail: lxm@pku.edu.cn.*

2 RELATED WORK

Visual clutter in graphs has been extensively studied in the graph drawing and information visualization fields. In this section, we only review the papers that are closely related to our work (i.e., graph layout). Thus, we omit other effective clutter reduction techniques such as sampling, filtering, clustering, and animation because of the limited space. An excellent survey on general clutter reduction techniques can be found in [7].

Many efforts have been devoted to generate good graph layouts [2, 15]. They can be divided into two major categories: node-based techniques and edge-based techniques. Node-based techniques focus on adjusting node positions to improve the overall graph layouts while edge-based techniques try to reduce visual clutter by either dispersing or clustering edges.

Node layout Rearranging the nodes can decrease the number of edge crossings and thus reduce visual clutter. Force-based methods are widely used in node layout algorithms. In force-based approaches, graphs are considered as physical systems, in which nodes are modeled as rigid bodies, and edges are modeled as elastic springs. According to different aesthetic criteria or specific requirements, appropriate energy models [4, 8, 20, 17] can be formulated. In general, force-directed algorithms can successfully produce good results for relatively small graphs, but they do not scale well with size. Large graphs often make the energy function difficult to be optimized. To improve the time performance, fast multilevel algorithms [1] and simplified energy functions [16] are proposed. Recently, Frishman and Tal [9] introduced a GPU-accelerated force-based model that can provide a promising speedup to generate high-quality layouts for large graphs. To visualize large graphs at different level of details, a topological fisheye view technique [10] has been proposed to allow users to interactively examine local areas of a graph in detail and still preserve the display of the graph's global structure. Compared with the above node layout methods, our approach does not change node positions or merge nodes. For some applications such as communication and transportation networks, node layout methods are not applicable because the semantic meanings of the node positions prevent spatial adjustment of nodes.

Edge dispersing For dense graphs with a large number of edges, a good node layout cannot reduce the edge clutter to a satisfactory level. Thus, various methods are proposed to further adjust edges. One significant approach is to disperse edges away from a local area so the underlying patterns can be revealed. Wong et al. [21] introduced EdgeLens for interactively managing edge congestion in graphs. Without changing node positions, they displaced edges in a local area with a high degree of edge overlap to reveal hidden information in that area and thus clarified graph structures. Wong and Carpendale [22] further proposed another interactive technique, Edge Plucking, which temporarily pulls edges apart to clarify underlying node-edge relationships. These interactive graph exploration tools are very useful to reveal the local structures in a region of interest, while our method aims at revealing the global structures and large-scale patterns of the graph. Actually, our method can complement the strengths of edge-dispersing techniques and can be used together with them.

Edge clustering Another kind of edge-based techniques focuses on merging edges to reduce visual clutter. Confluent drawings [5] exploit curves to visualize non-planar node-link diagrams in a planar way. However, not all the graphs can be drawn confluently. In addition, the complexity of deciding whether a graph is confluent or not remains open [13]. By curving and merging edges, Phan et al. [18] presented flow map layouts to draw single-source graphs whose edges share a common end point as a "free-style" binary tree. Considering the common end point as the tree root, the algorithm automatically generates a hierarchical structure based on the leaf positions. By making the line widths proportional to the edge weights, a flow map can provide a clear flow distribution and reduce visual clutter. Their results are very encouraging; however, it is not clear how to extend their method to general graphs.

Edge Bundles [12] are designed for visualizing datasets containing both hierarchical structures and adjacency relations, such as ref-

erence relations among the elements of a file directory. Linking two leaf nodes in the tree, each edge is curved according to the tree path that connects the two leaf nodes. If two edges share some segments in their tree paths, they will be bundled at those common segments. This method demonstrates the effectiveness of using curves to reduce visual clutter, but the technique is designed for graphs with known hierarchical structures. Gansner and Koren [11] improved circular layouts by grouping edges to maximize area utilization and readability. Compared with the previous work on edge clustering, our method works for general graphs. Qu et al. [19] proposed a novel edge-clustering framework for general node-link diagrams. By grouping links based on their intersections with the edges in the Delaunay triangulation of the nodes, this method reduces edge clutter and gives an overall abstraction of graphs. However, for large graphs, their method generates many zigzag edges, making it difficult for users to discern the curve direction and end points. Our method introduces a local-smoothing technique to address the zigzag problems. In addition, we demonstrate that using Delaunay triangulation does not work for many graphs; therefore, we design another mesh generation method that can better capture the underlying graph patterns for edge clustering. Furthermore, three novel visualization and interaction techniques are introduced to make our method more effective.

3 GEOMETRY-BASED EDGE CLUSTERING OVERVIEW

In this section, we give a brief overview of our edge-clustering framework. We assume that the positions of the nodes in the input graph are already available. For some applications, node positions encode geographic information and any dramatic adjustment of node positions may cause confusion for users. For other applications, the positions of nodes can be computed by methods such as force-based models [17] and thus a relatively good initial layout can be obtained. Therefore, we do not further change node positions and the original node layout is preserved. Our goal is to convert general straight line graphs into road-map-style graphs, and the basic idea of our method is to cluster the edges based on a control mesh that reflects the underlying graph structures.

Fig. 1 illustrates the framework of our approach. It consists of three major steps: 1) control mesh generation, 2) edge clustering, and 3) visualization. Control mesh generation has two components: graph analyzer and mesh generator. The node and edge information of the original graph is first sent to the analyzer to detect underlying edge distribution patterns. After that, some representative primary edge directions are output to the mesh generator, which then generates some mesh edges perpendicular to each selected primary direction. These mesh edges serve as basic control-mesh edges. By further adding more mesh nodes and triangulating the nodes and basic edges, the mesh generator completes the control mesh and sends it to the bundler. Based on the intersections between the original graph and the control mesh, the edge bundler sets some control points on the control-mesh edges and curves the original graph edges to pass through these control points to form edge clusters. In the edge smoother, some curved edges with too many zigzags are further fine-tuned to become visually pleasing. Finally, in the visualizer, an intuitive exploration interface is provided for users to interact with the edge-clustering results.

4 CONTROL MESH GENERATION

We use a triangle mesh, called control mesh, to guide the edge clustering process. The control mesh plays a very important role in the edge-clustering process and is critical for the final graph layout. A good control mesh will lead to an informative layout, which can reduce the number of edge crossings, bundle edges with similar directions and lengths, and minimize the distances between original straight-line edges and resulting polyline edges. In other words, a good layout should faithfully reveal and enhance the underlying graph patterns and effectively reduce visual clutter. In this section, we first discuss the overall strategy for control mesh generation and then introduce three mesh generation methods.
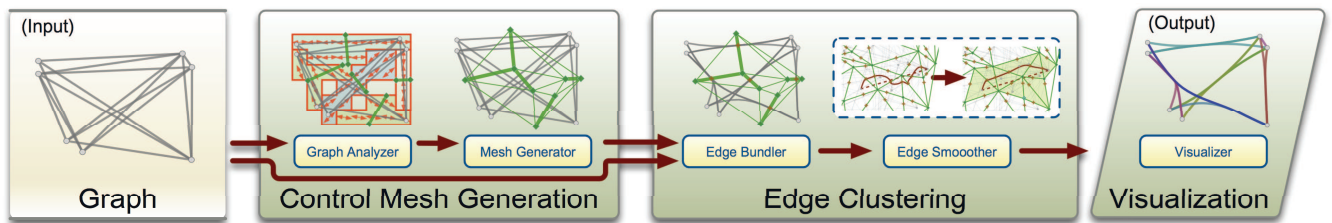
Fig. 1. Framework of our graph visualization system.

4.1 Control Mesh Generation Strategy

The control mesh should be generated based on the underlying graph structures. One simple strategy is to generate control meshes based on the node distributions. For example, we can triangulate the nodes into a control mesh using Delaunay triangulation [19]. However, we find that this kind of control mesh does not work for many graphs because the underlying edge distribution is not taken into account. Fig. 2a shows such an example. If we use the Delaunay mesh in Fig. 2b as the control mesh, the edge cluster linking the east nodes and the west nodes cannot be bundled together. Therefore, a good control mesh should not be computed solely based on the nodes of the graphs.
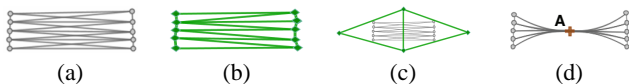

Fig. 2. Control meshes: (a) a graph; (b) a control mesh generated by the Delaunay triangulation of nodes. With this mesh, no edges in graph (a) will be clustered. (c) a control mesh generated according to the edge distribution pattern; (d) the layout after clustering the edges along the control point A.

One of the most interesting patterns in a graph is the edge clusters consisting of edges with similar directions and lengths. If these edges are bundled together, the visual clutter can be reduced (see Fig. 2d). Therefore, the control mesh should facilitate grouping spatially close edges with similar directions. In order to do so, some control points (e.g., control point A in Fig. 2d) must be located in the middle of the edge cluster. After these edges are forced to pass through the control points, edge bundles can be generated accordingly. Because our control points are set on the mesh edges, we need to make some mesh edges (e.g., the vertical green edge in Fig. 2c) crossing the edge cluster. Therefore, our control mesh generation strategy is to first detect edge clusters manually or automatically and then generate mesh edges to pierce through these edge clusters.

4.2 Manual Mesh Generation

One straightforward solution is to allow users to manually generate a control mesh according to the data. Our system can provide some visual cues such as edge densities and direction variations to users. The basic guideline is that some edges in the control mesh should cross edge clusters. Based on this guideline, users can either manually set vertices around the edge clusters or directly draw mesh edges crossing through the clusters. Users can draw the whole mesh by themselves or let our system automatically connect these chosen vertices and edges to form a triangle mesh. Fig. 3 illustrates such an example. Fig. 3a shows the original graph. We can clearly see that there are some clusters of almost parallel lines. Users can then directly click on the graph display to generate a set of vertices (see Fig. 3b) and edges, which can then be connected to form a triangle mesh (see Fig. 3c) by Constrained Delaunay triangulation [3]. For simple graphs with some obvious edge cluster patterns, users can manually set the mesh and thus obtain the final edge-clustering results. For some dense graphs, it becomes difficult and time-consuming for users to visually find the edge clusters and set the entire mesh manually; therefore, we introduce two more sophisticated mesh generation schemes in the next two subsections.
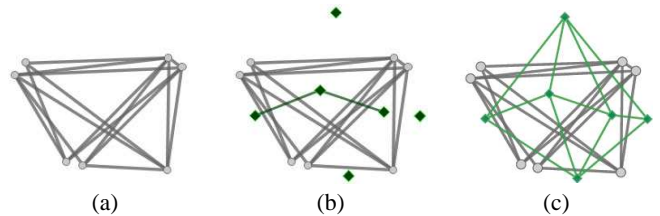

Fig. 3. Manual mesh generation: (a) a graph; (b) users click a set of vertices and edges; (c) a mesh is generated by Constrained Delaunay triangulation of the vertices and edges.

4.3 Automatic Mesh Generation

A better solution is to automatically generate a control mesh by analyzing the underlying edge patterns. Fig. 4 illustrates the basic idea of our automatic mesh generation method. We first compute the bounding box for the input graph. Then we divide the bounding box into cells using a regular grid (see Fig. 4b). The resolution of the grid can be configured by users. For each grid cell, we compute the number of nodes falling into this region and the number of links passing through it. A feature vector can be constructed to record the direction of each passing link. Then we use Kernel Density Estimator [6] to detect whether there is a strong clustering of those feature vectors. If so, this clustered direction will be selected as a primary direction of this cell (see red arrow in Fig. 4b). Otherwise, this cell will be ignored in the following steps. Next, we merge smaller regions with similar primary directions into some larger regions (see thick red polygons in Fig. 4b) until the maximum angular difference of primary directions in the region is beyond a threshold (e.g., 15°) specified by users. Then the weighted average of the primary directions in the smaller regions will become the primary direction of the resulting larger region. For each region, we want to cluster the links along the primary direction and minimize the average distance between the clustered line and the original straight lines. To achieve this goal, we found that it is better to make mesh edges pierce through the clusters and become perpendicular to the clusters' primary direction. Under this guideline, our system can automatically generate a set of mesh edges (see green edges in Fig. 4c). After processing all grids, we get a set of vertices and edges. We first merge some vertices which are too close to one another, and then we use Poisson sampling to generate more vertices if needed. Finally, a triangle mesh as shown in Fig. 4d can be generated using Constrained Delaunay triangulation [3]. This automatic approach is used as the default mesh generation method for the remaining sections.

4.4 Hierarchical Mesh Generation

The level-of-detail graph visualization can be achieved through a set of hierarchical control meshes. The hierarchical meshes can be generated in two ways: discrete level-of-details and continuous level-of-details. The concept of continuous level-of-details is borrowed from the computer graphics field. It indicates a smooth transition from a high-resolution mesh to a low-resolution mesh by edge collapse. In the automatic mesh generation process, we allow the merging of smaller regions with similar primary directions into a larger region based on a user-specified angular difference threshold. The discrete level-of-
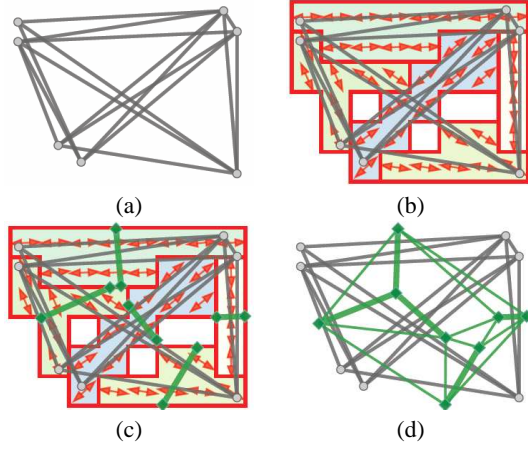
Fig. 4. Automatic mesh generation: (a) a graph; (2) grid the graph, calculate a primary direction for each grid and merge them based on their primary directions; (3) set some mesh edges perpendicular to the blocks' primary directions; (4) link the edges together to generate a mesh.
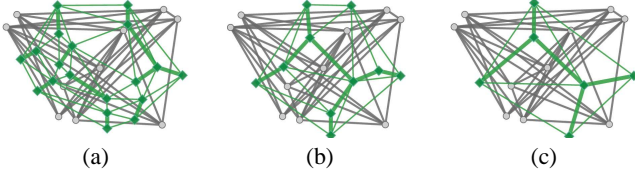

Fig. 5. The hierarchical meshes generated using three angular difference thresholds, i.e., 5°, 12°, and 40°, respectively.

details can be achieved by specifying a series of discrete thresholds (e.g., 5°, 10°, 15°) and then generate the control meshes accordingly. The continuous level-of-details can be constructed by merging cells one by one based on the difference of the primary directions. The two neighboring cells with the smallest difference of primary directions will get merged first. After each merge, we can generate a new control mesh that has fewer triangles than the previous one. We keep doing the merging and then a sequence of control meshes with continuous level-of-details can be generated.

Another possible way to generate hierarchical meshes is to change the grid resolution. For example, the graph region can be divided into 64×64, 128×128, and 256×256 grids, which lead to three discrete levels of hierarchical meshes. For continuous level-of-details, we can start from a high-resolution control mesh and then simplify it using some well-established computer graphics techniques such as vertex merging or quadratic error metrics. Because mesh simplification is thoroughly studied, we can leverage those advanced techniques to achieve sophisticated graph visualization results. The flexible level-of-detail control is a major advantage of our geometry-based framework. Fig. 5 shows the control meshes at three discrete level-of-details.

Our automatic mesh generation methods can guarantee that the control meshes are generated solely based on information from the data and most likely reflect the underlying edge patterns. Even with these automatic methods, manual mesh generation may be still useful as it enables users to create control meshes in some local areas where the automatic methods fail to generate adequate mesh edges.

5 EDGE CLUSTERING

After we have the control mesh, the next step is to compute the control points and conduct edge clustering based on the control mesh and control points. In this section, we first introduce a straightforward edge-clustering scheme. Then we present a local-smoothing method to address some unwanted features in the clustered graph.
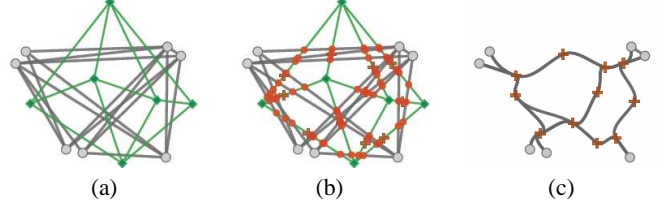

Fig. 6. Edge clustering by control points: (a) a graph with a control mesh; (b) the intersections and the control points; (c) the merged graph.

5.1 Edge Clustering by Control Points

Fig. 6a shows a graph and the corresponding control mesh. All the intersection points between the links and control-mesh edges are shown as red dots in Fig. 6b. Intuitively, the control point(s) on each edge should be in the center of these intersection points. Then, after original links are forced to pass through the control point(s) instead of the intersection points, the overall distortion can be minimized. Therefore, we apply the K-means clustering method to compute one or several control points for each edge. After forcing the links to pass through these control points, we can get an edge-clustered graph (see Fig. 6c). The method is intuitive to use, and different graph layouts can be generated by using different control meshes and control points. In addition, the merged curves can be drawn using different curve styles.

5.2 Local Smoothing

The edge-clustered result generated by the previous method may not be visually pleasing because some edges may have too many zigzags. Fig. 7 illustrates this problem. Fig. 7a shows one original straight-line edge (dotted red line) and the resulting polyline edge (solid red line), which has severe zigzag. The zigzag edge is not pleasing and can even indicate wrong direction of the original link and thus cause misleading comprehension of the graph.

To alleviate the problem and make the edges as smooth as possible, we introduce a local-smoothing algorithm. Because each straight line becomes a polyline in the final layout, we first develop a quality metric to measure how well the polylines represent the original straight lines. The quality metric should consider the polyline's curvature, the number of turning points, and the maximum distance between the polyline and the original straight line. After experiments with various metrics, our path quality for a polyline edge e is quantitatively modeled as follows:

$$\mathbf{Q}(e) = \alpha \mathbf{Q}_{angle}(e) + \beta \mathbf{Q}_{distance}(e)$$

where $\mathbf{Q}_{angle}(e)$ and $\mathbf{Q}_{distance}(e)$ are the two terms computing the angle and distance variation. α and β are the corresponding weights for each term. The first term $\mathbf{Q}_{angle}(e)$ is defined as follows:

$$\mathbf{Q}_{angle}(e) = -\sum_{i=3}^{n} \gamma_i |\Delta_i|$$

We assume that e consists of n segments and $(n-1)$ control points. Δ_i records the angular difference between the ith segment and the $(i-1)$th one. Boolean variable γ_i indicates whether there is a zigzag or direction change for control point i. The formulations of Δ_i and γ_i are listed below:

$$\Delta_i = \begin{cases} A_i - A_{i-1} & \text{if } -\pi < |A_i - A_{i-1}| < \pi \\ |A_i - A_{i-1}| - 2\pi & \text{if } |A_i - A_{i-1}| > \pi \\ 2\pi + |A_i - A_{i-1}| & \text{if } |A_i - A_{i-1}| < -\pi \end{cases}$$

where A_i is the radian angle formed by ith segment and the original straight line e'.

$$\gamma_i = \begin{cases} 0 & \text{if sign}(\Delta_i) = \text{sign}(\Delta_{i-1}) \\ 1 & \text{if sign}(\Delta_i) \neq \text{sign}(\Delta_{i-1}) \end{cases}$$
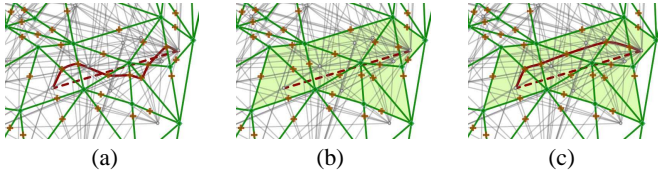
Fig. 7. Local smoothing: (a) a zigzag path (solid red line) without smoothing; (b) the search region (solid green region) to find the smoothest path; (c) the smoothest path (solid red line) found in this region. The dotted red line is the original straight line.

The second term $Q_{distance}$, which is to record the distance variation between the curved edge e and the straight line e', is approximated by the following equation:

$$\mathbf{Q}_{distance} = -\sum_{l=1}^{n-1} D_l$$

where D_l is the Euclidian distance from the lth control point to the straight-line edge e'. If users want the curved lines to have fewer zigzags, they can choose a large α value. If the curved lines should not be far away from their original positions, a large β value should be used.

Based on this quality measure, we can identify a set of polylines with poor quality and then do local smoothing for them. The basic idea is to find another path or a set of control points in a local area for the corresponding original edge of each poor polyline. The first step of our local-smoothing algorithm is to compute a local area for this edge to narrow the search space for the new path. All the triangles in the control mesh that the edge passes through and some neighboring triangles whose vertices are within a certain distance threshold to the edge will form the search region. The distance threshold can be configured by users. A larger threshold will result in a larger search region and a better chance that a smooth path can be found but at the cost of longer computation time. After that, we just search all the possible paths in the search region for the original link and choose the one that has the highest quality according to our quality metric. We exploit dynamic programming and thus the local-smoothing algorithm can be run at $O(n^2)$ time complexity, where n is the total number of triangles in the search region. Compared with the global optimization used in energy-based methods, our local smoothing can be performed much faster as it is a one pass process with narrowed search space. Fig. 7c shows the computed smooth path.
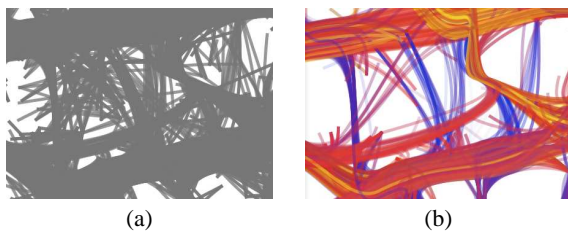

Fig. 8. Color and opacity enhancement: (a) an edge-clustered graph; (b) the graph after color and opacity enhancement. The color encodes the orientations of original links and the opacity indicates the line density of overlapped segments.

6 VISUALIZATION TECHNIQUES

The layout generated by our edge-clustering method can be further explored with some advanced visualization techniques such as color and opacity enhancement, mesh adjustment, and animation.

6.1 Color and Opacity Enhancement

For dense graphs, the patterns may still be obscured after edge clustering because of occlusion. To reveal these patterns, we can assign different colors and opacities to edge segments based on certain attributes.

After edge clustering, we can compute various attributes for the polyline segments of the graph. For example, each polyline segment in the edge-clustered graph may represent a certain number of original edges; the distance of each polyline segment to the original straight lines may be different; and these original straight lines may have different directions. Therefore, we can compute the line density, the average distance of these edges to their original links, and the direction variation for each polyline segment. We can then design a transfer function that maps these attributes to color and opacity values to enhance different patterns in the graph. For example, for a polyline segment e, we can compute its weighted density attribute as follows:

$$D_e = \sum_{i=1}^{n} c_i l_i$$

where n is the number of e's sub-segments, c_i is the number of original straight-line edges that are bundled or merged into the ith sub-segment, and l_i is the length of the ith segment serving as the weight.

We design an interface similar to the transfer function specification in volume rendering and parallel coordinates [14] to assign the color and opacity values based on different attributes. Users can then interactively manipulate the transfer function and thus selectively enhance different edge bundles. Fig. 8 shows an example of using color and opacity enhancement.
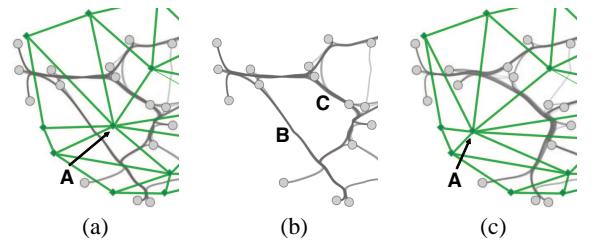

Fig. 9. Mesh adjustment: (a) one control mesh; (b) the result with control mesh a; (c) after moving mesh node A, edge bundle B and C in (b) are merged into one bundle.

6.2 Mesh Adjustment

To further explore the data, users can interactively adjust the control mesh so that different layouts may be generated. In this way, different clusters may be revealed. Some typical mesh adjustment operations include: adjusting vertex positions; merging two vertices; splitting an edge; subdividing a triangle into four sub-triangles. By adjusting the meshes, some otherwise separated clusters may get merged. Fig. 9 demonstrates that different meshes can lead to different graph layouts.

6.3 Animation

Different animation schemes can be used together with our edge-clustering method. For example, we can change the level of clustering to allow the edges to be grouped instead of being merged such that each individual edge is still discernible. We can also generate an animation to show the whole process of edge clustering, i.e., how edges are changed from straight lines to polylines and then gradually merged together. In our system, we provide two animation techniques: animated transitions from the original straight line graph to the resulting edge-clustered graph, and animated sequences to display the layout at different levels of detail. By viewing the animations, users will have a better idea about the data and may detect some patterns that may otherwise disappear in the final static layouts. Fig. 10 shows some frames during an animation sequence, which shows the transition from a straight line graph to an edge-clustered graph.
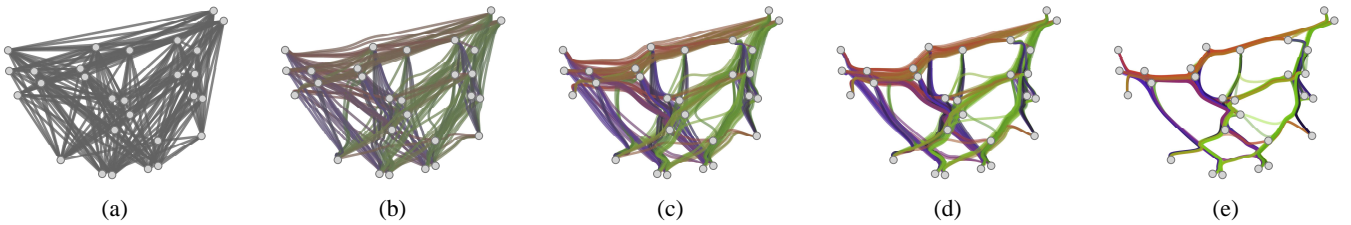
(a) (b) (c) (d) (e)

Fig. 10. An animation sequence for an edge-clustering process. The color is used to encode the edge directions.
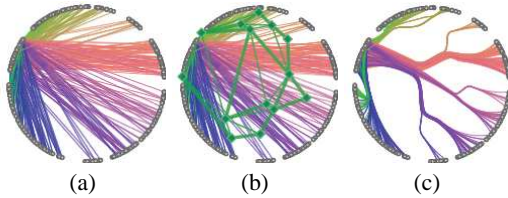

(a) (b) (c)

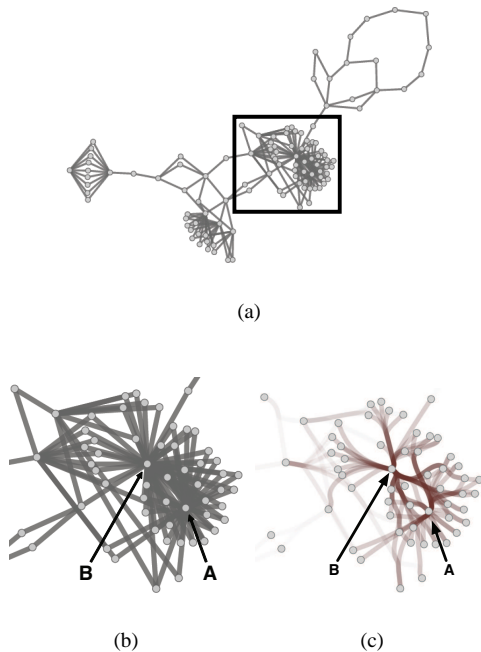Fig. 11. Edge clustering on a synthesized dataset.

(a)

(b) (c)

Fig. 12. Experiments on the GD'96 contest data.

7 EXPERIMENTAL RESULTS

In this section, we apply our geometry-based edge-clustering method to several graphs and demonstrate the effectiveness of our approach.

First we tested our method with a synthesized graph with simple patterns. Fig. 11a shows a layout which is also used in [12]. Fig. 11b is the control mesh automatically generated based on the underlying edge patterns. Then an edge-clustered graph layout that is similar to the result using Edge Bundles [12] can be easily generated using our method (see Fig. 11c). This example demonstrates that our method works well for graphs with simple patterns.

Next we tested our algorithm on a benchmark dataset used in the Graph Drawing 96 contest [1]. Fig. 12a shows the result of this graph using a force-based method [23]. As pointed out in [23], the force-based approach can reveal most of the major features in this dataset,

[1] http://www.research.att.com/conf/gd96/contest.html

except one root node "A" in the rectangular area that is enlarged in Fig. 12b. Another root node "B" is clearly shown in Fig. 12b, while root node "A" is embedded in a massive number of nodes and edges. However, after applying our edge-clustering method, both root nodes are highlighted by dark red edges linking to them (see Fig. 12c), because our method can successfully detect and enhance the edge bundles with high density and then encode them with high opacity values.

The third dataset is about the major airline routes of Northwest Airlines in the United States. Fig. 13a shows the original graph. Because of severe clutter, not much information is revealed. After applying our method, some high-level patterns are revealed (see Fig. 13b). From the result, we can clearly see that there are some major clusters of airline routes going from the west coast to the east coast, while the directions of the airline route clusters are more diversified in the northeast region. After zooming into the northeast region, more details are displayed with our hierarchical control meshes (see Fig. 13d and 13f). Fig. 13e and 13g show the results after applying different transfer functions. One disadvantage of our approach is that the individual link direction and length information is lost after edge-clustering or edge-bundling. However, we can compensate for this by color encoding. For example, in Fig. 13e where edges are bundled instead of merged, we can use color to encode the original edge directions. Red indicates east-west direction while blue means north-south direction. In Fig. 13e, we can see that edge bundle "A" mainly consists of red colors; therefore, most of its edges connect the east region and the west region. The blue edge bundle "B" has some orange edges in it, which means that some of its edges are linking the northeast region and the southwest region. In Fig. 13g, color is used to encode the edge length information. Blue indicates short edges while red means long. We can easily find that edge bundle "C" consists of some long edges (red) and also some relatively short edges (blue). Therefore, our color and opacity enhancement tool can further help users explore the clustered graph by providing more information about the original edge attributes.

The last example is a dense graph, representing the migration among the states in the United States. The same dataset has also been used in [18]. The straight line graph layout (see Fig. 14a) has numerous line crossings that obscure any patterns and is therefore impossible to interpret. After applying our method, some patterns become visible as shown in Fig. 14b, but some parts are still very fuzzy (see the rectangle region of Fig. 14b). We then applied a transfer function based on the number of gross migration (i.e., the sum of inmigration and outmigration). We used red to encode the highest gross migration value and blue to encode the lowest value. The patterns are beautifully revealed. For example, the state of California has thick red edges linking to it. This state is also the most active state with highest gross migration numbers. Fig. 14d shows the result after applying the same kind of transfer function without edge clustering. Not much pattern is revealed. This example clearly demonstrates that our method can reveal the patterns in a very large graph and the color and opacity enhance scheme will be especially effective after edge clustering. Fig. 14e shows a flow map result [18] that only reveals the immigration from a west coast city. Our graph can reveal much more information than a single flow map because the overall context is also displayed with the flow map. In a sense, our method can be thought of as embedding multiple flow maps into one graph display.

We implemented our algorithm on a Macbook Pro with Intel Core 2 Duo 2.2GHz CPUs and 2GB Memory. The computation times of our

edge clustering for datasets used in Fig. 13 and Fig. 14 are 2.5s and 12.9s respectively. There are some configurable parameters in our system but our default setting works well for many graphs. For example, all the experiments in this paper were generated using grid size $= 30 \times 30$, angular threshold $= 20°$, $\alpha = 0.7$, and $\beta = 0.3$. These parameters are intuitive to use so users can easily change them to reveal different patterns. For example, a large grid size and a large angular difference threshold will result in a layout revealing large patterns, and vice versa.

8 DISCUSSION

One major advantage of our framework is that our system is highly configurable and provides excellent interactions. It is very intuitive to adjust the layout. We give users great control and flexibility in the whole process. In addition, our method can be computed very fast. Very efficient algorithms exist for all the major computation tasks in our algorithm. For example, Delaunay triangulation, line-triangle intersections, and K-means clustering can all be accelerated by GPUs. Our method is also very stable. Changing a graph node or control-mesh node position will be localized and will not dramatically affect the whole layout. Furthermore, our method can scale well with data size and elegantly handle levels of detail.

Compared with data-domain filtering and clustering techniques, our method shows all the data to users and lets users identify the patterns in the data. We do not remove any data item from the graph display. Instead, we enhance the patterns and then show both the patterns and context to users. Users can easily use transfer functions to emphasize the patterns and suppress the context. Compared with other curve-based graph layout methods [12, 18], our framework works for general graphs. To the best of our knowledge, our method is the first framework to generate road-map-style layouts for general graphs. As demonstrated in Fig. 11c and Fig. 14e, for special graphs, our method can generate similar results as some previous work.

Curves and straight lines all have their advantages and disadvantages. Straight line graphs are good at revealing the line direction and the connection between two nodes, while curve graphs are good at showing clusters and making the overall layout more discernible. Therefore, we suggest that users use our system along with straight line systems. In some situations, the patterns may be better perceived using straight-line graphs. Fortunately, users can easily switch back to the original straight line layout using our system.

Our method also has some weaknesses. The effectiveness of our approach highly relies on the quality of control meshes. Even though we introduce an automatic mesh generation algorithm and provide some visual cues for manual mesh generation, there is no guarantee that an effective mesh can always be obtained. The global topology of the original graphs may not be preserved in the edge-bundled layouts and the edge bundles created by our method may not have strong semantic meanings. Our method focuses on finding the clusters of lines with similar directions. For data without such patterns, our approach cannot help much. If users are interested in information such as connectivity between two nodes, other representations such as matrix can be used together with our system.

9 CONCLUSION AND FUTURE WORK

We have presented a mesh-based edge-clustering method for graphs. Our approach is intuitive, efficient, and highly configurable. We introduced different control mesh generation techniques that can capture the underlying edge patterns and generate informative and less cluttered layouts. The quality of clustered graphs can be further improved by local smoothing. Several advanced visualization techniques are specifically designed for edge-clustered graphs. Our method can improve the layouts generated by other methods such as force-based models, and provide excellent user interactions, which are critically important for large graphs. Users can easily change the layout by adjusting the mesh and transfer function.

There are several avenues for future work. Triangle meshes are currently used as the control meshes in our system. We will investigate other types of control meshes such as curvilinear grids. The current color and opacity enhancement scheme is still primitive. More sophisticated transfer function design schemes taking both node position and edge directions into consideration will be explored.

REFERENCES

[1] D. Archambault, T. Munzner, and D. Auber. TopoLayout: Multilevel graph layout by topological features. *IEEE Transactions on Visualization and Computer Graphics*, 13(2):305–317, 2007.

[2] G. D. Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing; Algorithms for the Visualization of Graphs*. Prentice Hall, 1999.

[3] L. P. Chew. Constrained Delaunay triangulations. In *Proceed. of the Symposium on Computational Geometry*, pages 215–222, 1987.

[4] R. Davidson and D. Harel. Drawing graphs nicely using simulated annealing. *ACM Transactions on Graphics (TOG)*, 15(4):301–331, 1996.

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

[6] R. O. Duda and P. E. Hart. *Pattern Classification and Scene Analysis*. Wiley, 1973.

[7] G. Ellis and A. Dix. A taxonomy of clutter reduction for information visualisation. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1216–1223, 2007.

[8] A. Frick, A. Ludwig, and H. Mehldau. A Fast Adaptive Layout Algorithm for Undirected Graphs. *Proceedings of the DIMACS International Workshop on Graph Drawing*, pages 388–403, 1994.

[9] Y. Frishman and A. Tal. Multi-level graph layout on the GPU. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1310–1319, 2007.

[10] E. Gansner, Y. Koren, and S. North. Topological fisheye views for visualizing large graphs. In *Proc. of the IEEE Symp. on Information Visualization*, pages 175–182, 2004.

[11] E. R. Gansner and Y. Koren. Improved circular layouts. In *Proceed. of Symposium on Graph Drawing*, pages 386–398, 2006.

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

[13] P. Hui, M. Pelsmajer, M. Schaefer, and D. Stefankovic. Train Tracks and Confluent Drawings. *Procced. of Symposium on Graph Drawing*, pages 465–479, 2004.

[14] J. Johansson, P. Ljung, M. Jern, and M. Cooper. Revealing structure within clustered parallel coordinates displays. In *Proc. of IEEE Symp. on Information Visualization*, pages 125–132, 2005.

[15] M. Kaufmann and D. Wagner. *Drawing Graphs: Methods and Models*. Springer, 2001.

[16] Y. Koren, L. Carmel, and D. Harel. Drawing huge graphs by algebraic multigrid optimization. *SIAM Multiscale Modeling and Simulation*, 1(4):645–673, 2003.

[17] A. Noack. An energy model for visual graph clustering. In *Proceed. of Symposium on Graph Drawingg*, pages 425–436, 2003.

[18] D. Phan, L. Xiao, R. Yeh, P. Hanrahan, and T. Winograd. Flow map layout. *IEEE Symposium on Information Visualization 2005*, pages 219–224, 2005.

[19] H. Qu, H. Zhou, and Y. Wu. Controllable and progressive edge clustering for large networks. In *Proceed. of Symposium on Graph Drawing*, pages 399–404, 2006.

[20] F. van Ham and J. J. van Wijk. Interactive visualization of small world graphs. *IEEE Symposium on Information Visualization 2004*, pages 199–206, 2004.

[21] N. Wong, M. Carpendale, and S. Greenberg. Edgelens: An interactive method for managing edge congestion in graphs. *IEEE Symposium on Information Visualization 2003*, pages 51–58, 2003.

[22] N. Wong and S. Carpendale. Interactive poster: Using edge plucking for interactive graph exploration. *Poster in the IEEE Symposium on Information Visualization*, 2005.

[23] P. C. Wong, H. Foote, G. C. Jr., P. Mackey, and K. Perrine. Graph signatures for visual analytics. *IEEE Transactions on Visualization and Computer Graphics*, 12(6):1399–1413, 2006.
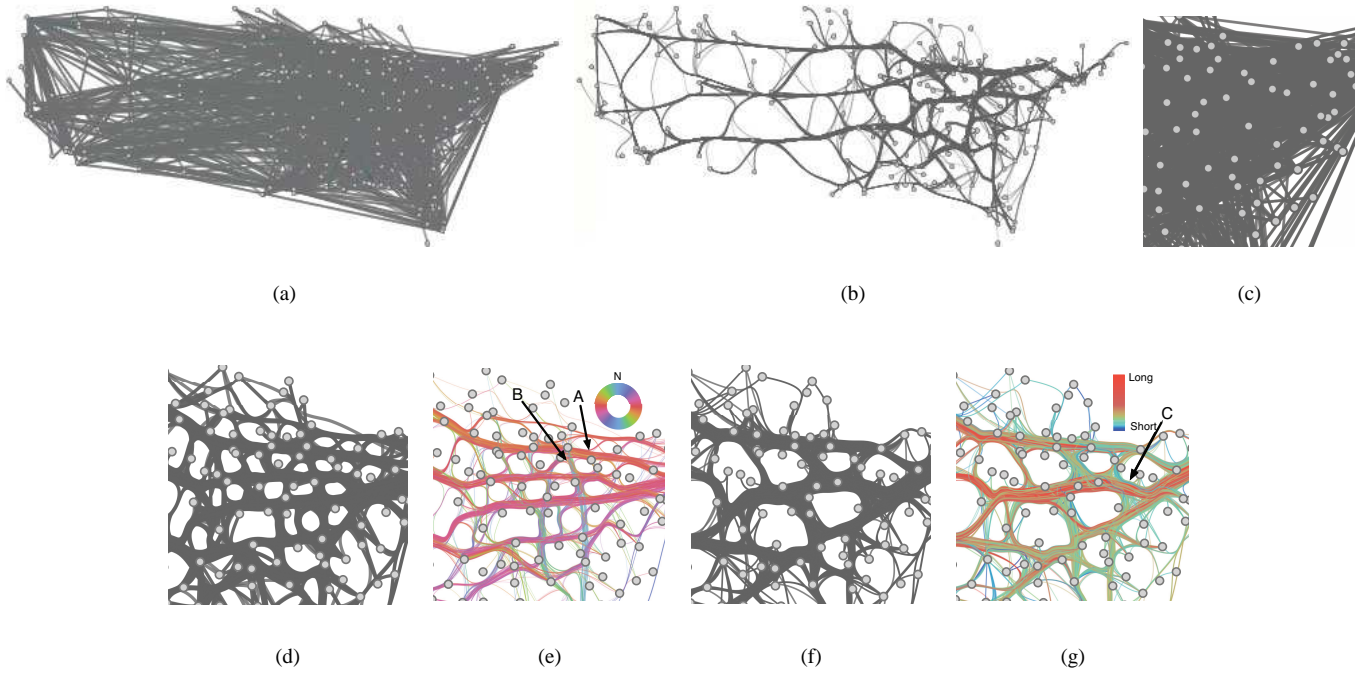
Fig. 13. Airline routes with 235 nodes and 2101 edges: (a) original layout; (b) our layout; (c) the original layout after zooming into the northeast region; (d)(f) our layout with two different control meshes; (e)(g) our result after color and opacity enhancement.
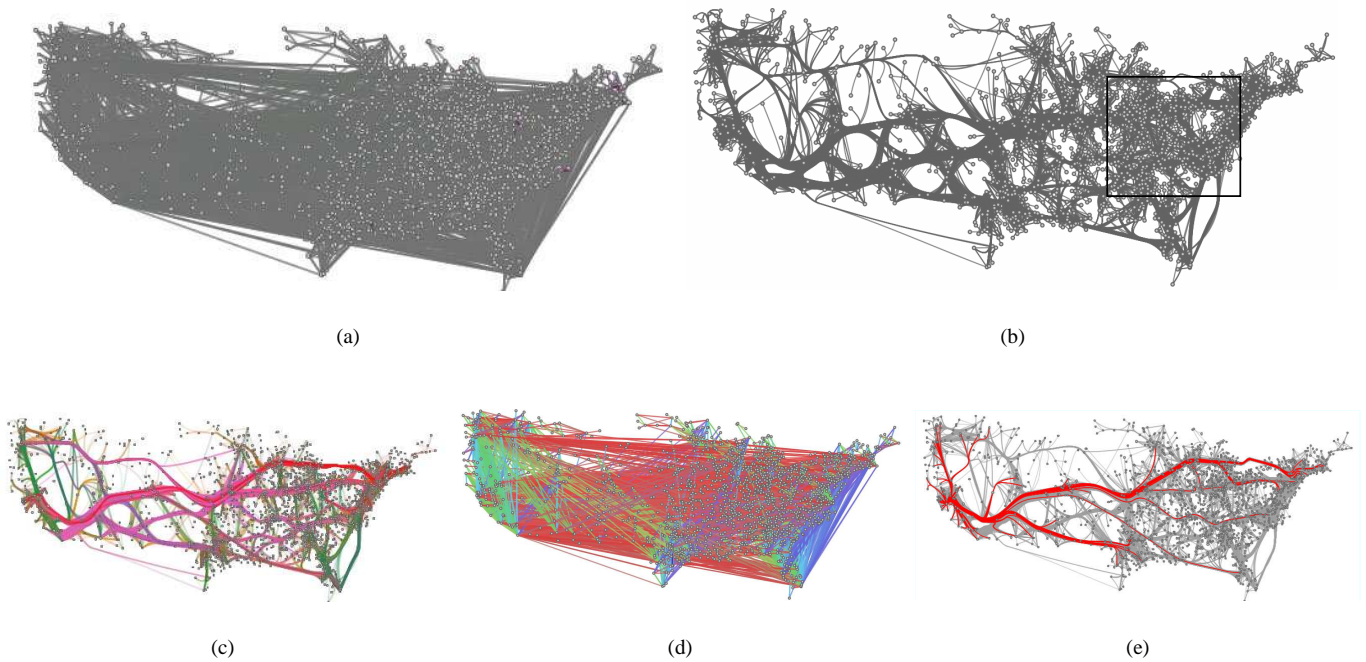

Fig. 14. U.S. immigration graph with 1790 nodes and 9798 edges: (a) original layout; (b) the edge-clustered result; (c) the result after applying edge clustering and transfer function; (d) the result after applying only transfer function; (e) a flow map layout highlighted in orange color.