DEPLOYING, IMPROVING AND EVALUATING EDGE BUNDLING METHODS FOR VISUALIZING LARGE GRAPHS

Jieting Wu
University of Nebraska - Lincoln, jtwu1986@gmail.com

Follow this and additional works at: http://digitalcommons.unl.edu/computerscidiss

Part of the Computer Engineering Commons, and the Computer Sciences Commons

http://digitalcommons.unl.edu/computerscidiss/164
DEPLOYING, IMPROVING AND EVALUATING EDGE BUNDLING METHODS FOR VISUALIZING LARGE GRAPHS

by

Jieting Wu

A DISSERTATION

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfilment of Requirements

For the Degree of Doctor of Philosophy

Major: Computer Science

Under the Supervision of Professor Hongfeng Yu

Lincoln, Nebraska

November, 2018
A tremendous increase in the scale of graphs has been witnessed in a wide range of fields, which demands efficient and effective visualization techniques to assist users in better understandings of large graphs. Conventional node-link diagrams are often used to visualize graphs, whereas excessive edge crossings can easily incur severe visual clutter in the node-link diagram of a large graph. Edge bundling can effectively remedy visual clutter and reveal high-level graph structures. Although significant efforts have been devoted to developing edge bundling, three challenging problems remain. First, edge bundling techniques are often computationally expensive and are not easy to deploy for web-based applications. The state-of-the-art edge bundling methods often require special system supports and techniques such as high-end GPU acceleration for large graphs, which makes these methods less portable, especially for ubiquitous mobile devices. Second, the quantitative quality of edge bundling results is barely assessed in the literature. Currently, the comparison of edge bundling mainly focuses on computational performance and perceptual results. Third, although the family of edge bundling techniques has a rich set of bundling layout, there is a lack of a generic method to generate different styles of edge bundling.

In this research, I aim to address these problems and have made the following contributions. First, I provide an efficient framework to deploy edge bundling for web-based platforms by exploiting standard graphics hardware functions and libraries. My framework can generate high-quality edge bundling results on web-based platforms, and achieve a
speedup of 50× compared to the previous state-of-the-art edge bundling method on a graph with half of a million edges. Second, I propose a new moving least squares based approach to lower the algorithm complexity of edge bundling. In addition, my approach can generate better bundling results compared to other methods based on a quality metric. Third, I provide an information-theoretic metric to evaluate the edge bundling methods. I leverage information theory in this metric. With my information-theoretic metric, domain users can choose appropriate edge bundling methods with proper parameters for their applications. Last but not least, I present a deep learning framework for edge bundling visualizations. Through a training process that learns the results of a specific edge bundling method, my deep learning framework can infer the final layout of the edge bundling method. My deep learning framework is a generic framework that can generate the corresponding results of different edge bundling methods.
ACKNOWLEDGMENTS

Professor Hongfeng Yu, my advisor, is on top of my acknowledgement list. Through his support, guidance, feedback and critique throughout my Ph.D. study, I have achieved many of my goals and also developed a broad set of skills ranging from critical thinking, programming to academic writing.

I would also like to thank Professor David Swanson, Professor Ying Lu, and Professor Zhenghong Tang for their time, guidance, patience, and insights on my Ph.D. study as committee members.

I would like to appreciate the valuable comments and guidance from my colleagues: Jianping Zeng, Lina Yu, Shruti Daggumati, Xin Liu, Feiyu Zhu, Yu Pan, Jin Wang, and Tian Gao.

I would also like to thank all my friends and my family for their encouragement and support whenever I faced challenges.

At the last, I would like to thank the one who I love. This love goes beyond words, distance, and time.

This dissertation has been sponsored by the National Science Foundation through grants IIS-1652846, IIS-1423487, and ICER-1541043.
## Contents

<table>
<thead>
<tr>
<th>Contents</th>
<th>ix</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Figures</td>
<td>xi</td>
</tr>
<tr>
<td>List of Tables</td>
<td>xi</td>
</tr>
</tbody>
</table>

### 1 Introduction

1.1 Motivation .......................................................... 1
1.2 Challenges .......................................................... 2
1.3 Contributions ......................................................... 4

### 2 Background

2.1 Basic Concept .......................................................... 6
   2.1.1 Definition of Edge Bundling ...................................... 7
   2.1.2 Pipeline of Edge Bundling ....................................... 7
2.2 Related Work .......................................................... 9

### 3 A Web-Based Framework for Edge Bundling

3.1 Introduction .......................................................... 13
3.2 Related Work .......................................................... 16
3.3 Textured-Based Edge Bundling ....................................... 17
### 3.3 Background

- **3.3.1 Background** 
- **3.3.2 Parallelization Strategy** 
- **3.3.3 Implementation**
  - **3.3.3.1 Concept**
  - **3.3.3.2 Algorithm**
  - **3.3.3.3 Rendering**

### 3.4 Pixel-Based Edge Bundling

- **3.4.1 Kernel Density Estimation Edge Bundling**
- **3.4.2 Implementation**

### 3.5 Results

- **3.5.1 Visualization**
- **3.5.2 Performance Evaluation**

### 3.6 Conclusion

### 4 Edge Bundling using Moving Least Squares Approximation

#### 4.1 Introduction

#### 4.2 Related Work

#### 4.3 Our Method

- **4.3.1 Background**
- **4.3.2 Sampling**
- **4.3.3 Moving Least Squares Approximation**

#### 4.4 Implementation

#### 4.5 Results

- **4.5.1 Visualization and Performance Results**
- **4.5.2 Quality Assessment of Bundled Graphs**

#### 4.6 Conclusion
5 An Information-Theoretic Framework for Evaluating Edge Bundling Visualization

5.1 Introduction ................................................................. 60
5.2 Related Work ................................................................. 61
  5.2.1 Graph Visualization and Evaluation Metrics ......................... 61
  5.2.2 Studies of Information Theory in Visualization and Computer Graphics 62
5.3 Our Method ................................................................. 63
  5.3.1 Background ............................................................... 63
  5.3.2 Uncertainty in Edge Bundling Visualizations .......................... 65
  5.3.3 An Information-Theoretic Metric for Edge Bundling Visualizations 69
  5.3.4 Algorithms and Implementation ....................................... 75
5.4 Application Examples ...................................................... 85
  5.4.1 Revisit FDEB, FFTEB, and MLSEB ................................. 87
    5.4.1.1 FDEB ............................................................. 87
    5.4.1.2 FFTEB ............................................................ 87
    5.4.1.3 MLSEB ........................................................... 88
  5.4.2 Heuristic Study .......................................................... 88
  5.4.3 Comparison I ............................................................ 91
  5.4.4 Comparison II ........................................................... 96
5.5 Conclusion ................................................................. 100

6 A Deep Learning Framework for Edge Bundling

6.1 Introduction ................................................................. 102
6.2 Related Work ................................................................. 104
6.3 Problem Formalization ...................................................... 106
  6.3.1 Model Description ...................................................... 107
6.4 Synthetic Dataset .............................................................. 110
6.5 Training and Testing .......................................................... 111
6.6 Results .............................................................................. 113
6.7 Conclusion ......................................................................... 115

7 Conclusion and Future Work .................................................... 117

Bibliography ............................................................................. 120
List of Figures

2.1 A general pipeline of edge bundling. ............................................. 8
2.2 Visualizing an airlines dataset with the state-of-art methods. .......... 12

3.1 Edge subdivision and subdivision point movement in FDEB. ........... 18
3.2 2D matrix representation of graph used in our parallel FDEB strategy. 20
3.3 Visualizing an airlines dataset with node-like diagram, FDEB, MINGLE, TBEB, and PBEB. ......................................................... 38
3.4 Visualizing US migrations with node-like diagram and PBEB. ........... 39

4.1 Two steps of our bundling operator. .............................................. 49
4.2 An example with two different iterations. ..................................... 51
4.3 Visualizing US airlines and US migrations with FDEB, FFTEB, and MLSEB. ................................................................. 54
4.4 Visualizing France airlines with FDEB, FFTEB, and MLSEB. .......... 55
4.5 Comparison of FFTEB and MLSEB using a large US migrations dataset. 55

5.1 A general visualization pipeline. ............................................... 64
5.2 A Venn diagram showing the relation of entropies $H(U)$ and $H(O)$, and mutual information $I(U;O)$. ........................................... 65
5.3 Comparison to find whether or not Chicago and Miami are connected using different methods. ...................................................... 66
5.4 An example illustrating uncertainty in edge bundling visualizations. 69
5.5 Graph drawings of two simple graphs to illustrate multiple possible paths between two nodes. 74
5.6 An illustration of finding number of paths between a pair of vertices in a drawing. 80
5.7 An example to approximate the number of paths between a source pixel and a target pixel. 82
5.8 Visualizations of different datasets with different methods. 89
5.9 FDEB drawings of U.S. airlines using different compatibility threshold values. 93
5.10 FFTEB drawings of U.S. airlines using different kernel threshold values. 94
5.11 MLSEB drawings of U.S. airlines using different compact support values. 95
5.12 The bivariate analysis of FDEB, FFTEB, and MLSEB using different parameters. 96
5.13 The bivariate analysis of FDEB, FFTEB, and MLSEB using different datasets. 99

6.1 The pipeline of our CNN framework for edge bundling. 107
6.3 Visualization results of US migrations using CNN-bundling and CUBu. 112
6.5 Visualization results of France airlines using CNN-bundling, and MLSEB. 115
List of Tables

3.1 Performance comparison using the US airlines, France airlines, and US migrations graph on a desktop. .................................................. 41

3.2 Performance comparison using the US airlines, France airlines, and US migrations graph on a tablet. .................................................. 42

4.1 Performance comparison. .................................................. 57

4.2 The quality comparison using the US migrations graph. ............. 58

5.1 A heuristic study to choose color and angle thresholds. .............. 91

6.1 Comparison between CUBu and CNN-bundling ......................... 114

6.2 Comparison between MLSEB and CNN-bundling ....................... 115
Chapter 1

Introduction

1.1 Motivation

Graphs are universally used to depict data relationships arising from applications such as taxonomy, social science, bioinformatics, and network engineering. To understand the immense amount of information of a graph, visualization is helpful to reveal significant patterns and subtle details. Traditionally, a graph is often visualized as a presentation of vertices and edges, known as a node-link diagram, because of its simplicity and intuitiveness [43, 112]. As the quality and the quantity of data from scientific and engineering applications increase rapidly, the scale of graphs grows accordingly. Traditional exploration of a node-link diagram is overwhelmed by severe visual clutter such as excessive vertex overlapping and edge crossing. Researchers and visualization designers have developed many methods to simplify the visualization of node-link diagram. To address visual clutter of a node-link diagram, a new concept, named edge bundling, has been proposed. Holten’s Hierarchical Edge Bundling (HEB) [44] pioneered the idea and applied edge bundling to graphs with hierarchical structures. Since then, edge bundling methods have evolved rapidly to adapt for general static and dynamic graphs. Edge bundling is now one of the
most effective techniques to reveal salient graph structures and patterns, as well as similar communities and subtle outliers for visualizing large graphs.

Edge bundling provides a simplification of a node-link diagram that is easier for comprehension. The basic idea of edge bundling is to visually group similar edges, such that edge crossings are visually reduced, and main graph structure is revealed. Edge bundling has been used in various scientific and engineering applications, such as life science [11, 4, 29], software engineering [19, 24, 95], social science [52, 50], and so on. Applications of edge bundling have significant impacts on diverse fields.

1.2 Challenges

Edge bundling is very effective to visualize real-world large graphs, but it requires non-trivial computational steps to provide a reliable result. Generally, most edge bundling methods first measure the similarity, namely, compatibility to identify which edges are compatible to be bundled. Existing methods use different compatibility metrics and advect the edges based on the compatibility to generate different visual bundle results. However, the resulting graph drawings can often contain certain visual artifacts. Expensive post-processing steps, such as resampling and smoothing, are used to alleviate the visual artifacts. The post-processing steps negatively impact the performance of edge bundling applications. To simplify these computational steps and generate effective visual results remains a big challenge in developing edge bundling algorithms.

Although web-based platforms have gradually become front-ends for informatic visualization tools, to deploy edge bundling applications for web access is a challenging task. Edge bundling techniques have expensive computational steps, which can be possibly accelerated using hardware and parallel processing libraries, such as CUDA and OpenCL. However, the support of such parallel tools on web-based platforms is limited. This prevents edge
bundling applications from portable usages. Moreover, efficiency plays an important role in edge bundling. An efficient and interactive edge bundling application can help users quickly identify data patterns in dynamic graphs [48]. Hence, to facilitate an efficient edge bundling framework for web access is vital but still an open problem.

Although a rich family of edge bundling has been supporting the data analysis for many scientific and applicable domains, there is a lack of a quantitative quality assessment to evaluate edge bundling methods in the literature. So far, the comparisons between different edge bundling methods mostly focused on computational performance and perceptual results. Few studies have evaluated the quantitative quality of edge bundling methods. Domain experts can benefit from a quantitative evaluation to choose effective edge bundling methods for their applications, and users can select the optimal parameters for an edge bundling method from the evaluation feedback. Developers can also leverage a quantitative evaluation to improve edge bundling methods. A quantitative quality assessment is thereby critical for the development of edge bundling.

In addition, although the family of edge bundling methods provides a wide spectrum of edge bundling layouts, a style (layout) framework is still missing for edge bundling methods. On one hand, few unified frameworks of general graph visualizations have been presented. On the other hand, to develop a unified framework for generating the layouts of edge bundling methods is very challenging since different methods typically employ different statistical or computational models. To mimic the computational processes of different models using a general framework is formidably difficult. The lack of the style (layout) tractability of edge bundling methods has hindered the progress towards the layout analysis and the deployment efficiency of edge bundling applications.
1.3 Contributions

My research in this field leads to the following four main contributions:

First, I tackle edge bundling by exploiting standard graphics hardware functions and libraries, and present a novel framework that enables edge bundling on web-based platforms in Chapter 3. I adopt a force-directed edge bundling method and an image-based edge bundling method in my framework, combining a parallel texture-based processing scheme to accelerate the bundling process and enable interactivity on web-based platforms. In this framework, I take advantage of texture-based parallel processing, which is a standard feature of WebGL. My framework optimizes an end-to-end (from bundling to rendering) process that enables an effective and interactive large graph visualization on web-based platforms. My web-based implementation achieves $50 \times$ speedups compared to the fastest CUDA-based edge bundling method on a large graph with half a million edges. I also demonstrate the performance of my framework by comparing the existing web-based and CUDA-based edge-bundling methods using different standard graphics cards on different devices.

Second, I further lower the complexity of edge bundling from an algorithmic perspective, and introduce a new edge bundling method that leverages a Moving Least Squares (MLS) approximation to simplify the computational steps of edge bundling methods in Chapter 4. My approach generates bundle effects by iteratively projecting each sample point of an edge to its local regression curve based on its neighborhood’s density. Such a local regression method can produce a better convergence result using fewer iterations. Complex processing steps, such as resampling and smoothing, are thus avoided. Compared to other edge bundling methods, my method provides a better quality based on a preliminary quality metric. The performance result also shows that my CUDA-based implementation is at the same order of magnitude of the fastest edge bundling method in terms of computational speed.
Third, I propose an information-theoretic framework to evaluate the visual results of
the edge bundling techniques in Chapter 5. I first illustrate the advantage of edge bundling
visualizations for large graphs, and pinpoint the ambiguity resulting from drawing results.
Second, we define and quantify the amount of information delivered by edge bundling
visualization from the underlying network using information theory. Third, we propose
a new algorithm to evaluate the resulting layouts of edge bundling using the amount of
the mutual information between a raw network dataset and its edge bundling visualization.
Comparison examples based on the proposed framework between different edge bundling
techniques are presented.

Fourth, I present a generic deep learning framework to generate different styles of edge
bundles in Chapter 6. Our deep learning technique can mimic the styles and layouts of a
specific edge bundling. The deep learning framework mainly uses a Convolutional Neural
Network (CNN) architecture to learn and mimic the local information estimation of edge
bundling methods. Using the trained feature vector from the results of a specific edge
bundling method, we can infer the final layout result of the corresponding edge bundling
method. Our framework is to inspire researchers to investigate how to use deep learning
techniques to generate effective graph visualizations such that more mature edge bundling
visualizations can be pushed forward.
Chapter 2

Background

2.1 Basic Concept

Edge bundling methods can be categorized into explicit methods and implicit methods [64]. Explicit methods bundle edges based on a control structure. The typical examples are Hierarchical Edge Bundling (HEB) [44] and Geometric-Based Edge Bundling (GBEB) [21]. HEB bundles edges based on a predefined hierarchical structure. The advantage is efficient and predictable, but knowing a hierarchical structure in advance from real-world datasets is less practical. GBEB computes a control structure only once and bundles edges based on the control structure. It can reduce visual clutter, whereas the disadvantage is obvious: the bundles lack flexibility in terms of geometry shape and the generation of the control structure is expensive. Implicit methods work in a self-organizing way. The idea is to iteratively compute a local optimal layout to self-adjust. Bundles are gradually constructed based on the local optimal layout. This idea has dominated in developing edge bundling since it does not require a known hierarchical structure, and the bundles are more flexible. In Section 2.1.1, I will revisit the formal definition and the basic pipeline of edge bundling. In Section 2.2, I will discuss the state-of-the-art edge bundling methods.
2.1.1 Definition of Edge Bundling

We first revisit a formal definition of edge bundling [64]. Let $G = (V, E) \subset \mathbb{R}^2$, $V = \{v_i\}$, $E = \{e_i\}$ be a graph, where $v_i$ is a vertex and $e_i$ is an edge of $G$. Let $D : E \rightarrow \mathbb{R}^2$ be a drawing operator, such that $D(G)$ represents the drawing of $G$ and $D(e_i)$ represents the drawing of an edge $e_i$. We define a compatibility operator $\kappa$, where $\kappa(e_i, e_j)$ measures the similarity of two edges $e_i$ and $e_j$. Edges that are more similar than a threshold $\kappa_{\text{max}}$ should be bundled together, and $\kappa$ can be used with some reasonable attributes and metrics (e.g., spatial information [45]). Let $B : \mathcal{D} \rightarrow \mathcal{D}$ be a bundling operation, where $\mathcal{D} \subset \mathbb{R}^2$ denotes the space of all graph drawings, and $B(D(e_i))$ denotes the resulting bundled drawing of $e_i$. For example, $D(e_i)$ can be a straight line drawing and $B(D(e_i))$ can be a drawing of curve or polyline. Hence, an edge bundling algorithm can be expressed as:

$$\forall (e_i \in G, e_j \in G) | \kappa(e_i, e_j) < \kappa_{\text{max}} \rightarrow$$

$$\delta(B(D(e_i)), B(D(e_j))) \ll \delta(D(e_i), D(e_j)),$$

where $\delta$ is a distance metric in $\mathbb{R}^2$. Different edge bundling approaches have explored various $\kappa$, $B$, and $\delta$ in Equation 2.1 to gain different visual effects of edge bundling [64].

Next, we briefly introduce a general pipeline of edge bundling, and explain their work using Equation 2.1.

2.1.2 Pipeline of Edge Bundling

Figure 2.1 shows a general pipeline of graph visualization using edge bundling. Given a node-link diagram, edges are first discretized into sample points, namely, sites. Second, the similarity of edges is computed to distinguish which edges should be bundled. Here, similarity can be a compatibility measurement (spacial information of edges) or a predefined hierarchical structure. Third, sites are advected to form bundles based on specific criteria.
Most of the edge bundling methods construct a set of smoothly curved bundles by using 1-D Laplacian smoothing, Bézier or B spline. Finally, the generated geometry data of bundles are sent to GPU for rendering.

This framework typically has several problems for large graphs: As mentioned above, the explicit methods either compute a control structure once, which can be expensive, or construct bundles based on a known structure, which can be efficient but less practical to real-world datasets and be short of flexibility. Implicit methods require iterative refinement processing to generate reliable results, which is expensive because resampling, compatibility, and advection are iteratively computed. Iterative edge bundling requires considerable computing power for a large number of edges. Moreover, if we conduct edge bundling using a GPU, transferring the data of bundles from the main memory to the GPU memory is costly for a large graph. For an interactive graph visualization, edge bundling and rendering need to be carried out frequently in response to each user interaction. Thus, these problems can be severe in the application of large graphs that require real-time efficiency and interactivity.


2.2 Related Work

This research focuses on addressing the three challenging problems for the development of edge bundling. I refer those who are interested in information visualization to the following work and surveys. Liu et al. [69] presented a comprehensive survey and key insights into the fast-rising information visualization domain. Landesberger et al. [114] and Beck et al. [8] surveyed many techniques for large graphs and dynamic graphs respectively. Herman et al. [43] and Vehlow et al. [112] conducted comprehensive surveys for graph visualization. In this section, I will first recap some of the state-of-the-art methods that have significantly impacted the development of edge bundling. For a comprehensive survey of edge bundling methods, I redirect readers to the edge bundling survey [64]. I will briefly discuss the current web-based tools and studies that also adopt node-link diagram visualization for graph visualizations in Chapter 3. Since I propose a novel edge bundling algorithm with Moving Least Squares approximation, I will cover some studies and applications on Moving Least Squares approximation in Chapter 4. Next, I will also recap some studies of evaluation for graph drawing in Chapter 5. Lastly, in Chapter 6, I will revisit some existing work using convolutional neural network (CNN) based deep learning techniques in the literature.

Holten [44] pioneered the edge bundling techniques in graph drawing. He introduced a novel edge clutter reduction technique for a graph that has a hierarchical structure. The edges are bundled using B-splines, of which the control points are defined by the hierarchical structure. It is computationally efficient, but a hierarchical structure must be known in advance. The following work [120] also employed hierarchical structures as a guideline to organize edge bundles.

Energy-based minimization methods are used in many studies. Gansner et al. [36] applied a global ink minimization into edge bundling for the circular layout of graphs. Multilevel Agglomerative Edge Bundling (MINGLE, Figure 2.2.(b)) [35] used the same
principal idea of the global ink minimization, and leveraged a kd-tree implementation to
calculate compatibility measures to distinguish the bundled edges. Their quality metric was
based on an ink minimization.

Zhou [121] employed a force model to visualize high-dimensional and multivariate
datasets in parallel coordinates. Holten et al. [45] further developed the force model in
general graphs, namely, *Force-Directed Edge Bundling* (FDEB, Figure 2.2.(a)). Similar
methods and applications [82, 99, 123] have followed the same basic idea.

Geometric grids and meshes have been also employed to retrieve local geometric infor-
mation of graphs for generating edge bundles. Cui et al. [21] proposed *Geometric-based
Edge Bundling* (GBEB, Figure 2.2.(c)). They used a pre-computed geometric mesh to guide
the bundling process. The mesh works as an analyzer for detecting the directions and the
distributions of edges. According to the statistic information gained from the mesh, control
points are generated to cluster the edges. Similarly, Lambert et al. [59] proposed *Winding
Roads* (WR, Figure 2.2.(d)) that also discretized the input graph into multiple regions using
geometric grids. They used a hybrid quad-tree and Voronoi approach to generate supervising
grids. Then, they routed the edges in a Bézier spline manner. The work [58] extended
the method of WR in 3D geographical layouts. Luo et al. [70] posed a problem that edge
bundling may cause ambiguous interpretations of graphs. They proposed a space-detection
metric to produce control meshes with a kd-tree for edge bundle generation.

Image-based techniques used a density assessment to guide bundling processes [11, 28,
47, 63, 84, 110]. A skeleton-based edge bundling is given in SBEB [28]. They combine
distance field and an edge clustering method to create a 2D skeleton that is suitable for
converging the edges. The following methods are generally based on *Kernel Density
Estimation*. Kernel density estimation edge bundling (KDEEB, Figure 2.2.(f)) [47] first
introduced this idea. The basic pipeline is to first transform an input graph into a density map
using kernel density estimation, and then moved the sample points of edges towards the local
density maxima to form bundles. Peysakhovich et al. [84] extended KDEEB using edge attributes to distinguish bundles. CUDA Universal Bundling (CUBu, Figure 2.2.(g)) [110] used GPU acceleration to enable interactively bundling a graph with a million edges. Fast Fourier Transform Edge Bundling (FFTEB, Figure 2.2.(h)) [63] improved the scalability of density estimation by transforming the density space to the frequency space.

There are other edge bundling studies. Bach et al. [5] investigated the connectivity of edge bundling methods on Confluent Drawings. Nguyen et al. [79] proposed an edge bundling method for streaming graphs, which extended the idea of TGI-EB [82]. Kwon et al. [57] showed their layout, rendering, and interaction methods for edge bundling in an immersive environment.
Figure 2.2: Visualizing an airlines dataset with 2180 edges (a-b) and a US migration dataset with 9780 edges (c-h) using the state-of-art methods.
Chapter 3

A Web-Based Framework for Edge Bundling

3.1 Introduction

Web-based information visualization is becoming increasingly ubiquitous. Web-based tools and techniques have provided a portable means for designers to build visualization across different platforms. Representative examples include D³ [9] and Protovis [10]. However, none of these methods or applications can facilitate efficient edge bundling for large graphs on web-based platforms. The datasets used in the edge bundling applications of D³ and Protovis are modest, and it is difficult to achieve interactive framerates in the resulting visualizations. As the size of datasets increase drastically in both scientific and applicable domains, it is becoming more and more challenging to effectively and efficiently bundle a large graph on web-based platforms. Edge bundling methods can concisely reveal the main structure of a graph by grouping related edges together into a set of smoothly curved bundles. However, as mentioned in Section 2.1.2, their execution typically involves expensive computational steps to achieve a visually appealing result for a large graph.
Although researchers have experimented with acceleration methods [47, 63, 84, 110, 122], these solutions mostly rely on features dedicated to special graphics hardware or libraries. The support of these standard parallel programming libraries and architectures, such as CUDA and OpenCL, is limited in web-based platforms. Although some lightweight libraries, such as WebCL, emerged in recent years, they are still limited in certain smart devices and web browsers. This is a critical problem because the complicated computational steps of edge bundling methods demand parallel acceleration. Therefore, it remains a challenging task to efficiently construct and visualize edge bundles of large graphs on devices with a limited computational capacity and library support, in particular, on ubiquitous smart mobile devices.

To address the challenge, I present a new framework for constructing and visualizing edge bundles of large graphs using standard functions of graphics cards for web access. My method adopts a force-based model and a kernel density based model. The reason why I choose to focus on these two models is that the force-based model is one of the most popular graph models that have been widely used in diverse fields while kernel density based model is very effective to bundle very large graphs. In particular, for the force-based model, I employ the computation model of Force-Directed Edge Bundling (FDEB) [45], encode the data of lines and forces into standard texture images, and conduct iterative line refinement using vertex and fragment shaders that are the standard functions of graphics systems. As the major steps are conducted through the usage of textures, I name this method as Texture-based Edge Bundling (TBEB). For the kernel density based model, I exploit the basic idea from Kernel Density Estimation Edge Bundling (KDEEB) [47] and its optimized implementation using CUDA [110]. They essentially use the mean shift principle [18]. The idea is to iteratively advect the edges by using the normalized gradient of an edge density map. The edge density map can be encoded as a histogram into a texture. Every pixel in the texture gives an accumulation value, and then a density gradient can be calculated to advect
edges. Iterative advection uses programs of vertex and fragment shaders that is the same as TBEB. Since the histogram and gradient calculation of this model is conducted on the pixel level, I name this method as Pixel-Based Edge Bundling. To enable real-time interaction of the result, I optimize the data transfer in implementation to enable an efficient framework for force-based and kernel density based edge bundling methods on web-based platforms.

The parallel processing of my method uses the standard graphics hardware functions and libraries, and can be universally applied on devices with standard graphics processing units. In addition, I present a web-based framework that can be easily migrated across devices and platforms without any significant overhead. I demonstrate the effectiveness of my framework using multiple data sets on devices with standard graphics cards. My framework can achieve interactive frame rates to conduct and render the edge bundles of a graph with thousands of edges in a web browser. More importantly, my proposed framework outperforms the state-of-the-art edge bundling techniques in terms of real-time execution.

Hence, my contribution in this chapter is twofold. First, I develop a web-based framework to effectively visualize large graphs ranging from thousands of edges to half of a million edges using an edge bundling method. Our visualization results can reveal the high-level structures and backbones of the graphs on web-based platforms. Second, we address the parallel processing issue of edge bundling using a web-based method and minimize the data movement between CPU and GPU to optimize the performance of the real-time interaction. In our experiments, we show that our web-based framework outperforms the current fastest method [63] by a speedup of $50\times$ on a graph with half of a million edges. At the same time, we ensure the high quality of our visualization result on web-based platforms.
3.2 Related Work

In this chapter, I will address how to deploy efficient edge bundling implementations on portable devices. Hence, I will introduce existing approaches that develop visualizations using portable media, and review some acceleration methods of edge bundling methods. Web-based visualization systems have been providing a portable means for designers and researchers to build customizable visualization system. Protovis [9] was a visualization system that employed HTML, JavaScript, and Scalable Vector Graphics (SVG) to generate interactive web-based visualization. A web-based library Document-Driven Documents ($D^3$) [10] emerged as a new visualization system to directly manipulate the elements of page content by binding data to document elements. The performance of $D^3$ was demonstrated to be twice as fast as Protovis. However, when the problem size reaches a certain large amount, e.g. visualizing thousands of edges with force-directed layouts (force-directed node-link diagram and force-directed edge bundling graph) [22], the performance of the interactive visualization of $D^3$ drops drastically. A web-based implementation [77] embedded MINGLE [35] in graph visualization for web access. But it is not efficient for bundling a large graph with tens of thousands of edges.

Although these edge bundling based approaches can achieve visually appealing results, they are characterized by high algorithmic complexities. The execution of such an algorithm may take several to hundreds of seconds to compute bundles for large graphs. This performance lag hinders the usage of these methods on interactive applications. A few efforts have been perceived to accelerate the construction of edge bundles. Ersoy et al. [28] used GPUs to accelerate their skeleton-based edge bundling. Zhu et al. [122] presented a parallelized FDEB on GPUs, and achieved an $11 \times$ speedup compared to the original FDEB. However, these methods rely on NVIDIA’s CUDA architecture, which cannot be easily adopted to general graphics cards.
3.3 Textured-Based Edge Bundling

In this section, I aim to accelerate force-directed edge bundling (FDEB) [45] using standard functions of general graphics cards, and make edge bundling deployable and compatible with web-based platforms.

3.3.1 Background

We first revisit the computational model of FDEB. As shown in Figure 2.1, FDEB takes a node-link diagram with straight edges as its input, and uses an iterative simulation to refine the bundling. There are total $C$ simulation cycles. In each cycle, FDEB first subdivides each edge and then iteratively moves each subdivision point to a new position by modeling and computing forces among the points.

In the first simulation cycle $C_0$, FDEB starts with $R_0$ subdivision points for each edge. For example, in Figure 3.1(a), each edge has two endpoints (in orange) and one subdivision point (in blue), and is subdivided into two segments (i.e., $R_0 = 1$ in this case). The subdivision is essentially a resampling process. Then, two types of forces, the spring forces $F_s$ and the electrostatic forces $F_e$, are modeled at a subdivision point $p_{ij}$, where $p_{ij}$ is the $j$th point on an edge $l_i$, as shown in Figure 3.1(a). $F_s$ is defined as:

$$F_s = k_p (\| p_{i(j-1)} - p_{ij} \| + \| p_{ij} - p_{i(j+1)} \|),$$  \hspace{1cm} (3.1)

where $k_p$ is a spring constant, and $p_{i(j-1)}$ and $p_{i(j+1)}$ are the neighboring points of $p_{ij}$ on the edge $l_i$. $F_e$ is defined as:

$$F_e = \sum_{m\in E} \frac{1}{\| p_{ij} - p_{mj} \|},$$  \hspace{1cm} (3.2)

where $E$ is a set of edges where each edge $l_m$ interacts with $l_i$. $p_{mj}$ is the corresponding subdivision point on such an interacting edge $l_m$. Thus, the combined force $F_{p_{ij}}$ exerted on
Figure 3.1: Edge subdivision and subdivision point movement in FDEB.

\( p_{ij} \) is:

\[
F_{p_{ij}} = F_s + F_e. \tag{3.3}
\]

The position of \( p_{ij} \) is updated by moving it a small distance in the direction of \( F_{p_{ij}} \). This is an iterative process in that \( F_e, F_s, \) and \( F_{p_{ij}} \) are also updated according to the new position of \( p_{ij} \). A specific number of iteration steps \( I \) is conducted to move the subdivision points to reach an equilibrium between forces. Figure 3.1(b) shows the equilibrium state of \( C_0 \).

The edge subdivision and the subdivision point movement are continued in the following simulation cycles. Figure 3.1(c) and (d) illustrate the process of \( C_1 \).

The number of iteration steps during the first cycle is \( I_0 \). We can easily see that the number of subdivision points is doubled to smoothen the edges after performing a cycle.
Meanwhile, the number of iteration steps $I$ is decreased by a factor $R$. The original paper of FDEB [45] reported that a configuration of $P_0 = 1$, $C = 6$, $I_0 = 50$, and $R = 2/3$ leads to appropriate results.

The size of $E$ can be significant for a large graph, and thus increase the computing cost. To address this, FDEB uses four criteria, angle, scale, position, and visibility, for edge compatibility measures. Only the edges with the compatibility measures greater than a threshold are considered as interacting edges. In this way, FDEB can control the amount of interaction between edges, and thus reduce the computing overhead.

### 3.3.2 Parallelization Strategy

We characterize FDEB and observe several possibilities to parallelize the computation with respect to simulation cycles and iteration steps:

- We can see that each line can be subdivided independently in each simulation cycle. On the other hand, each line should have the same number of subdivision points to compute the combined force. This implies that the line subdivision needs to be synchronized in each simulation cycle. That is, asynchronous line subdivision across simulation cycles is not allowed.

- We can see that in each iteration step, the combined force of each subdivision point can be computed independently. Similarly, the position of points can be updated in parallel as well.

- Given a specific configuration of $P_0$, $C$, $I_0$, and $R$, the total amount of subdivision points of a graph is deterministic after the last simulation cycle of FDEB. This implies that we can pre-allocate the memory to accommodate the data of final result.

Based on these observations, we can derive the following parallelization strategy:
Figure 3.2: The 2D matrix representation of graph used in our parallel FDEB strategy.

First, we use a 2D matrix to represent the graph. Each row corresponds to one edge $l_i$, and each entry records the position of a point $p_{ij}$. The total matrix entry number is equal to the total point number that will be generated by FDEB in the final cycle. Initially, only the first two columns are filled, which correspond to the two endpoints of the original straight edges. We then insert the subdivision points into the 2D matrix at each simulation cycle. For example, Figure 3.2 (a) and (b) show the matrixes of $C_0$ and $C_1$, respectively, where the orange entries correspond to the endpoints, and the blue entries correspond to the subdivision points at each cycle. Compared to Figure 3.2 (a), we can see that a new blue entry (i.e., a new subdivision point) has been inserted into each row (i.e., an edge) of the matrix in Figure 3.2 (b). This process will be continued until the whole matrix is filled.

Second, given the positions of all points, we can compute the combined force $F_{p_{ij}}$ of a point $p_{ij}$ using Equations 3.1, 3.2, and 3.3. The position value of $p_{ij}$ then is updated by moving it a small distance according to $F_{p_{ij}}$. The updated position is directly recorded into the entry of $p_{ij}$ in the 2D matrix.

Figure 3.2 illustrates this parallelization strategy. We can see that the computing task associated with $p_{ij}$ is largely independent of each other. Intuitively, we can assign each task of $p_{ij}$ to a thread, and execute them in parallel. Algorithm 1 shows a realization of our parallelization strategy\(^1\). Lines 12-18 correspond to the operation of edge subdivision,

\(^1\)To simplify the discussion, the technical detail of compatibility measure is not described, but its timing
Algorithm 1 PARALLELFDEB

1: // Initialization
2: $C \leftarrow$ the number of simulation cycles
3: $I_0 \leftarrow$ the number of iteration steps in the first cycle $C_0$
4: $R \leftarrow$ the decreasing factor
5: $t \leftarrow$ the number of points of each edge after the last cycle
6: Given $n$ edges, create one 2D matrix $M$ with $n$ rows and $t$ columns.
7: Fill the two endpoints of all edges into the first and third columns of $M$, respectively.
8: $C_i \leftarrow 0; I_i \leftarrow I_0$
9: // Simulation
10: while $C_i < C$ do
11:   // Subdivide each edge
12:     for each entry $p_{ij}$ of $M$ in parallel do
13:       if $p_{ij}$ is an endpoint at $C_i$ and $C_i > 0$ then
14:         Copy the corresponding endpoint at $C_{i-1}$ to $p_{ij}$.
15:       else
16:         Compute $p_{ij}$ as a subdivision point according to the previous subdivision points at $C_{i-1}$.
17:     end if
18:   end for
19:   Synchronization
20:   // Iteratively move the subdivision points
21:     for each iteration step of $I_i$ do
22:       for each entry $p_{ij}$ of $M$ in parallel do
23:         Compute $F_s$ with respect to $p_{ij}$.
24:         Compute $F_e$ with respect to $p_{ij}$.
25:         $F_{p_{ij}} \leftarrow F_s + F_e$.
26:         Move $p_{ij}$ a small distance in the direction of $F_{p_{ij}}$, and update $p_{ij}$ in $M$.
27:       end for
28:     end for
29:   Synchronization
30: $C_i \leftarrow C_i + 1$
31: $I_i \leftarrow I_i \times R$
32: end while

and Lines 21-29 correspond to the operations of force calculation and subdivision point movement. Special care has been taken to ensure the synchronization across the simulation cycles (Line 19) and the iteration steps (Line 28).

result is included in our performance evaluation.
Conceptually, Algorithm 1 can be easily implemented using multi-thread techniques on CPUs or GPUs (Line 12 and Line 22). For example, one implementation has been proposed using CUDA on NVIDIA graphics cards [122]. GPU-based implementations are especially attractive as a significant speedup can be possibly achieved by leveraging the massive parallelism of a GPU. However, these implementations often rely on specific graphics cards or graphics features, such as direct GPU memory write, that is not universally supported. In particular, mobile devices are often equipped with lower-end GPUs, and are less capable to carry out the calculation in Algorithm 1.

3.3.3 Implementation

We present a new method to realize the visualization framework in Figure 2.1 by holistically addressing edge bundling and rendering. Our method uses standard graphics features specified in OpenGL and WebGL, and thus can be easily adapted by various devices with standard GPUs. In particular, if we conduct our method using WebGL, the web page of interactive edge bundling can be displayed on most web browsers without any modification.

3.3.3.1 Concept

The main challenge for us to perform FDEB using OpenGL or WebGL is that there is a lack of functionality of direct GPU memory access in OpenGL or WebGL. When we use OpenGL or WebGL, GPU memory access is typically conducted through texture functions. For GPU memory read, we can first upload data from the main memory to the GPU memory via texture binding, and then access texture data via texture lookup in vertex or fragment shaders. For GPU memory write, we can first bind a texture with a Framebuffer Object (FBO) and render data into FBO. The rendering function can be customized using vertex or fragment shaders so that we can control the contents that are written to texture. Because
we cannot directly read and write one texture simultaneously, we can use the ping-pong buffering (or double buffering) method [85] to read the input and write the output through textures, and thus enable the operations of edge subdivision, force computing, and position updating.

### 3.3.3.2 Algorithm

The main algorithm of our method is illustrated in Algorithm 2. Given a specified configuration (i.e., \( P_0, C, I_0, \) and \( R \)), we can predict the number of points of each edge, \( t \), after the last cycle. Given a node-link diagram with \( n \) edges, we construct two 2D matrixes \( M_a \) and \( M_b \), and each matrix has \( n \) rows and \( t \) columns. Our goal is to fill each entry of one matrix with the final position of a subdivision point after the completeness of simulation.

Initially, we fill the two endpoints of all edges into the first and third columns of a matrix \( M_a \). We then bind \( M_a \) as the input texture \( T_{in} \), and bind \( M_b \) as the output FBO \( T_{out} \). The 2D or 3D position coordinates are encoded into the color components of the input texture \( M_a \). Then, during each simulation cycle, we use a fragment shader \text{TEXTUREBASED_SUBDIVISION} to conduct edge subdivision (Line 13 in Algorithm 2), and then iteratively call another fragment shader \text{TEXTUREBASED_UPDATE} (Line 17 in Algorithm 2) to compute the forces and update the point position.

Algorithm 3 lays out the \text{TEXTUREBASED_SUBDIVISION} fragment shader. Each pixel \( p_{ij} \) of \( T_{out} \) represents the \( j \)th point of an edge \( l_i \) at a simulation cycle \( C_i \). If \( p_{ij} \) is an endpoint at \( C_i \), we can directly fetch the corresponding endpoint \( p_{ij} \) in \( T_{in} \) using texture lookup and copy the value to \( p_{ij} \) in \( T_{out} \). Otherwise, we compute \( p_{ij} \) as a subdivision point according to the previous subdivision points in \( T_{in} \) using texture lookup. We then render \( p_{ij} \) as a color into \( T_{out} \). The key to this step is the transform between texture coordinates and the indexes of lines and points. When we encode a data set into a texture, a data point is accessed through a 2D texture coordinate \((x, y)\), where the \( x \) or \( y \) component of texture coordinate is
Algorithm 2 TEXTUREBASEDEDBUNDLING

1: // Initialization
2: $C \leftarrow$ the number of simulation cycles
3: $I_0 \leftarrow$ the number of iteration steps in the first cycle $C_0$
4: $R \leftarrow$ the decreasing factor
5: $t \leftarrow$ the number of points of each edge after the last cycle
6: Given $n$ edges, create two 2D matrixes $M_a$ and $M_b$. Each matrix has $n$ rows and $t$ columns.
7: Fill the two endpoints of all edges into the first and third columns of $M_a$, respectively.
8: $T_{in} \leftarrow M_a$ // Bind $M_a$ as the input texture $T_{in}$
9: $T_{out} \leftarrow M_b$ // Bind $M_b$ as the output FBO $T_{out}$
10: $C_i \leftarrow 0; I_i \leftarrow I_0$
11: // Simulation
12: while $C_i < C$ do
13:   TEXTUREBASEDSUBDIVISION($T_{in}$, $T_{out}$, $C_i$)
14:   Swap $T_{in}$ and $T_{out}$ (i.e., the previous $T_{out}$ becomes the input texture, and the previous $T_{in}$ becomes the output FBO).
15:   // Iteratively move the subdivision points
16:   for each iteration step of $I_i$ do
17:     TEXTUREBASEDUPDATE($T_{in}$, $T_{out}$) Line
18:   Swap $T_{in}$ and $T_{out}$
19:   end for
20: $C_i \leftarrow C_i + 1$
21: $I_i \leftarrow I_i \times R$
22: end while

typically within the range of 0.0 and 1.0. If we encode a 2D $n \times t$ matrix into a 2D texture, for an entry at the $i$th column and the $j$th row of the matrix, its 2D texture coordinate $(x, y)$ is computed as:

\[
x = i/(n - 1), \quad y = j/(t - 1). \quad (3.4)
\]

Accordingly, a 2D texture coordinate $(x, y)$ is corresponding to the entry $(i, j)$:

\[
i = \text{ceil}(x \times n), \quad j = \text{ceil}(y \times t). \quad (3.5)
\]

We use this transformation to read (write) the points from (into) the textures.

Algorithm 4 lays out the TEXTUREBASEDUPDATE fragment shader. According to
Algorithm 3 TextureBasedSubdivision($T_{in}$ : input texture; $T_{out}$ : output fbo; $C_i$ : simulation_cycle)

1: for each pixel $p_{ij}$ of $T_{out}$ in parallel do
2:   if $p_{ij}$ is an endpoint at $C_i$ and $C_i > 0$ then
3:     Fetch the corresponding endpoint in $T_{in}$ using texture lookup and copy the value to $p_{ij}$.
4:   else
5:     Compute $p_{ij}$ as a subdivision point according to the previous subdivision points in $T_{in}$ using texture lookup.
6:   end if
7: end for

Algorithm 4 TextureBasedUpdate($T_{in}$ : input texture; $T_{out}$ : output fbo)

1: for each pixel $p_{ij}$ of $T_{out}$ in parallel do
2:   Fetch the corresponding points in $T_{in}$ using texture lookup and use them to compute $F_s$ and $F_e$ with respect to $p_{ij}$.
3:   $F_{p_{ij}} \leftarrow F_s + F_e$.
4:   Move $p_{ij}$ a small distance in the direction of $F_{p_{ij}}$.
5: end for

the texture coordinate of each pixel $p_{ij}$ of $T_{out}$, we can use Equation 3.5 to compute the corresponding indexes of line and point, and locate the indexes of the points that are needed in Equations 3.1, 3.2, and 3.3. These indexes are then translated into the texture coordinates in $T_{in}$ using Equation 3.4, which allows us to fetch the corresponding points in $T_{in}$ using texture lookup and use them to compute $F_s$, $F_e$, and $F_{p_{ij}}$ with respect to $p_{ij}$. We then move $p_{ij}$ a small distance in the direction of $F_{p_{ij}}$, and render $p_{ij}$ as a color into $T_{out}$.

We swap $T_{in}$ and $T_{out}$ during each simulation cycle and each iteration step (Lines 14 and 18 in Algorithm 2) to enable GPU memory read/write via ping-pong buffering. In this way, we can always compute the new subdivision points and write them into the output texture according to the previous points stored in the input texture. Similarly, we can iteratively compute the forces and update the position of each point by flipping the input and output textures.
The entire bundling process is completed until we perform all $C$ simulation cycles. We obtain the output texture where the entries are filled with the final positions of subdivision points of all edges. Then we need to render the edges encoded in the texture.

### 3.3.3.3 Rendering

After the final simulation cycles, we transform the structure of the 2D matrix of the output texture into a 1D array, and bind this array into a Vertex Buffer Object (VBO) [85]. Then we can interactively render a large number of lines using VBO. Because the output texture and the VBO are all located in the GPU memory, we can directly conduct rendering in GPU and avoid the costly data transferring between CPU and GPU in the conventional method shown in Figure 2.1.

We note that texture lookup, FBO, VBO, and fragment shaders are standard graphics functions in the specification of OpenGL and WebGL, and are nearly fully supported by all graphics vendors. Thus, our method can be adapted to most devices with standard graphics processing units.

### 3.4 Pixel-Based Edge Bundling

TBEB has used parallel features of WebGL for edge bundling processing (Section 3.3). We further improve the scalability of parallel bundling for large graphs, and ensure interactivity and quality. After a careful study of the family of edge bundling methods, I found that the **Kernel Density Estimation Based** (KDE-based) method is one of the most appropriate methods to visualize large graphs. The rationale to choose a KDE-based method is discussed in Section 3.4.1. The current fastest edge bundling methods CUBu and FFTEB are KDE-based, and use parallel computing architectures such as NVidia’s CUDA or OpenCL. It is non-trivial to deploy these methods on web-based platforms because the support of the
parallel computing library is limited. In this section, I present another new method that uses WebGL’s texture feature and shaders programming for parallel computing, and enables a real-time edge bundling construction for large graphs on any standard web browsers. The modification required to adapt KDE-based method on web-based platforms and the implementation details will be discussed in Section 3.4.2.

3.4.1 Kernel Density Estimation Edge Bundling

Given an input graph \( G = (V, E) \subset \mathbb{R}^2, V = \{v_i\}, E = \{e_i\} \) where \( v_i \) is a vertex and \( e_i \) is an edge of \( G \), my framework aims to visually bundle the edges based on a similarity metric, and the bundles construction can be rendered in real-time on web-based platforms. In edge bundling visualization, an edge \( e_i \) is visualized as a straight-line or curve, and can be modeled as a polyline. The control points, namely sites, of the polylines, are either subdivided (like FDEB and TBEB) or uniformly sampled by a predefined sampling step \( \sigma \). Formally, the sites are presented as \( x_j \), and thereby \( e_i = \{x_j\} \) [47, 63, 110, 122]. In this section, we use the algorithm in KDE-based edge bundling methods. We will describe the basic pipeline of the KDE-based methods, and how I exploit the idea to our implementation.

Researchers have presented several KDE-based methods [47, 63, 84, 110]. They essentially use the Mean Shift principle [18]. The idea is first to uniformly sample each edge with a predefined step \( \sigma \), i.e., edges with different length have a different number of sites. The second step is to iteratively group the edges using the normalized gradient of an edge density map \( \rho \) of \( E \). To obtain the edge density map \( \rho : \mathbb{R}^2 \rightarrow \mathbb{R}^+ \), all sites \( x_j \) of set \( P \) on all edges of \( E \) are convolved with a radial kernel \( K \):

\[
\rho(x \in \mathbb{R}^2) = \sum_{y \in P} K\left(\frac{|x - y|}{p_R}\right),
\]

where \( p_R \) is the radius of \( K \), and \( K \) is a kernel function (e.g., Epanechnikov or Gaussian).
Next, the sites $x_j$ are advected according to the normalized gradient of $\rho$ with a distance $p_R$. Formally,

$$x_j^{\text{new}} = x_j + p_R \frac{\nabla \rho}{||\nabla \rho||},$$ \hspace{1cm} (3.7)

CUBu, one of the representative KDE-based edge bundling methods, proposes a solution that a per-pixel site-density buffer $\mathcal{C} : \mathbb{R}^2 \rightarrow \mathbb{N}$ is created to accumulate the sites that are rendered into $\mathcal{C}$. $\mathcal{C}(x)$ gives the number of sites of $E$ that fall inside pixel $x$. Hence, for each pixel $y$, the density map $\rho$ is computed as:

$$\rho(y) = \sum_{x \in T(y)} K(||x - y||) \mathcal{C}(x),$$ \hspace{1cm} (3.8)

where $T(y)$ is a disk of radius $p_R$ centered at $y$. Equation 3.7 is applied to advect sites after $\rho$ is computed. Every edge $e_i$ is resampled by replacing $\{x_j\}$ with $\{x_j^{\text{new}}\}$ after every advection iteration. At the final step of the bundling process, a 1-D Laplacian smoothing is performed to remove the jitter effect of the resulting polylines. The complexity of KDE method is $O(p_I \cdot p_N \cdot p_S)$, where $p_I$, $p_N$, and $p_S$ are the image resolution, the number of bundling iterations, and the counts of all sites, respectively.

I adopt the idea of KDE-based method in our framework. Using KDE-based on web-based platforms has multiple advantages. First, optimizing the implementation of Mean Shift principle on GPUs, the KDE-based method reveals several accuracy and scalability advantages [64]. Edges are sampled and accumulated into a buffer of pixels, normalized gradients are thus generated to advect the sampled points. The computational complexity is independent of the number of edges and vertices. Hence, it is suitable to bundle very large graphs. Second, KDE-based methods can be easily parallelized. The density map can be easily computed using OpenGL, WebGL, or other web-based rendering tools by encoding the sites’ locations $x_j$ with a 2D floating-point texture, and accumulating the sites
in a floating-point image buffer. Similarly, the calculation of normalized gradients can also be encoded with a 2D floating-point texture, and advect the sites in a ping-pong buffer manner. Third, the web-based implementation I use in our framework is WebGL. To my best knowledge, it has been nearly universally supported by nowadays web browsers, and thereby ensures the portability. Therefore, I choose to use the texture feature of WebGL to implement KDE-based methods to visualize large graphs on web-based platforms. Next, I will describe the modification to the KDE-based method on web platforms.

3.4.2 Implementation

To enable interactive edge bundling on web-based platforms, GPU acceleration is required for the bundling process. However, the main challenge to perform parallel computing on web-based platforms is that the current web-based techniques lack the support of direct GPU memory access. I build my framework upon the method of TBEB. The basic idea of TBEB is to leverage the texture feature and shader programs of WebGL to conduct GPU memory access and operation. For GPU memory read, data can be first uploaded from the main memory to the GPU memory via texture binding, and then access texture data via texture lookup in vertex and fragment shader programs. For GPU memory write, a Framebuffer Object (FBO) is bound with a texture, and then the data are rendered into the FBO. The rendering function can be customized using programs of vertex and fragment shaders so that the data written to the FBO can be manipulated. It is not feasible in WebGL to simultaneously read and write one texture. To address this problem, the ping-pong buffering (or double buffering) method [85] is used to read the input and write the output through different textures.

The limitation of TBEB is that it requires non-uniform sampling, i.e., each edge has the same number of sampling points. It encodes each sample point as a 4-component pixel in a 2-
D floating point texture. It works well with some modest datasets with just up to 2 thousand edges. It cannot address very large graphs since the subdivision sampling may make the size of sample points be very big and exceed the size of the maximal texture. TBEB and FDEB typically require a large number of sampling points to generate a pleasing result [81]. To solve this scalability problem, I exploit the idea of the KDE-based methods. As mentioned in Section 3.4.1, I use a uniform sampling method to sample edges, such that the number of sample points is significantly reduced. Then, the sample points can be accumulated on an image buffer, technically, an FBO. The subsequent operations, including kernel splatting, gradient computing, and position updating, can be conducted in a double buffering manner. Leveraging the KDE-based methods and the parallel computing of WebGL together, I realize an effective and interactive edge bundling visualization on web-based platforms. Next, we describe our implementation and technical details. Algorithm 5 layouts the pipeline of my implementation.

Initially, every edge $e_i$ of the input graph $G$ is sampled with a sample step $\sigma$. The sample points, namely, sites $\{x_i\}$ are stored in a texture $T_i$. We refer to this texture as an indexing texture. It stores the coordinates of $\{x_i\}$ by encoding the coordinates of the sites with 4-component RGBA pixel values. In particular, we intentionally subdivide each edge into an odd number of segments so that a 4-component RGBA pixel value stores a segment with 4 coordinates (one segment consists of 2 points), which facilitates a relatively simple encoding. Hence, the width of $T_i$ can be defined:

$$I_w = \begin{cases} \frac{p_s}{2} & \text{if } \frac{p_s}{2} < \text{max}, \\ \text{max} & \text{otherwise}, \end{cases}$$

(3.9)

where $\text{max}$ is a predefined parameter. It can not be larger than the maximum width of an allocated texture. Recall that $p_s$ is the counts of all sites. The height of the texture $I_h$ is
**Algorithm 5** PARALLELWDEB

1: // Initialization
2: \( I \leftarrow \) the number of iteration steps
3: \( D \leftarrow \) the decreasing factor
4: \( R \leftarrow \) the kernel radius
5: \( I_i \leftarrow I_0 \) // initialize the iteration number
6: \( E \leftarrow \) the edges of the graph
7: \( \sigma \leftarrow \) the sampling step
8: \( \{x\} \leftarrow Function(E, \sigma) \) // sample \( E \) based on \( \sigma \)
9: \( T_I \leftarrow \) indexing texture
10: \( T_O \leftarrow \) indicator texture
11: \( T_{in} \leftarrow \) input position texture
12: \( T_{out} \leftarrow \) output position texture
13: Build location array \( L \) and indexing array \( T_I \) based on \( \{x\} \) and \( E \)
14: \( T_{in} \leftarrow T_I \) // initialize \( T_{in} \) with \( T_I \)

15: // Simulation
16: while \( I_i < I \) do
17:     // Compute histogram
18:     \( T_H \leftarrow \) histogram texture
19:     \( F_H \leftarrow \) histogram FBO
20:     Bind \( F_H \) with \( T_H \) // bind texture with FBO
21:     Render \( \{x\} \) into \( F_H \) // accumulate pixel
22:     Synchronization
23:     // Compute gradients
24:     \( T_G \leftarrow \) gradient texture
25:     \( F_G \leftarrow \) gradient FBO
26:     Bind \( F_G \) with \( T_G \) // bind texture with FBO
27:     Render \( F_G \) with \( T_H \) and \( R \) // compute the gradient
28:     Synchronization
29:     // Update Positions
30:     \( F_P \leftarrow \) position FBO
31:     Bind \( F_P \) with \( T_{out} \) // bind texture with FBO
32:     Render \( F_P \) with \( T_{in}, T_O \) and \( T_G \) // update position
33:     Synchronization
// Resampling
32: \( F_R \leftarrow \) resampling FBO
33: Bind \( F_R \) with \( T_{in} \) // bind texture with FBO
34: Render \( F_R \) with \( T_{out} \) // resample sites
35: Synchronization

// Smoothing
36: \( F_S \leftarrow \) smoothing FBO
37: Bind \( F_S \) with \( T_{out} \) // bind texture with FBO
38: Render \( F_S \) with \( T_{in} \) // smooth edges
39: Synchronization
40: Swap \( T_{in} \) and \( T_{out} \) // swap texture for next iteration
41: \( I_i \leftarrow I_i + 1 \)
42: \( R \leftarrow R \times D \)
43: end while

\[ I_h = \text{ceil} \left( \frac{p_s}{\max} \right), \]  \hspace{1cm} (3.10)

Additionally, we need to create a texture \( T_O \) that records a point in \( T_j \) is either an endpoint or a site since endpoints of edges and sites may overlap in some pixel. Similar to CUBu [110], we accumulate sites on an image buffer. In our implementation, we render all the sites of the graph into a Frame Buffer Object (FBO) \( F_H \). The width and height of \( F_H \) are set to be the resolution of the display. \( F_H \) is bound with a 2D floating-point texture \( T_H \). After rendering, \( T_H \) gives a histogram, where the number of sites falling into every pixel are recorded. The value of the pixel in the histogram is encoded as either \( R \), \( G \) or \( B \) value in \( T_H \). Here, we restrict the rendering point size to be exactly 1 pixel, i.e., one site can only be rendered in one pixel in \( T_H \). To accumulate the number of rendering sites in \( T_H \), the source and destination factors of the blending function must be set to be \textit{GL_SRC_COLOR}. The width and height of \( F_H \) and \( T_H \) are the same as the width and height of the display resolution.

Second, after the histogram is built, we can compute the normalized gradients by splatting a kernel function. To do that in WebGL, we need to bind an FBO \( F_G \) with a 2D
Algorithm 6 PARALLELGRADIENT($T_H$ : input.texture; $T_G$ : output.fbo; $R$ : kernel.radius)
1: for each pixel $p_{ij}$ of $T_G$ in parallel do
2: for each pixel $p_{ij}$ inside the disk centered at $p_{ij}$ do
3: // apply Gaussian splatting with radius $R$
4: Use texture lookup from $T_H$, apply weighted function $W$, and copy the sum of
5: weighted values to $p_{ij}$
6: $p_{ij} \leftarrow W(p_{ij}) + W(p_{ij})$.
7: end for
8: Render $p_{ij}$ into $T_G$.
9: end for

Algorithm 7 PARALLELUPDATE($T_O$ : input.texture; $T_G$ : input.texture; $T_{in}$ : input.texture; $T_{out}$ : output.fbo)
1: $G_{p_{ij}}$ // the advection vector of $T_G$ in the position of $p_{ij}$.
2: for each pixel $p_{ij}$ of $T_{out}$ in parallel do
3: Fetch the corresponding sites in $T_{in}$ using texture lookup and learn the position of the
4: advection vector in $T_G$.
5: Fetch the corresponding sites in $T_G$ using texture lookup and use them to copy the
6: value of the advection vector to $G_{p_{ij}}$.
7: if $p_{ij}$ is a site according to $T_O$ then
8: Move $p_{ij}$ based on $G_{p_{ij}}$.
9: end if
10: Render $p_{ij}$ into $T_{out}$.
11: end for

floating-point texture $T_G$. The 2D gradient values are encoded into the color components of
the texture $T_G$. The kernel splatting is conducted in a customized fragment shader program.
During the rendering process, $T_H$ is also passed to the GPU memory. In the fragment shader
program, kernel splatting can be easily done by applying a Gaussian weighted function
$W$. Then, a gradient filter is used to compute the normalized gradient for each pixel in $T_G$.
After rendering, every color component of $T_G$ contains an advection vector for later position
updating. The width and height of $F_G$ and $T_G$ are identical to the width and height of the
display resolution. Algorithm 6 shows the PARALLELGRADIENT fragment shader.

Third, we update the positions of the sites. Recall that we have created an indexing
texture $T_I$ and $T_O$. We can update the position of each site of $T_I$ in a customized fragment
shader program with $T_I$, $T_O$, and $T_G$ as inputs. We first make $T_{in} = T_I$. We need another texture buffer $T_{out}$ to be bound with an FBO $F_p$. We then pass $T_{in}$, $T_O$, and $T_G$ to the GPU memory, and render the new positions as a color component into $T_{out}$. Note the sizes of the aforementioned texture $T_H$ and $T_G$ are identical to the display resolution, while the width and the height of $T_{in}$ and $T_{out}$ are $I_w$ and $I_h$, respectively. The key to this step is the translation between the texture coordinates and the indexes of sites (pixels). When we encode a data set into a texture, a data point is accessed through a 2D texture coordinate $(x, y)$, where the $x$ or $y$ component of texture coordinate is typically between 0.0 and 1.0. Recall that, if we encode a 2D $n \times t$ matrix into a 2D texture, the transformation between an entry at the $i$th column and the $j$th row of the matrix and its 2D texture coordinate $(x, y)$ can be computed by Equations 3.4 and 3.5.

We use this transformation to read the points from the textures. In this step, the rendering resolution is set to be $I_w \times I_h$. In the fragment shader program, each pixel contains the position of one segment (4-component RGBA value stores two 2D sites). From the color component $R$ and $G$ or $B$ and $A$ in $T_{in}$, we can gain the index of the gradient value in $T_G$. Using Equations 3.4 and 3.5 to translate the corresponding coordinate in $T_G$, we can retrieve the gradient value for this site. We ignore the update of a point if its record in $T_O$ indicates it is an endpoint. Algorithm 7 shows the PARALLELUPDATE fragment shader. After rendering, $T_{out}$ contains the updated position of all sites and the original endpoints.

KDE-based methods generate lattice artifacts. Hence, resampling and smoothing are applied to increase the readability of the final drawing. The resampling and smoothing can also be parallelized. Algorithm 8 shows the PARALLELRRESAMPLE fragment shader. $T_{out}$ has been generated from the position updating step. We need to create an FBO $F_R$, and bind it with $T_{in}$. In this step, $T_{out}$ is the input texture and $T_{in}$ is the output texture. In the resampling fragment shader program, we first define a distance threshold $H$. For every site $p_{ij}$, search forward along the edge to find the previous site $pre_{p_{ij}}$ whose distance to $p_{ij}$ is
Algorithm 8 PARALLELRESAMPLE($T_{in}$ : input texture; $T_{out}$ : output fbo)

1: $|\cdot|$ ← euclidean distance
2: $H$ ← distance threshold
3: $pre_n$ ← number of previous adjacent sites that are close to the current site $p_{ij}$
4: $next_n$ ← number of next adjacent sites that are close to the current site $p_{ij}$
5: $pre_{p_{ij}}$ ← the previous adjacent site of the current site $p_{ij}$
6: $next_{p_{ij}}$ ← the next adjacent site of the current site $p_{ij}$
7: for each pixel $p_{ij}$ of $T_{in}$ in parallel do
8: while $|pre_{p_{ij}} - p_{ij}| < H$ do
9: $pre_{p_{ij}}$ = the previous sites of $pre_{p_{ij}}$
10: $pre_n$++;
11: end while
12: while $|next_{p_{ij}} - p_{ij}| < H$ do
13: $next_{p_{ij}}$ = the next sites of $pre_{p_{ij}}$
14: $next_n$++;
15: end while
16: if $pre_n \leq next_n$ then
17: Move $p_{ij}$ a small distance $\frac{|pre_{p_{ij}} - p_{ij}|}{pre_n + 1}$ towards $pre_{p_{ij}}$.
18: else
19: Move $p_{ij}$ a small distance $\frac{|next_{p_{ij}} - p_{ij}|}{next_n + 1}$ towards $next_{p_{ij}}$.
20: end if
21: Render $p_{ij}$ into $T_{in}$.
22: end for

larger or equal to $H$. Count the site number $pre_n$ between $p_{ij}$ and $pre_{p_{ij}}$. So on and so forth for the backward search to find $next_{p_{ij}}$ and $next_n$. We then compare $pre_n$ and $next_n$. If $pre_n$ is smaller, we move $p_{ij}$ towards $pre_n$ with a distance of $\frac{|pre_{p_{ij}} - p_{ij}|}{pre_n + 1}$, and vice versa.

The smoothing step conducts a 1-D Laplacian smoothing on each edge, i.e., convolve the position of a site with its adjacent sites. Algorithm 9 shows the PARALLEL.SMOOTH fragment shader. After writing the new positions to $T_{out}$, we swap $T_{in}$ and $T_{out}$ for the next iteration. After the iterations, we bind the output texture $T_{out}$ into a Vertex Buffer Object (VBO). Then we render the segments as polylines to the display. Because the output texture and the VBO are all located in the GPU memory, we can directly conduct rendering in GPU and avoid the costly data transferring between CPU and GPU.
Algorithm 9 PARALLELSMOOTH($T_D : input \_texture; T_{out} : input \_texture; T_{in} : output \_fbo$)

1: $\{pre_{p_{ij}}\} \leftarrow$ the previous adjacent sites of $p_{ij}$ ($\{pre_{p_{ij}}\}$ and $p_{ij}$ are in the same edge)
2: $\{next_{p_{ij}}\} \leftarrow$ the next adjacent sites of $p_{ij}$ ($\{next_{p_{ij}}\}$ and $p_{ij}$ are in the same edge)
3: for each pixel $p_{ij}$ of $T_{out}$ in parallel do
4: Fetch the previous sites $\{pre_{p_{ij}}\}$ of $p_{ij}$ from $T_{in}$.
5: Fetch the next sites $\{next_{p_{ij}}\}$ of $p_{ij}$ from $T_{in}$.
6: Convolve the position of $p_{ij}$ with $\{pre_{p_{ij}}\}$ and $\{next_{p_{ij}}\}$.
7: Render $p_{ij}$ into $T_{out}$.
8: end for

3.5 Results

3.5.1 Visualization

In this section, I will compare TBEB and PBEB with other web-based methods to justify the visualization result of my work. Figure 3.3 shows the visualization results of the US airlines graph. As shown in Figure 3.3 (a), a direct visualization of node-link diagram can easily incur visual clutter, and it is relatively difficult to perceive the main airline patterns from the graph. In the visualization results of edge bundling methods, relational patterns can be easily perceived. For example, the highlight area in Figure 3.3 (a) is visually occluded. In the edge bundling visualization results (Figure 3.3 (b), (c), (d) and (e)), we can clearly perceive a “hub” in the center of the highlight area. Geographically, this “hub” is Atlanta. From the visualization results of edge bundling methods, we learn that Atlanta is an airline hub that is used by multiple airlines to concentrate passenger traffic and flight operations. Moreover, the high quality of TBEB (Figure 3.3 (d)) and PBEB (Figure 3.3 (e)) method are arguably better than FDEB and MINGLE in terms of revealing subtle graph structure and details.

From Figure 3.3 (b), (d) and (e), we can see a major pattern corresponding to the airline routes between the west coast of the US and the Northeastern US. Specifically, one side of the airline routes connect the Northeastern US, and then the routes split into two groups. One connects to the Northwestern US (mainly Washington), and the other connects to the
Southwestern US (mainly California). There is a very subtle detail in this pattern. From the results of FDEB and MINGLE, we might misunderstand that the airline routes directly connect the Northwestern and the Southwestern with the Northeastern. TBEB (Figure 3.3 (d)) and PBEB (Figure 3.3 (e)) show that this is not the case. Some of the airlines first route to another airline hub, which is geographically the Indianapolis. This subtle pattern cannot be perceived in Figure 3.3 (b) and (c). We can see that the results generated by FDEB and MINGLE have significantly reduced the visual clutter compared to the original node-link diagram. However, some dense areas still have visual clutter. That can mislead users in some subtle details. TBEB and PBEB not only reveal the high-level relational patterns, but also preserve most subtle graph structures, avoiding the problem of FDEB and MINGLE.

The complexity of PBEB is independent of the number of edges and vertices as its complexity is $O(p_I \cdot p_N \cdot p_S)$, where $p_I$, $p_N$ and $p_S$ are the image resolution, the number of bundling iterations and the counts of all sites, respectively. However, the model of FDEB and TBEB cannot enable an edge bundling visualization with a very large graph because of the limitation of the texture size. PBEB accumulates sample points on pixels, such that a density histogram is generated. Gradient calculation and advection can be conducted based on the density histogram, thus avoiding the issue of FDEB and TBEB. Next, I will show a visualization using PBEB on a large graph with half of a million edges. To the best of my knowledge, PBEB is the first web-based edge bundling method that can handle such a large graph on web browsers. Figure 3.4 shows the result of PBEB on the large US migrations dataset using a Chrome browser.

In Figure 3.4 (a), a considerable number of edges incur severe visual clutter. By using PBEB, the high-level patterns of migration can be perceived. Since PBEB uses the shader rendering method [110], the brighter area means the denser place where the edges converge. We can observe north-south and the east-west migration patterns, and learn that north-south migrations mainly happened in the Mideast US, while the pattern is not salient in the Western
Figure 3.3: Visualizing an airlines dataset (2100 edges) using a node-link diagram (a), a web-based implementation of FDEB (b), a web-based implementation of MINGLE (c), my implementation of TBEB (d), and my implementation of PBEB (e).

US. This observation demonstrates that PBEB is effective to visualize large graphs using web-based techniques, whereas the original node-link diagram and other edge bundling implementations are not able to achieve this on web-based platforms.

3.5.2 Performance Evaluation

In this section, I evaluate the performance of my implementations by comparing the following methods:
Figure 3.4: The visualization results of the large US migration graph (545,881 edges) using a direct visualization of node-link diagram (a) and the PBEB method (b).

- the FDEB method using a CPU implementation [45],
- the FDEB method using a CUDA implementation [122],
- the FDEB method using a $\mathbb{D}^3$ implementation [22],
- the TBEB method using a WebGL implementation,
- the MINGLE method using a JavaScript implementation [77],
• the FFTEB method using a CUDA implementation [63],

• the PBEB method using a WebGL implementation.

I use the following device in the experiment:

• a desktop with an 8X Intel Core i7 3.60GHz CPU and an NVIDIA GeForce GTX 1080 ti GPU,

• a Nexus 9 tablet with a dual-core Denver 2.3GHz CPU and a Kepler DX1 GPU.

For the web-based edge bundling applications, I use a Google Chrome web browser to generate the results. I use the following three typical graph datasets in the experiment:

• a US airlines dataset with 2100 edges and 235 vertices,

• a France airlines dataset with 17,273 edges and 34,194 vertices,

• a US migrations dataset with 545,881 edges and 3075 vertices.

Table 3.1 shows the performance comparison on the desktop, where Sites are the number of sample points generated in the sampling step, and Time is the average elapsed time that one iteration takes for similarity (or compatibility) calculation, bundling, and rendering. For FDEB, TBEB, and MINGLE, each edge has the same number of sites in every iteration. For FFTEB and PBEB, uniform sampling is used. I use a sample step of 5% of the display size to sample each edge in the implementation of PBEB. For FFTEB, I use the uniform sampling strategy of CUBu [110]. Hence, the number of the sites of different methods may be different. The timing result is formally calculated as:

\[
Time = \frac{pT}{pN},
\]  

(3.11)
where $p_T$ is the total elapsed time of an edge bundling application taking to generate the final graph drawing, and $p_N$ is the number of bundling iterations. For some existing implementations that only render once in the final result, I scale their rendering time according to the number of iterations to avoid bias. As shown in Table 3.1, the performance of TBEB is significantly better than the CPU- and JavaScript-based methods. PBEB outperforms other methods on all datasets, and achieves interactive framerates on the datasets of US airlines and France airlines. For the large US migrations dataset, the PBEB method achieves $50 \times$ speedups to the fastest state-of-the-art method, FFTEB. More importantly, the TBEB and PBEB methods can be deployed on web-based platforms, which is more portable.

Table 3.1: Performance comparison using the US airlines, France airlines, and US migrations graph on a desktop.

<table>
<thead>
<tr>
<th>Graph</th>
<th>FDEB (CPU) Sites, Time (ms)</th>
<th>FDEB (CUDA) Sites, Time (ms)</th>
<th>FDEB ($D^3$) Sites, Time (ms)</th>
<th>TBEB Sites, Time (ms)</th>
<th>MINGLE (JavaScript) Sites, Time (ms)</th>
<th>FFTEB (CUDA) Sites, Time (ms)</th>
<th>PBEB Sites, Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>France airlines</td>
<td>2.2M, 2.1M</td>
<td>2.2M, 926</td>
<td>2.2M, n/a</td>
<td>2.2M, n/a</td>
<td>2.2M, 1.04K</td>
<td>864K, 380</td>
<td>557K, 20</td>
</tr>
<tr>
<td>Large US migrations</td>
<td>70M, n/a</td>
<td>70M, n/a</td>
<td>70M, n/a</td>
<td>70M, n/a</td>
<td>70M, 93.6K</td>
<td>6.4M, 5.2K</td>
<td>2.4M, 103</td>
</tr>
</tbody>
</table>

Next, I compare TBEB and PBEB with FDEB using ($D^3$) [22] and MINGLE using (JavaScript) [77] in a Chrome Browser using the Nexus 9 tablet. I use the datasets of US airlines and France airlines. In my experiment, I found that all the methods cannot run the Large US migrations, and skipped the comparison on this dataset. In Table 3.2, we can see that the TBEB and PBEB are more significantly efficient than other web-based edge bundling methods. The results of Table 3.1 and Table 3.2 demonstrates my framework can work on not only high-end machines with dedicated graphics cards, but also ubiquitous smart devices.
Table 3.2: Performance comparison using the US airlines, France airlines, and US migrations graph on a tablet.

<table>
<thead>
<tr>
<th>Graph</th>
<th>FDEB (D^3)</th>
<th>TBEB</th>
<th>MINGLE (JavaScript)</th>
<th>PBEB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sites</td>
<td>Time (ms)</td>
<td>Sites</td>
<td>Time (ms)</td>
</tr>
<tr>
<td>US airlines</td>
<td>270K</td>
<td>3423</td>
<td>270K</td>
<td>49</td>
</tr>
<tr>
<td>France airlines</td>
<td>2.2M</td>
<td>n/a</td>
<td>2.2M</td>
<td>n/a</td>
</tr>
</tbody>
</table>

3.6 Conclusion

I have presented a new efficient framework for the construction and visualization of edge bundles of large graphs using standard features of consumer graphics hardware. My current framework supports TBEB and PBEB by adopting two prevailing edge bundling methods. The performance evaluation shows the TBEB and PBEB methods significantly improve the performance of edge bundling on web-based platforms, and the visualization results clearly demonstrate that the TBEB and PBEB methods can achieve high-quality visualization results to reveal high-level relational patterns. My proposed web-based implementation can be versatility applied on various devices, including ubiquitous smart mobile devices, without any modification. My solution has significantly enhanced the interactivity and the usability of edge bundling, and enabled a user to interactively explore a large graph on a mobile device without visual clutter.
Chapter 4

Edge Bundling using Moving Least Squares Approximation

4.1 Introduction

In Chapter 3, I have exploited the standard graphics hardware functions and libraries to improve the performance and the portability of exiting edge bundling algorithms on web-based platforms. The presented web-based framework has enabled interactive edge bundling on various machines, including ubiquitous smart mobile devices. However, the performance gain and the maximal graph size are still determined by underneath graphics hardware constraints (e.g., the maximal texture size).

In this chapter, I improve the performance of edge bundling from an algorithm complexity perspective. I introduce a novel edge bundling technique to generate edge bundles with moving least squares (MLS) approximation, namely MLSEB. Inspired by thinning an unorganized point cloud to curve-like shapes [61], I use a distance-minimizing approximation function to generate bundle effects. In particular, I first sample a graph into a point cloud data, and then use a moving least squares projection to generate curve-like bundles. Such
a local regression method can produce a better convergence result using fewer iterations. Therefore, MLSEB can avoid the costly processing steps, such as resampling and smoothing, in the existing methods, and lower the algorithm complexity of edge bundling.

I further evaluate the effectiveness of MLSEB. In general, edge bundling helps simplify graph drawings and increase readability, but yields distortion that makes it hard to preserve the faithfulness of original graphs [81]. To holistically address the evaluation of both readability and faithfulness for edge bundling visualization, Lhuillier et al. [64] suggested a general metric where a ratio of clutter reduction to the amount of distortion is computed to measure the quality of edge bundling visualization. Based on Lhuillier’s suggestion, I develop a quality assessment to evaluate edge bundling results. Using different real-world datasets, I demonstrate that MLSEB can produce bundle results with a higher quality, and is scalable and efficient for large graphs by comparing different edge bundling methods.

4.2 Related Work

Chapter 2 revisited edge bundling techniques. In this section, we will review some studies on quality evaluation in edge bundling and moving least squares approximation.

Serval studies introduced general metrics to quantify the readability [7, 88, 93, 106]. A few studies in edge bundling have defined quality assessments to evaluate the resulting bundles. Nguyen et al. [81] conducted a study on the faithfulness of force-directed edge bundling methods. Telea et al. [107] posed a comparison between different hierarchical edge bundling implementations, and surveyed the hierarchical edge bundling applications. Pupyrev et al. [87] and Kobourov et al. [54] worked towards measuring edge crossings. KDEEB [47] and CUBu [110] proposed post-relaxation if the distortion of edge bundles is too large, such that the mental map is preserved. For sequence graph edge bundling, Hurter et al. [48] used interpolation to preserve the mental map between sequence graphs. McGee
et al. [75] conducted an empirical study on the impact of edge bundling.

Moving least squares (MLS) has been widely used to approximate smooth curves and surface from unorganized point clouds [2, 62, 76]. Lee [61] constructed a curve-like shape from unorganized point clouds using a Euclidean minimum spanning tree. Least square projection (LSP) has been used in graph drawings [83], where multidimensional data points are projected into lower dimensions, and the similar relationship in neighboring points can be perceived.

4.3 Our Method

4.3.1 Background

Edge bundling techniques trade the increase of readability for overdrawing by bending edges to form bundle effects. Hence, edge bundle techniques naturally generate distortion from original graphs. Chapter 2 provided a formal definition of edge bundling. Let $G = (V, E) \subset \mathbb{R}^2, V = \{v_i\}, E = \{e_i\}$ be a graph, where $v_i$ is a vertex and $e_i$ is an edge of $G$. Then, for an edge $e_i$, the distortion between an unbundled drawing $D(e_i)$ and a bundled result $B(D(e_i))$ is measured by computing the distance between them, i.e., $\delta(D(e_i), B(D(e_i)))$.

To quantify the quality of a bundled graph, Lhuillier et al. [64] suggested using the ratio of clutter reduction $C$ to amount of distortion $T$ as a quality metric $Q$, i.e.,

$$Q = \frac{C}{T}, \quad (4.1)$$

In general, a larger $Q$ corresponds to a higher quality, and vice versa. Lhuillier et al. [64] further posed a distortion measure. The overall distortion $T$ between an original unbundled
graph and its bundled result can be defined as:

$$T = \sum_{i=1}^{n} \delta(D(e_i), B(D(e_i)))$$

(4.2)

where \(n\) is the number of edges. Equation 4.2 provides an intuitive metric to evaluate the distortion generated by a bundled graph. The calculation of clutter reduction has not been fully concluded in the existing work. I propose a simple method to evaluate clutter reduction \(C\), modify Equation 4.2 to compute \(T\), and then use \(C\) and \(T\) to quantify the quality \(Q\) of edge bundling (Section 4.5.2).

The main purpose of edge bundling is to achieve appealing bundle effects by bending edges, expressed by Equation 2.1 in Chapter 2. Meanwhile, according to Equation 4.1, an ideal algorithm should increase clutter reduction \(C\), while decrease amount of distortion \(T\), in order to achieve a higher quality \(Q\) of edge bundling. Therefore, we should holistically address Equations 2.1 and 4.1, which, however, has not been fully investigated in the existing work [64].

### 4.3.2 Sampling

In general, given a graph \(G\), a polyline is used to draw the line or curve presentation of an edge \(e_i\). Sample points \(x_i^k\), namely sites, are used to discretize the drawing of \(e_i\). Formally,

$$\{x_i^k | 1 \leq k \leq m_i\} \approx D(e_i)$$

(4.3)

where \(m_i\) is the number of sites for \(D(e_i)\). Note, many methods [47, 63, 84, 110] use a sampling step that is a small fraction of the size of the display to sample each edge, which means the number of sites of \(D(e_i)\) may be different. Similarly, the bundled drawing can
also be discretized as:

\[ B(\{x^i_k | 1 \leq k \leq m_i\}) \approx B(D(e_i)). \quad (4.4) \]

We measure the distortion between \( D(e_i) \) and \( B(D(e_i)) \) by summing the Euclidean distance between each pair of \( x^i_k \) and \( B(x^i_k) \). Let \(|\cdot|\) denote the Euclidean distance. Replace the edges in Equation 4.2 using Equation 4.3 and Equation 4.4, we have

\[ T = \sum_{i=1}^{n} \sum_{k=1}^{m_i} |\{x^i_k\}, B(\{x^i_k\})|. \quad (4.5) \]

Similarly, Equation 2.1 can be modified as:

\[ \forall (e_i \in G, e_j \in G) |\phi(e_i, e_j) < \phi_{max} \rightarrow |B(\{x^i_k\}), B(\{x^j_k\})| \ll |\{x^i_k\}, \{x^j_k\}|. \quad (4.6) \]

Therefore, we discretize each edge drawing \( D(e_i) \) of \( G \) by Equation 4.3. All the sample points generated by Equation 4.3 form a point cloud. According to Equation 4.6, \( x^i_k \) is moved to a new position \( B(x^i_k) \) by a bundling operator \( B \). In the case of kernel density estimation edge bundling [47, 63, 84, 110], \( x^i_k \) is moved to \( B(x^i_k) \) according to its local density gradient. These methods form the bundles by gathering sample points to their local density maxima, but do not consider the distortion of edges when moving sample points. Therefore, certain artifacts, such as lattice effects and subsampled edge fragments, can be incurred. The methods, such as resampling and post-relaxation [47, 110], have been proposed to address these issues. However, these methods typically introduce a significant performance overhead that is challenging to alleviate [110]. I develop a new bundling operator \( B \) with respect to Equation 4.6, and minimize the distortion of each sample point locally. Moreover, my method does not require resampling, and thereby can reduce the computational cost.
4.3.3 Moving Least Squares Approximation

I consider all the points formed by sampling, and assess the global distortion $T$ by expressing Equation 4.5 as:

$$T = \sum_{i=1}^{S} |x_i - B(x_i)|^2,$$

(4.7)

where $x_i$ is a site in the point cloud, and $S$ is the number of sites of all edges.

We assume there is a skeleton near $x_i$ and its neighborhood locally. A skeleton can be a suitable place to gather curves to form bundles [28]. Assume a skeleton can be interpreted as an implicit polynomial or piece-wise polynomial curve $f_i$, which is unknown. The unknown $f_i$ can be gained by computing the coefficients of $f_i$, i.e., by minimizing the following weighted least squares error $\varepsilon$ within a set $\mathcal{H}(x_i)$ consisting of $x_i$ and its neighbor sites:

$$\varepsilon = \sum_{j=1}^{h_i} |x_j - f_i|^2 \theta(|x_j - x_i|),$$

(4.8)

where $x_i \in \mathcal{H}(x_i)$, $x_j \in \mathcal{H}(x_i)$, $h_i$ is the size of $\mathcal{H}(x_i)$, and $|x_j - f_i|$ means the shortest Euclidean distance between $x_j$ and $f_i$. We define the bundling operator $B$ on $x_i$ as a two-step procedure: first to construct $f_i$, and then to project $x_i$ onto $f_i$. The projected point is thereby $B(x_i)$ that is on $f_i$. The distance $|x_i - B(x_i)|$ from $x_i$ to $B(x_i)$ is locally minimized by an appropriate nonnegative weighting function $\theta$. The input of $\theta$ is $|x_j - x_i|$, which is the distance of neighborhood $x_j$ to the site $x_i$. Instead of taking all sites of a graph into account, we use a circle of radius $r$ (bandwidth) centered at $x_i$ to collect the neighborhood $x_j$ for $x_i$.

If $\theta \equiv 1$, a least square (LS) approximation is generated. However, LS approximation does not work well to generate a polynomial curve that locally reflects the density distribution of neighborhood. Alternatively, the moving least squares (MLS) method can reduce a point cloud to a thin curve-like shape that is a near-best approximation of the point set [61, 62]. Hence, we use a local assessment to approximate $f_i$ [60]. The weighting function we use is
Figure 4.1: Two steps of our bundling operator $B$ on a site $x_{i,u}$ in an iteration $u$. First, a local implicit regression curve $f_{i,u}$ is constructed by the neighborhood of $x_{i,u}$ with a bandwidth $r$ using the MLS approximation. Second, $x_{i,u}$ is moved to a new position $x_{i,(u+1)}$ that is the projection of $x_{i,u}$ on $f_{i,u}$.

A cubic function [76]:

$$\theta(d) = \begin{cases} 
2\frac{d^3}{r^3} - 3\frac{d^2}{r^2} + 1 & \text{if } d < r, \\
0 & \text{if } d \geq r,
\end{cases}$$

(4.9)

where $d = |x_j - x_i|$. In this sense, minimizing Equation 4.8 leads to an MLS approximation so that $f_i$ is a local regression curve, and $|x_i - B(x_i)|$ is locally minimized. In other words, the distortion is locally minimized.

In our work, we use an MLS approximation to evaluate the distance $|x_j - f_i|$ for the neighborhood $\mathcal{H}(x_i)$ of $x_i$. Therefore, we use a basic projection [60] to construct the implicit local regression curve $f_i$: We take a partial derivative of Equation 4.8 with respect to each coefficient of $f_i$, make each partial derivative equal to zero, and then solve the system
of equations to generate all the coefficients of $f_i$ [78].

Similar to existing work [28, 47, 63, 84, 110], I implement the bundling operator $B$ through an iteration strategy. In our method, two steps are applied iteratively, as shown in Figure 4.1. We initially treat $x_i$ as $x_{i,0}$. Then, in each iteration $u$, the first step is to construct an optimal regression curve $f_{i,u}$ by thinning the unordered point cloud within $\mathcal{H}(x_{i,u})$, the neighborhood of $x_{i,u}$. In the second step, we project $x_{i,u}$ onto $f_{i,u}$ and obtain the projected point $x_{i,(u+1)}$, i.e., $B(x_{i,u})$. In this way, a site $x_{i,u}$ is moved to $x_{i,(u+1)}$ based on the weighting function $\theta$ of its neighborhood $\mathcal{H}(x_{i,u})$. Different from the kernel density estimation methods [28, 47, 63, 84, 110], MLS moves the site $x_{i,u}$ in the sense that the local error $\varepsilon$ is bounded with the error of a local best polynomial approximation [61]. In our current work, this process stops when the iteration number reaches a predefined threshold. Then, for each edge, we compute a B-spline curve based on the final positions of its sites. Figure 4.2 shows an example with two different iterations. For an illustration purpose, we show the corresponding B-spline curves for the iterations. In Figure 4.2, we can see that a curve-like skeleton is gradually formed from the point cloud through the iterations in the top row, and a bundle effect becomes increasingly distinct as shown by the B-spline results in the bottom row.

Most of the existing image-based techniques use kernel density estimation (KDE), essentially, a mean-shift method that evaluates the local density maxima and advects a site based on the gradients of the local density. However, KDE does not consider the distortion (Equation 4.2) when moving sample points, and thus resampling or post-relaxation is often required [47, 110]. Alternatively, our MLSEB method uses an MLS approximation that projects a site $x_i$ to its local regression curve $f_i$, where $f_i$ is locally approximated by minimizing the distance between $\mathcal{H}(x_i)$ and $f_i$ with a weighted function (Equation 4.8). Therefore, the distance between its original position $x_i$ and its projected position $B(x_i)$ is locally minimized based on the density of its neighborhood $\mathcal{H}(x_i)$. One advantage of our
Figure 4.2: Using a US airlines dataset as an example, I first sample each edge into a set of points (or sites). The resulting sites form a point cloud (top-left). The top row shows the point cloud is converged through an iterative MLS processing. The bottom row shows the corresponding B-spline results. The first column shows the initial result before MLS. The following columns show the results generated after the 2nd and 8th iteration, respectively.

method is that MLS does not need to resample each edge in bundling iterations because sites are projected into curves that do not generate over-converge artifacts or lattice effects. Fröhlich et al. [33] showed that MLS produced better convergence results than KDE in biological studies. However, it remains an open question to determine if KDE or MLS is better than one another in edge bundling. In Section 4.5.2, I will develop a quality assessment from Equation 4.1, and use it to evaluate and compare the quality of the drawings generated by our MLSEB method, the FFTEB method (a KDE-based method), and the FDEB method (a force-directed method).

4.4 Implementation

Our implementation involves simple data structures and computations, and thus is easy to implement. First, we sample the edges of an input graph. We use the same scheme as
KDEEB’s [47] to sample the input edges with a uniform step \( \rho \). The most time consuming step in our method is gathering the neighborhood for every site. A typical solution in a GPU implementation is to use Uniform Grid [40] that subdivides the space into uniformly sized cells. We use this method and set the size of the cell to be \( \frac{2}{3}r \) (\( r \) is a prescribed radius or bandwidth) such that we can limit the search space of each site to only cover at most 9 grid cells [40], thus avoid an \( O(S^2) \) search time for \( S \) sites.

At the start of each iteration, all the sites are put into the corresponding cells according to their current positions. This can be easily parallelized using CUDA on a GPU [40]. Then, we project each site onto its local regression line. The solution to compute the coefficients of Equation 4.8 is introduced in the work [60]. It only requires a constant time to solve the coefficients of a linear or quadratic system of equations. This can also be parallelized using a GPU because computing the new projection position for every site is independent.

To enhance the visualization of a bundled graph, we use the same shader scheme of CUBu [110]. We use the HSVA (i.e., hue \( H \), saturation \( S \), value \( V \), and alpha \( A \)) color representation to visualize edges. Each edge site \( x_i \) is encoded with an HSVA value. We encode the direction and the length of the corresponding edge into \( H \) and \( S \), respectively. \( V \) and \( A \) are used with a parabolic profile function \( P \), formally:

\[
\mathcal{P}(x) = \sqrt{1 - 2|t(x) - \frac{1}{2}|}, \quad (4.10)
\]

\[
V(x) = \frac{l}{l_{\text{max}}} + (1 - \frac{l}{l_{\text{max}}}) \mathcal{P}, \quad (4.11)
\]

\[
A(x) = \alpha(1 - \frac{l}{l_{\text{max}}} + \frac{l}{l_{\text{max}}} \mathcal{P}), \quad (4.12)
\]

where \( l \) is the length of the edge, \( l_{\text{max}} \) is the longest edge in the graph, \( t \in [0, 1] \) is the edge arc-length parameterization, and \( \alpha \) controls the overall transparency of all edges.

Next, we analyze the complexity of our MLSEB method. Similar to the existing KDE-
based methods [47, 63, 84, 110], MLSEB requires gathering neighbor sites for computation. After gathering, KDE-based methods conduct kernel splatting, gradient calculation, and site advection, which use a constant time for each site. In MLSEB, the time to solve Equation 4.8 and project a site to its local approximated curve is also constant for each site. Thereby, the complexity of MLSEB is the same as the traditional KDE-based methods, which is \( O(I \cdot N \cdot S) \), where \( I \) is the image resolution, \( N \) is the number of bundling iterations, and \( S \) is the number of sample points. However, MLSEB does not need additional operations, such as resampling, that is employed in the existing KDE-based methods.

We explore the parameter choices of MSLEB as follows. Similar to most the existing edge bundling methods, we use a step \( \rho \), which is 5% of the image resolution \( I \), to sample each edge. The bandwidth, \( r \), plays an important role in MLS to estimate the density information around each site. A larger bandwidth captures more sample sites to reflect a more global feature, while a smaller bandwidth reveals a more local feature. By following a similar strategy in FDEB [123] and KDEEB [47], we decrease \( r \) by a reduction factor \( \lambda \) after each iteration. Hurter et al. [47] stated that a kernel size follows an average density estimation when \( 0.5 \leq \lambda \leq 0.9 \). We set \( r \) to be \( 5\% \leq r \leq 20\% \) of the display size \( I \) to generate a stable edge-convergence result. Through a heuristic study, we found that it is sufficient to yield good results by setting the iteration number \( N \) between 3 and 10 and making the polynomial order of \( f_i \) in Equation 4.8 to be 1 or 2.

## 4.5 Results

### 4.5.1 Visualization and Performance Results

We apply our MLSEB method to several graphs and compare its effect and computational performance to the two existing methods: FDEB that is the classic force-directed method,
Figure 4.3: Visualize the US airlines dataset (the left column) and the US migrations dataset (the right column) with three different edge bundling methods, FDEB, FFTEB and MLSEB, respectively.
Figure 4.4: Visualize the France airlines dataset (17,274 edges) with FDEB, FFTEB, and MLSEB.

Figure 4.5: Comparison of FFTEB and MLSEB using a large US migrations dataset (545,881 edges).

and FFTEB that is the latest enhanced KDE-based method of image-based edge bundling algorithms (such as KDEEB and CUBu).

The left column in Figure 4.3 compares the visualization results of our MLSEB method with other bundling methods using the US airlines dataset (2101 edges). Our MLSEB method provides similar results, and generates tight, smooth and locally well-separated
bundles. High-level graph structures are also revealed in our results. The right column in Figure 4.3 shows the comparison using the US migrations dataset (9780 edges). Figure 4.4 shows another example using the France airlines dataset with 17274 edges. In these results, the main migration and airline patterns are clearly revealed using MLSEB. In the migrations dataset, FDEB and FFTEB fall short in showing some subtle structures of the original graph. For example, in the original node-link diagram of Figure 4.3(b), the edges (within the red box) connect the city of Portland to some cities in the northern U.S are distorted significantly from their original positions in the results of the FDEB (Figure 4.3(d)) and FFTEB (Figure 4.3(f)), while our MLSEB result has a distinguished bundle effect that reveals this subtle graph structure. In Figure 4.5, we compare the visual result of MLSEB to FFTEB using a large US migrations dataset with 545,881 edges. We encode the color of an edge with only its length in this example. MLSEB shows more long-length edge patterns than FFTEB.

Table 4.1 shows the performance comparison between our MLSEB method and the current fastest edge bundling method FFTEB. In our performance comparison, we used the US airlines graph, the US migrations graph, the France airlines graph, and the large US migrations graph. The timing results for MLSEB and FFTEB are based on one iteration, and we excluded the timing of memory allocation and data transferring for both methods. The devices used in our experiments are a desktop with an 8X Intel Core i7-6700K 4.0GHz CPU with 32GB memory and a NVIDIA GeForce GTX TITAN X GPU. Comparing with the fastest algorithm FFTEB in the state-of-the-art, we can clearly see that MLSEB is at the same order of magnitude of FFTEB in terms of computational speed, as shown in Table 4.1.
Table 4.1: Performance comparison.

<table>
<thead>
<tr>
<th>Graph</th>
<th>Edges</th>
<th>FFTEB</th>
<th></th>
<th>MLSEB</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Samples</td>
<td></td>
<td>Samples</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Time (ms)</td>
<td></td>
<td>Time (ms)</td>
</tr>
<tr>
<td>US airlines</td>
<td>2180</td>
<td>105K</td>
<td>40</td>
<td>85K</td>
<td>22</td>
</tr>
<tr>
<td>US migrations</td>
<td>9780</td>
<td>489K</td>
<td>48</td>
<td>207K</td>
<td>38</td>
</tr>
<tr>
<td>France airlines</td>
<td>17,274</td>
<td>864K</td>
<td>70</td>
<td>990K</td>
<td>94</td>
</tr>
<tr>
<td>Large US migrations</td>
<td>545,881</td>
<td>6.4M</td>
<td>123</td>
<td>5.8M</td>
<td>554</td>
</tr>
</tbody>
</table>

4.5.2 Quality Assessment of Bundled Graphs

Apart from comparing the visualization and performance results, we propose a quality metric to evaluate the quality of bundling drawings based on Equation 4.1.

Equation 4.1 gives a general quality metric $Q$ based on the ratio of clutter reduction $C$ to amount of distortion $T$. However, the quantification of clutter reduction $C$ has been not fully concluded in existing work. We propose to employ the reduction of the used pixel number $\Delta P$ in a graph drawing to measure $C$. Specifically, $C = \Delta P = P - P'$ that is the difference of the used pixel number $P$ of the original drawing and the used pixel number $P'$ of the bundled drawing.

Intuitively, $T$ can be given by Equation 4.5 that quantifies the total distortion of all the sample points. However, different methods can generate different numbers of sample points. For example, FDEB generates the same number of sample points for each edge, while our MLSEB method and the KDE-based methods sample different edges into different numbers of points. Thus, instead of the total distortion of all the sample points, we use the average distortion: $\bar{T} = \frac{T}{S}$, where $S$ is the total number of the sample points in the graph. Therefore,
Table 4.2: The quality comparison using the US migrations graph.

<table>
<thead>
<tr>
<th>Graph</th>
<th>Edges</th>
<th>FDEB</th>
<th>FFTEB</th>
<th>MLSEB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>s</td>
<td>p</td>
<td>p'</td>
</tr>
<tr>
<td>US airlines</td>
<td>2180</td>
<td>813K</td>
<td>32K</td>
<td>25K</td>
</tr>
<tr>
<td>US migrations</td>
<td>9780</td>
<td>3785K</td>
<td>34K</td>
<td>26K</td>
</tr>
<tr>
<td>France airlines</td>
<td>17274</td>
<td>6685K</td>
<td>81K</td>
<td>72K</td>
</tr>
<tr>
<td>Large US migrations</td>
<td>545881</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
</tbody>
</table>

we modify Equation 4.1 to

\[ Q = \frac{\Delta p}{T}. \]  

(4.13)

The rationale of Equation 4.13 is to measure how many pixels are decreased by generating one unit distortion. A higher value of \( Q \) means a better quality result. Table 4.2 shows the quantitative quality comparison between our MLSEB method, FDEB and FFTEB. Our comparison is based on the drawings with an image resolution of \( 400 \times 400 \), as shown in Figures 4.3, 4.4, and 4.5. All the statistic results are generated after a graph is bundled, i.e., after all iterations. We note that it makes less sense to compare the distortion in each iteration because the initial iterations of some methods, such as FDEB and FFTEB, may have surprisingly large distortion. It is more reasonable to compare the quality of results after the bundling iterations are finished. We also note that using different parameters, such as different iteration numbers and different bandwidths for different methods, can yield different results. We use the recommended parameters in FDEB’s and FFTEB’s papers [45, 63], which are the best results we can get from the existing work. The S columns in Table 4.2 show the numbers of the sample points in a graph using different methods.

We can see that the quality of MLSEB is generally better than the other two methods in terms of Equation 4.13. For the four different datasets, FFTEB makes the most clutter reduction. However, it also incurs more distortion. FDEB achieves a comparable quality.
as ours for the US migrations dataset; whereas, when the dataset is getting larger (France airlines), FDEB will generate tremendous distortion, as shown in Table 4.2 and Figure 4.4, thus lowering the quality score. Note when using the large US migrations dataset, the advantage of MLSEB over FFTEB becomes marginal. Overall, MLSEB gains the highest quantitative scores in terms of quality according to Equation 4.13.

4.6 Conclusion

I have presented a new edge bundling method MLSEB that holistically considers distortion minimization and clutter reduction. Inspired by the MLS work [2, 61], our approach generates bundle effects by iteratively projecting each site to its local regression curve to converge with other nearby sites based on its neighborhood’s density. Such a local regression curve can reduce the distortion of the local bundle. Our method is easy to implement. The timing result shows MLSEB is at the same order of magnitude of the current fastest edge bundling method FFTEB in terms of computational speed.

I use a quality assessment to evaluate the quality of resulting edge bundles. Our MLSEB method shows better results in our preliminary comparison. However, a more comprehensive comparison between our MLSEB method and the other methods requires further investigation, where other factors (e.g., edge crossing reduction) may be also considered.
Chapter 5

An Information-Theoretic Framework for Evaluating Edge Bundling Visualization

5.1 Introduction

Edge bundling is a promising graph visualization approach to simplifying the visual result of a graph drawing. Plenty of edge bundling methods have been developed to generate diverse graph layouts. However, it is difficult to defend an edge bundling method with its resulting layout against other edge bundling methods as a clear theoretic evaluation framework is absent in the literature. In this chapter, we propose an information-theoretic framework to evaluate the visual results of edge bundling techniques. We first illustrate the advantage of edge bundling visualizations for large graphs, and pinpoint the ambiguity resulting from drawing results. Second, we define and quantify the amount of information delivered by edge bundling visualization from the underlying network using information theory. Third, we propose a new algorithm to evaluate the resulting layouts of edge bundling using the
amount of the mutual information between a raw network dataset and its edge bundling visualization. Comparison examples based on the proposed framework between different edge bundling techniques are presented.

5.2 Related Work

We review existing work relevant to our information-theoretic framework for evaluating edge bundling visualization.

5.2.1 Graph Visualization and Evaluation Metrics

Graph visualization can help reveal the structure and patterns of an underlying network, and has been applied in various scientific and engineering domains, such as biological, technological, and social systems. The information visualization community has developed many sophisticated graph visualization methods. These methods mostly employ matrix, node–link diagram, and hybrid visual presentations, which help researchers and domain experts effectively gain insights into their data. A wide spectrum of graph visualization techniques and their applications have been surveyed in many studies. We refer readers to the following works and surveys for the diverse graph visualization approaches. Landesberger et al. [114] and Beck et al. [8] reviewed techniques for visualizing large graphs and dynamic graphs respectively. Herman et al. [43] and Vehlow et al. [112] conducted comprehensive surveys for graph visualization. Wang et al. [117] presented a visual analytics framework to reveal the ambiguity of graph drawings. To evaluate the quality of graph drawings, researchers have provided and discussed many metrics for graph readability [93, 88, 118, 90, 92, 91, 31, 23, 94, 54, 27] and graph faithfulness [80]. The readability of a graph drawing is often measured by aesthetics criteria. They can be concluded as to minimize edge crossings, edge bends, and area used, and maximize orthogonality and structure symmetry. To optimize
two or more of these criteria is NP-hard [7]. Among the criteria, edge crossing is widely acknowledged as the most important one [88]. Edge bends, orthogonality, and structure symmetry are not practical to evaluate the quality of edge bundling drawings. Edge bundling methods inherently create many edge bends, which indeed reduce many small-angle edge crossing to avoid visual occlusion. Orthogonality and structure symmetry criteria often require adjusting the positions of vertices. In many real-world applications, the positions of vertices are immovable to show the geo-location information. Nguyen et al. [80] introduced faithfulness of graph drawings, which is an important criterion that measures the degree that a graph drawing algorithm can uniquely present the raw network data. Simply, if a graph drawing algorithm is faithful, it can map different graph drawing results to distinct networks. However, Nguyen et al. indicated that it is only a semi-formal model without a precise description. In this chapter, we discuss the readability and faithfulness models and propose an information-theoretic framework to quantify the amount of information delivered by edge bundling visualization from original network data.

### 5.2.2 Studies of Information Theory in Visualization and Computer Graphics

Information theory was first introduced by Shannon to illustrate communication systems [101, 102]. Since then, information theory has been widely used in various fields. The information visualization and scientific visualization communities have been using information theory and its related concepts in many studies [89, 51, 15, 115]. Chen et al. [15, 14] adopted the model and presented an information-theoretic visualization system. Researchers have used entropy measures in viewpoint selection for time-varying volume rendering [49], vector field visualization [73], and isosurface rendering [105]. Measures of mutual information were used to select optimal viewpoints for 3D objects [39, 13]. For a
comprehensive survey of information-theoretic framework in scientific visualization, we refer readers to Wang et al.’s paper [115]. In computer graphics, an analysis of scene visibility and radiosity complexity was introduced by Feixas et al. [30]. Rigau et al. [96] leveraged the entropy of pixel color and geometry to guide pixel supersampling in stochastic raytracing. Fleishman et al. [32] proposed a viewpoint selection for modeling a 3D object from images. Gumhold [42] suggested a solution for light sources placement using information theory for image models. Vazquez et al. [111] used viewpoint entropy to select an optimal view angle for polygon models. We refer readers to Sbert et al.’s work [98] for more complete information of information theory in computer graphics. Since the concepts and details of information theory have been introduced and explained in many existing works [100, 41, 109, 56, 102, 101, 20], we only list the necessary mathematical formulas of information theory in this chapter to make the content concise.

5.3 Our Method

5.3.1 Background

Chen et al. [15, 14] presented a comprehensive information-theoretic pipeline to illustrate the visualization process using information theory, as shown in Figure 5.1. This pipeline can be generally applied to graph visualization. The pipeline shows that a graph visualization first encodes raw data, and then sends the encoded visual description, e.g., a drawing (image) via a visual channel. An observer receives the visual description and tries to decode the description for final comprehension. The general visualization pipeline has been used in many existing work. Since the knowledge about the decoder (human perception and cognition) in Figure 5.1 may require a tremendous number of user studies, we are reluctant to conduct a comprehensive study that covers the full span of the visualization pipeline.
Similar to the recent work [6], we only focus on the encoder subsystem. We discuss how the raw network data are encoded and described as an image by edge bundling algorithms and how to optimize the information transferring using Chen et al.’s pipeline and information theory.

\[ I(U; O) = H(U) - H(U|O), \]  

(5.1)

where \( H(U) \) is the total information of \( U \), and the conditional entropy \( H(U|O) \) is the amount of additional information needed to describe \( U \) given \( O \). It can also be regarded as the information loss in the encoding process. To maximize the mutual information \( I(U; O) \), we thus need to minimize \( H(U|O) \), i.e., to minimize the information loss. Figure 5.2 illustrates

Figure 5.1: A general visualization pipeline.
\(H(U), H(O)\) and \(I(U; O)\), and their relationship. Many visualization studies \([113, 72, 30, 108, 86, 53, 39, 13, 12, 6]\) have proposed solutions to maximize the mutual information to improve their visualization results.

![Figure 5.2: A Venn diagram showing the relation of entropies \(H(U)\) and \(H(O)\), and mutual information \(I(U; O)\).](image)

### 5.3.2 Uncertainty in Edge Bundling Visualizations

The pipeline of Figure 5.1 can also be applied in edge bundling visualizations. In many existing works, edge bundling algorithms are used to visualize large graphs since the algorithms help reduce visual clutter. We illustrate the advantage of edge bundling visualizations over traditional node–link diagrams in large graph visualizations. A classic example is to visualize the U.S. airline routes where domain experts want to see the airline routes between different cities with their geo-locations, as shown in Figure 5.3a. The dataset of Figure 5.3a has 2100 edges and 235 vertices. The drawing in Figure 5.3a shows a traditional node–link diagram that uses segment-based edges to encode the relations of the graph, resulting in severe visual clutter because of the edge crossings and edge overlapping. The visual clutter mainly affects human perception to track the edge between a pair of vertices. For example, observers cannot easily tell if Miami and Chicago have a connection based on the drawing.

Traditional node–link diagrams may fall short in visualizing large graphs since they do not meet some readability criteria. As mentioned in Section 5.2, there were several readability criteria to evaluate the quality of a graph drawing. To optimize two or more of the readability criteria is an NP-hard problem \([7]\). Among the criteria, edge crossing is
Figure 5.3: The comparison to find whether or not Chicago and Miami are connected using:
(a) traditional node–link diagram with a single color; (b) force-directed edge bundling visualization with a single color; (c) traditional node–link diagram with color-encoded edges; and (d) force-directed edge bundling visualization with color-encoded edges.

widely acknowledged as the most important one [88]. In graph drawings, edge crossings would cause visual ambiguity to observers. The ambiguity mainly affects human perception to identify the relations between pairs of vertices in graphs. In Figure 5.3a, the area between Miami and Chicago are occupied by many edges. Observers can hardly identify if Miami and Chicago are connected because of the visual clutter. To reduce the clutter, edge bundling techniques are often employed in the visualizations of large graphs. Edge bundling techniques mainly group similar edges to form bundles, such that the area used and edge crossing are significantly reduced. Figure 5.3b shows a force-directed edge bundling (FDEB) drawing using the U.S. airline dataset. It becomes much easier to identify the edge between Miami and Chicago compared to Figure 5.3a. Additionally, using color-encoded methods can help better identify paths and structural patterns. Color-encoded methods can
customize the transparency and color of edges in a drawing based on the attributes of the edges. Figure 5.3c shows a directional color-encoded method for the node–link diagram of the airline dataset. Comparing Figure 5.3a, c, the edges with different directions are more salient in Figure 5.3c, whereas the path between Miami and Chicago can still be hardly tracked. Using the same color-encoded method in the FDEB visualization of the same dataset, we can clearly see the blue path connecting Miami and Chicago, as shown in Figure 5.3d. The overall result of Figure 5.3d is even better than Figure 5.3b. Intuitively, we can conclude that color-encoded method can help track the relations between vertices in visualizations.

Although edge bundling techniques can improve the readability of graphs in terms of edge crossing and area used, they are not without disadvantages. Edge bundling methods visually create bundle effect to reduce edge crossings and area used, but the relationship details are thus hidden in the bundle. An example is shown in Figure 5.4. Figure 5.4c shows an edge bundling drawing. Figure 5.4b,e,f shows three possible network results that can generate the same edge bundling drawing in Figure 5.4c. This disadvantage is also discussed in several papers. Wang et al. [117] provided a visual analytic tool to show the ambiguous regions using heat map in graph drawings. They considered edge lengths, vertex and edge aggregations, and community structures; however, they did not consider the inter-bundle ambiguity, i.e., the uncertainty of two vertices from different bundles. Figure 5.4c illustrates an example about the inter-bundle condition. We can visually perceive that there are approximately two bundles in Figure 5.4c. Two examples of intra-bundle ambiguity are that the relation between vertices $v_2$ and $v_3$, and the relation between vertices $v_4$ and $v_5$ are unknown. Meanwhile, using the aforementioned method cannot identify the ambiguity between vertices $v_1$ and $v_6$ in the drawing, which corresponds to a case of inter-bundle ambiguity. Hence, we argue that the work is not sufficient to evaluate edge bundling drawings. Nguyen et al. [80] defined the uncertain presentation in an edge
bundling visualization as information loss in the edge bundling visualization by introducing information faithfulness. A visualization is information faithful if the visualization can uniquely represent the original graph. In their paper, they concluded that edge bundling visualization is inherently not information faithful and stated that it will be increasingly difficult for users to perceive the original network from an edge bundling visualization when more edges are bundled together. They gave a model to illustrate this situation. Given a graph $G = (V, E)$, the edge bundling visualization partitions the edges $E$ into $K$ bundles $E = B_1 \cup \ldots \cup B_i \cup \ldots \cup B_k$. Let $G_i$ presents a subgraph of $G$ that consists of only $B_i$. $G_i$ is essentially a bipartite graph where the set of vertices are $V_i$ and the set of links are $B_i$. For any two subgraphs $G_i$ and $G_j$, $B_i \cap B_j = \emptyset$. According to the definition of bipartite graph, we have $V_i = P_i \cup Q_i$, $P_i \cap Q_i = \emptyset$, where $P_i$ is the source vertices and $Q_i$ is the sink vertices in the bipartite graph $G_i$. Enumerating the bipartite graphs with all possibilities gives $2^{|P||Q|}$ combinations. The number of graphs that have the same link structure as the final edge bundling drawing of $G$ is $\prod_k 2^{|P||Q|}$, which means there are $\prod_k 2^{|P||Q|}$ different original networks. However, we argue the $2^{|P||Q|}$ different ways that a bundle may have is just a loose upper bound. The tight bound requires further investigation. On the other hand, it is hard to define an exact number of bundles in a drawing. Additionally, their work also did not consider the inter-bundle uncertainty, as they assumed $B_i \cap B_j = \emptyset$, which often is not held in practice. Thereby, to evaluate the quality and goodness of edge bundling visualizations is still an open problem. In Section 5.3.3, we introduce a formal information-theoretic metric to evaluate the drawing result of edge bundling techniques.
5.3.3 An Information-Theoretic Metric for Edge Bundling Visualizations

We introduce a general model to quantify the uncertainty delivered by graph visualizations. We define the objects of a graph as nodes or vertices, and the relationships among objects as links or edges. Take a graph $G = (V, E)$ where there are $|V|$ vertices and $|E|$ edges. In our study, we only consider simple paths in graph structures, and represent $G$ as an adjacency
matrix $A$:

\[
A_{ij} = \begin{cases} 
1 & \text{there is an edge } e_{ij} \text{ between two vertices } i \text{ and } j \\
\text{n/a} & i = j \\
0 & \text{otherwise}
\end{cases}
\] \hspace{1cm} (5.2)

Let $D(G)$ denote a graph drawing of $G$, the edges and vertices of $G$ are encoded by visual symbols (such as segments, curves, polylines, labels, points, etc.) with colors in $D(G)$. Based on the pipeline of Figure 5.1, a graph drawing method has a visual encoding process that transforms the underlying network relations and structure into visual symbols. Color mapping functions are also used in the process. The encoder process outputs a visual description, i.e., $D(G)$. Observers need to observe $D(G)$ in order to guess the value of $A_{ij}$ of $G$. Visually, $D(G)$ presents an adjacency matrix $A^{D(G)}$ indicating the relations among the vertices in the drawing:

\[
A_{ij}^{D(G)} = \begin{cases} 
N(N \subset \mathbb{N}) & \text{there are } N \text{ edges between two vertices } i \text{ and } j \\
\text{n/a} & i = j \\
0 & \text{otherwise}
\end{cases}
\] \hspace{1cm} (5.3)

To understand the relations of the underlying network, observers need to observe $D(G)$, and guess the value of $A_{ij}$ based on the value of $A_{ij}^{D(G)}$.

Figure 5.4 shows an example. Figure 5.4a shows an adjacency matrix $A$ of a graph $G$. In Figure 5.4b, a node–link drawing $D_n(G)$ correctly reveals the relations among vertices with the least ambiguity for this simple graph. In Figure 5.4c, an edge bundling drawing $D_e(G)$ encodes the edges with curves. One observation is that there seems to be an edge between the vertices $v_1$ and $v_2$. One possible reason is that there indeed is an edge between $v_1$ and $v_2$. However, it is also possible that there is no edge between $v_1$ and $v_2$, but an edge between $v_1$
and \( v_6 \) and an edge between \( v_2 \) and \( v_3 \), and these two edges are bundled together, causing an illusion edge between \( v_1 \) and \( v_2 \). Hence, the ambiguity arises that the relation between \( v_1 \) and \( v_2 \) is uncertain in Figure 5.4c. Note that even if we only consider simple paths in the visualization of a graph \( G \), \( D_e(G) \) may inadvertently have multiple edges between a pair of vertices. By using certain intuitive criteria, e.g. readable bendiness of curves (Section 5.3.4), we can guess that there is no edge among \( v_1, v_3, \) and \( v_5 \) in \( D_e(G) \). The same intuitiveness can be also applied to \( v_2, v_4, \) and \( v_6 \). However, all other relations among the vertices remain uncertain in \( D_e(G) \). We can use Equation (5.3) to construct an adjacency matrix \( A^{D_e(G)} \), as shown in Figure 5.4d. If an entry \( A_{ij}^{D_e(G)} > 0 \), we are not sure if there would be an edge between \( i \) and \( j \) in the original graph \( G \), and possibly drive multiple interpretations, such as Figure 5.4e,f, that can generate the same edge bundling draw in Figure 5.4c.

To assess an edge bundling drawing \( D_e(G) \), based on \( A \) and \( A^{D_e(G)} \), we first introduce a coverage rate \( \lambda \) to evaluate the percentage of how many edges in \( A \) are covered by \( A^{D_e(G)} \). The idea is intuitive: we want to know how many edges in an original network are presented in a corresponding drawing. In our definition, we only require the drawing to show at least an edge between two vertices (i.e., \( A_{ij}^{D_e(G)} > 0 \)) if the two vertices do have an edge in the underlying network (i.e., \( A_{ij} = 1 \)). Equation (5.4) expresses this idea as (the n/a entries of \( A \) and \( A^{D(G)} \) do not enter the computation of the following equations):

\[
\lambda = \frac{\sum_{i=1}^{m} \sum_{j=1}^{m} \mu(i, j)}{|E|},
\]

(5.4)

where \( m \) is the number of vertices in \( G \), and \( \mu(i, j) \) is a simple Heaviside step function:

\[
\mu(i, j) = \begin{cases} 
1 & \text{if } A_{ij} = 1 \text{ and } A_{ij}^{D_e(G)} > 0 \\
0 & \text{otherwise}
\end{cases}
\]

(5.5)
The higher the value of $\lambda$ is, higher the coverage is. If $\lambda = 1$, we say the corresponding $A^{D}(G)$ is saturated. However, only using Equation (5.4) cannot assess an edge bundling drawing effectively. For example, although Figure 5.4b,c covers the matrix $A$ in Figure 5.4a, the degrees of uncertainty are significantly different in Figure 5.4b,c. Hence, we need to introduce another metric to evaluate the uncertainty of edge bundling drawing results.

We provide an information-theoretic model to quantify the uncertainty of the above situation. Our information-theoretic model first assumes that graph drawing and visualization algorithms do not intend to underdraw a graph, i.e., given a graph $G$, graph drawing and visualization algorithms do not intend to show a wrong value of $A_{ij}$ in $A_{ij}^{D(G)}$. If $A_{ij} = 0$ in $G$, the encoder of visualization process should always intend to show $A_{ij}^{D(G)} = 0$ in $D(G)$, such that the observers may guess that $A_{ij} = 0$ in $G$, and vice versa. In addition, if $A_{ij}^{D(G)} = N(N \subset \mathbb{N}, N \geq 1)$ in $D(G)$, it becomes more uncertain to determine whether $A_{ij} = 1$ since $i$ and $j$ may overlap some edges that are not between $i$ and $j$ in $G$. Figure 5.4c shows an example. We know that even if only simple paths are allowed between a pair of vertices, a drawing result may still have multiple paths between this pair of vertices, which makes the visual description uncertain. We also define that the edge in Equation (5.2) is just a relation while the edge in Equation (5.3) can be a segment, curve, or polyline in the drawing.

We denote the relation between two vertices as a random variable $X$. A graph drawing or visualization algorithm fully understands the graph and tries to encode the relation with visual symbols and colors. After the encoding process, the algorithm outputs an image, i.e., a drawing $D(G)$. $D(G)$ provides a result that indicates the original relation of the two vertices. The visual result can be represented by an adjacency matrix $A^{D(G)}$ based on Equation (5.3). Given two vertices $i$ and $j$, we can quantify the amount of uncertainty of the relation of vertex $i$ and $j$ given $D(G)$ using information theory. Generally, if $A_{ij}^{D(G)} = N(N \subset \mathbb{N})$, there are $N$ paths between the vertices $i$ and $j$ in the drawing $D(G)$. Since we only consider simple
paths, one of the \( N \) paths may be encoded as an edge to present \( A_{ij} = 1 \) in \( G \). Another possibility is \( A_{ij} = 0 \), which means there is no edge between the vertices \( i \) and \( j \) in the original graph, but \( i \) and \( j \) overlap \( N \) other edges in the drawing \( D(G) \). Therefore, \( D(G) \) can have \( N + 1 \) different ways to present \( A_{ij} \) (i.e., \( N \) possible paths or no path).

Let \( Y \) denote the visual description of the relation between a pair of vertices. The amount of uncertainty of knowing the relation between the vertices \( i \) and \( j \) given \( D(G) \), i.e., the conditional entropy \( H(X_{ij}|Y_{ij}) \), can be quantified as:

\[
H(X_{ij}|Y_{ij}) = \sum_{t=0}^{A_{ij}^{D(G)}} -\left( \frac{1}{A_{ij}^{D(G)} + 1} \right) \log_2 \left( \frac{1}{A_{ij}^{D(G)} + 1} \right)
\]

\[
= \log_2 (A_{ij}^{D(G)} + 1),
\]

where \( A_{ij}^{D(G)} \) is the value of the \( i \) column and \( j \) row entry of \( A^{D(G)} \). Equation (5.6) indicates the necessary bits to visually describe the relation of the corresponding vertices \( i \) and \( j \). The more bit the visual description uses, the more uncertain the description is.

We use the graph drawings of two simple graphs \( G_1 \) and \( G_2 \) in Figure 5.5 to illustrate Equation (5.6). As shown in Figure 5.5a, visually, there are two paths, \( p_1 \) and \( p_2 \), between the vertices \( v_3 \) and \( v_4 \), and thus \( A_{3,4}^{D(G_1)} = 2 \). As we consider simple paths, there are three possible cases of connection between \( v_3 \) and \( v_4 \) in the original graph \( G_1 \): \( p_1 \), \( p_2 \), or no path. The probability of each case is \( \frac{1}{3} \). Therefore, given the visual description \( Y_{3,4} \), we can compute the amount of uncertainty to describe the real relation \( X_{3,4} \) as \( \sum_{t=0}^{2} -(\frac{1}{3})log_2(\frac{1}{3}) = log_2(3) \approx 1.58 \). Similarly, there are three paths between the vertices \( v_4 \) and \( v_5 \) of \( G_2 \) in its drawing \( D(G_2) \), as shown in Figure 5.5b. This leads to a higher amount of uncertainty, \( \sum_{t=0}^{3} -(\frac{1}{4})log_2(\frac{1}{4}) = log_2(4) = 2 \), for us to tell the real relation \( X_{4,5} \) of \( G_2 \).
We denote the total uncertainty of $D(G)$ as $W$. $H(W)$ can be formally written as:

$$H(W) = \sum_{i=1}^{m} \sum_{j=1}^{m} H(X_{ij}|Y_{ij}),$$

(5.7)

where $H(X_{ij}|Y_{ij})$ represents the amount of uncertainty of knowing $X_{ij}$ based on $Y_{ij}$. It can also be interpreted as how much information about $X_{ij}$ is still uncertain after observing $Y_{ij}$. As discussed in Section 5.3.1, the best description $Y$ should tell the most of $X$ (i.e., to maximize the mutual information $I(X;Y)$, we need to minimize the conditional entropy $H(X|Y)$). Hence, to holistically evaluate an edge bundling drawing, we argue that a good edge bundling visualization should minimize $H(W)$, and, at the same time, keep the coverage rate $\lambda$ as high as possible. For a undirected graph $G$, $A$ and $A^{D(G)}$ are symmetric, and thus only the upper right half of each matrix is used.

![Figure 5.5: The graph drawings of two simple graphs $G_1$ and $G_2$. (a) A simple path graph $G_1$, where there are visually 2 paths between $v_3$ and $v_4$. (b) A simple path graph $G_2$, where there are visually 3 paths between $v_4$ and $v_5.$](image)

We use Equation (5.7) to quantify the values of $H(W)$ of Figure 5.4b,c. On one hand, as Figure 5.4b shows, the relations among objects are clear and correct, and the amount of uncertainty of the corresponding $H(W)$ is 3. Figure 5.4b can use the least bits to describe a network since each edge can be distinguished in this drawing. On the other hand, the
amount of uncertainty of $H(W)$ of Figure 5.4c is 8. This comparison matches the results of the drawings.

Generally, Equation (5.6) can also be used to quantify the amount of uncertainty of relation between a pair of vertices in an edge bundling drawing $D(G)$. Figure 5.3a shows a more complex example. In the figure, we want to quantify the amount of uncertainty of relation between Miami and Chicago. Equation (5.6) counts the number of edges (paths) between the cities in the drawing. Recall that the paths can be segments, curves, or polylines in our definition. We can find multiple such paths between Miami and Chicago. In Figure 5.3b, the paths between Miami and Chicago are significantly reduced because of the bundle effect. The rationale behind Equation (5.6) is that, if more paths can be detected between the two vertices, the used area between the two vertices becomes larger. This reflects that a larger number of edge crossings and overlapping, which means the visual description of the relation between the two vertices is more uncertain. The greater the value $A_{ij}^{D(G)}$, the more uncertain the description is. In a general case, no matter $A_{ij} = 0$ or 1. If the number of paths between the two vertices is larger than one, i.e., $A_{ij}^{D(G)} > 1$, the visual description of the relation of the two vertices is uncertain. In Section 5.3.4, we introduce a method to count the number of paths between a pair of vertices based on the drawing.

5.3.4 Algorithms and Implementation

We introduce an algorithm to approximate the number of edges or paths between a pair of vertices in a drawing. Our algorithm mimics an observer’s perception to track the edges or paths between two vertices. As it would be tedious for an observer to manually count all paths in a drawing of a large graph, we propose a computational method for this task. We also use a heuristic method to discuss the parameters in our algorithm in Section 5.4. As described in Section 5.3.3, a path between two vertices in a drawing can be segments,
spline-curve, polylines, or a hybrid presentation of the above three. Generally, a qualified path between two vertices should meet two criteria: (1) the bendiness of the path should be reasonable; and (2) the color of the path should be similar. The bendiness criteria ensures that a qualified path have a reasonable smoothness and do not contain loops and abrupt turning angles, while the color criteria ensures that the color along the path is similar. Complying the two criteria, a path can be identified and tracked by observers. Although many studies have proposed path and road location and detection in remote sensing and image processing fields, approximating the number of paths between two vertices is a unique and non-trivial task in this study.

To approximate the number of qualified paths between two vertices, we need to find the region in the image connecting them. We first locate the pixel positions of a pair of vertices in an edge bundling drawing (image). Then, starting from one of the vertices, we conduct a region growing method to find a piece of region that connects the two vertices. We design two parameters in our region growing method to comply with the aforementioned criteria. Generally, given a drawing result, which is an image $I$ with a resolution of $M \times N$, we first locate the pixel positions of the two vertices in the image, and then use Algorithm 10 to find the number of paths between the two vertices. Assume the start pixel is $P_s$ and the target pixel is $P_t$. We specify a color threshold $C$ and an angle threshold $L$. The target region $R$ contains only $P_s$ initially. We start from the current pixel $P_c = P_s$, and search through all the neighboring pixels $\{P_n\}$ in a $W_1 \times W_1$ window, where $W_1$ is the size of the window. We need to find all the neighboring pixels $\{P_n\}$ that meet three conditions: (1) the angle between the vector $\overrightarrow{P_cP_t}$ and the vector $\overrightarrow{P_cP_n}$ is not greater than $L$; (2) the angle between the vector $\overrightarrow{P_sP_t}$ and the vector $\overrightarrow{P_cP_t}$ is not greater than $L$; and (3) the Euclidean distance between the color of $P_c$ and the mean color $C_m$ of the region is not greater than $C$. Conditions (1) and (2) ensure that the region will be growing from the start vertex towards the target vertex with a specified angle limitation. $L$ determines the sharpness of the paths in the region. Condition
(3) simply ensures the color criterion. The qualified pixels that have not been visited are added to a candidate set. We then set the pixel that is closest to $C_m$ to be the new current point $P_c$ and add $P_c$ to the region $R$. The process continues until $P_c = P_t$ or the candidate set is empty. The region growing algorithm is illustrated in Algorithm 11.

Figure 5.6a shows a magnified and highlighted area of a FFTEB visualization. We want to find the region connecting two vertices $a$ and $b$. Figure 5.6a.1 is the output of Algorithm 11. It shows that the region between $a$ and $b$ can be perfectly extracted. Another more complex example is shown in Figure 5.6b.1,c.1 presenting the impact of the input parameters $C$ and $L$. In the highlighted area, we want to find the region connecting the vertices $c$ and $d$. Using $C = 100$ and $L = 90$, we get the result of Figure 5.6b.1. Using $C = 190$ and $L = 150$, we have the resulting region of Figure 5.6c.1. The difference is obvious. In Figure 5.6b.1, we do not have the big hole in the middle of the region because the input color threshold $C$ and angle threshold $L$ are relatively small. $C$ controls the acceptable color difference between candidate pixels and the region, while $L$ determines that the sharpness of a portion of the region. They are very important parameters in our algorithm. The window size $W_1 = 1$ or 2 (1 or 2 pixel(s)) can generate very good results. However, $C$ and $L$ may impact the grown region largely, as in Figure 5.6b.1,c.1. We further discuss $C$ and $L$ in a heuristic study in Section 5.4.
Algorithm 10 FINDALLPATHS.
1: // Initialization
2: \(P_s\) // The start pixel
3: \(P_t\) // The target pixel
4: \(W_1\) // The size of sliding window for Algorithm 11
5: \(W_2\) // The size of sliding window for Algorithm 12
6: \(C\) // The color threshold
7: \(I\) // The \(M \times N\) image
8: \(R\) // The growing region
9: \(K\) // The clusters
10: \(P\) // The number of paths
11: \(N\) // The number of node in graph
12: \(VISITED[N]\) // The flag array that indicates if vertices are visited
13: Find the source pixel \(P_s\) and target pixel \(P_t\).
14: // Given \(I, W_1\) and \(C\), use region growing to find the region \(R\) connects \(P_s\) and \(P_t\)
15: \(R \leftarrow \text{REGIONGROWING}(I, P_s, P_t, W_1, C)\)
16: // Given the region \(R\), use mean shift to calculate the clusters \(K\)
17: \(K \leftarrow \text{MEANSHIFT}(R, W_2)\)
18: Find the number of vertices \(N\) based on the separate components of \(K\).
19: // Based on the clusters \(K\), find the source region \(R_s\) and the target region \(R_t\)
20: \(P \leftarrow \text{DEPTH-FIRSTSEARCH}(P, K, R_s, R_t, VISITED[R_s])\)
Algorithm 11 REGIONGROWING($I$: input_image; $P_s$: input_source_pixel; $P_t$: input_target_pixel; $W_1$: input_window_size; $C$: input_color_threshold; $L$: input_angle_threshold).

1: Assign the color of $P_s$ to $C_m$.
2: $R$ // The growing region
3: $C_m$ // The mean color of the growing region
4: $P_c \leftarrow P_s$ // Assign the source pixel to be the current pixel
5: $S \leftarrow \emptyset$ // Initialize the candidates set
6: Push $P_c$ into $R$.
7: while $P_c \neq P_t$ or $S \neq \emptyset$ do
8: for each neighboring pixel $P_n$ of $P_c$ using the window size $W_1$ do
9: if the angle $\theta_1$ between $\overrightarrow{P_cP_t}$ and $\overrightarrow{P_cP_n} \leq L$ and the angle $\theta_2$ between $\overrightarrow{P_tP_c}$ and $\overrightarrow{P_cP_t} \leq L$ and the color of $P_c - C_m \leq C$ then
10: Push $P_n$ into $S$.
11: end if
12: end for
13: // Compute the next $P_c$
14: Compute the pixel in $S$ whose color is closest to $C_m$, and assign the pixel to $P_c$.
15: Compute the mean color of $S$, and assign the mean color to $C_m$.
16: Pop $P_c$ from $S$.
17: Push $P_c$ into $R$.
18: end while
19: return $R$. 
After the region $R$ is gained, we can consider how to find the number of paths between two pixels. Here, the problem is typically a graph problem that finding the number of paths between a source vertex and a target vertex in a graph where every pixel in $R$ can be modeled as a vertex, and the connectivity of pixels can be modeled as edges. However, estimating the number of source-to-target paths in a graph is #P-complete [97]. To approximate the
number of source-to-target paths, we could use depth-first search to enumerate all the unique
paths from a source to a target, or a dynamic programming to statistically calculate the total
number of unique paths from a source to a target. However, in our experiment, we found
out that simply using the above methods is problematic in the applications of edge bundling
visualization. For example, in Figure 5.6a.1, intuitively, there should be one path connecting
a and b. However, we can show that a simple depth-first search will generate an incorrect
(significantly large) number of paths for Figure 5.6a.1.

Figure 5.7 shows an example to approximate the number of paths between a source
g pixel \( P_s \) and a target pixel \( P_t \). Every square in Figure 5.7a represents a pixel. All pixels
in Figure 5.7a show a connecting region. To approximate the number of paths between
the source pixel and the target pixel, we can transform Figure 5.7a into a graph, as shown
in Figure 5.7b. We limit the traverse direction of every pixel from going back in order to
avoid loop paths. Traversal methods, such as depth-first search, can be applied in order to
enumerate the unique paths between the two pixels. These traversal methods can generate
far more number of unique paths. However, visually, there is only one path between the
two pixels in Figure 5.7a, as the region can be considered as a path as a whole. Similarly,
the traditional dynamic programming method cannot solve this problem either. Assume
we define a subproblem as how many source-sink paths are through the current pixel.
The dynamic programming method is not effective because the subproblem is not solved
only once. Hence, depth-first search and dynamic programming cannot appropriately solve
this type of problem.
Figure 5.7: An example to approximate the number of paths between a source pixel $P_s$ and a target pixel $P_t$: (a) the connecting region between $P_s$ and $P_t$; and (b) the transform graph.

Additionally, simply modeling every pixel as a vertex will make the computation time-consuming since a resulting region could consist of a considerable number of pixels. Another problem is that neither the simple depth-first search nor dynamic programming cannot solve the problem that the small holes in the generated region, which is illustrated in Figure 5.6c.2. For instance, in Figure 5.6c.2, intuitively, there is only one path from $c$ to $d$, whereas the small holes can generate unnecessary loop paths, which should be addressed.

We propose to use a simple mean shift to cluster the resulting region, then model the clusters into vertices, and conduct a modified depth-first search to approximately find the number of paths between two vertices in a region $R$. After the region $R$ is found, we check that if the two vertices are in the region. If so, we use mean shift to cluster the region $R$. Otherwise, we conclude there is no path between the two vertices. The basic idea is that we first use mean shift to cluster $R$ into distinct regions. Second, we construct a transform graph $T$ that shows the connectivity of the distinct regions. We define the source region $R_s$ containing the pixel $P_s$, and the target region $R_t$ containing the pixel $P_t$. Finally, we use a modified depth-first search to calculate the number of paths between $R_s$ and $R_t$ in $T$.

The mean shift algorithm takes $R$ as input. For each pixel, we define a window with a size $W_2$ around it and compute the mean of the pixels that have some color other than the background color. Then, we shift the center of window to the mean and assign the new position to the current pixel. We repeat this process until all pixels converge or the
iterations exceed a certain amount of times. The simple mean shift algorithm is illustrated in Algorithm 12. Then, we consider the non-connected regions as distinct clusters. The distinct clusters can be considered as a graph $T$, where each cluster can be considered as a vertex in $T$. The connectivity of vertices in $T$ is determined by the connectivity of the pixels. For example, if two pixels from two different clusters are neighbors, the two clusters have an edge. The output results of Algorithm 12 is demonstrated in Figure 5.6a.3–c.3. Figure 5.6a.4–c.4 shows the corresponding transform graphs of Figure 5.6a.3–c.3, respectively. In the three graphs, different colors mean distinct cluster labels. The results in Figure 5.6a.4–c.4 avoid the problem of path-finding in Figure 5.7. In Figure 5.6b.3,c.3, the small hole problem is also addressed, where the hole is too small to be considered a branch or another path. However, if a hole is big enough, it can be considered as a branch or path, which is shown in Figure 5.6c.3. The window size $W_2$ determines the acceptable hole threshold. We find that it should be set to only 1 or 2 pixel(s), and Figure 5.6b.3,c.3 demonstrates the results.
Algorithm 12 MEANSHIFT($R : input\_region$; $W_2 : input\_window\_size$).

1: $K$ // the cluster result
2: $P_c$ // The position of the current pixel
3: $S$ // The temporal set
4: $ITR$ // The iteration number
5: $STOP$ // The flag that indicates all pixels do not move in the last iteration
6: $STOP \leftarrow False$
7: while $ITR < 300$ and $STOP = False$ do
8: \hspace{1em} for each pixel $P_c$ of $R$ do
9: \hspace{2em} $S \leftarrow \emptyset$
10: \hspace{2em} for each neighboring pixel $P_n$ of $P_c$ using the window size $W_2$ do
11: \hspace{3em} if the color of $P_c$ does not equal to the background color then
12: \hspace{4em} Push $P_n$ into $S$.
13: \hspace{3em} end if
14: \hspace{2em} end for
15: \hspace{1em} Compute the new position for $P_c$ based on $S$.
16: \hspace{1em} end for
17: // Check if some of the pixels have new positions
18: \hspace{1em} if none of the pixels in $R$ moves then $STOP \leftarrow True$
19: \hspace{1em} end if
20: end while
21: Give every separate component a distinct number, and assign the result to $K$.
22: return $K$.

Finally, a modified depth-first search algorithm is used to count all possible paths between $R_s$ and $R_t$ in $T$. In this algorithm, it first sets the flag of every vertex to be unvisited. The algorithm starts from the source region $R_s$, and find the adjacent regions in a depth-first search manner until it reaches the target region $R_t$. Every time the algorithm reaches a new
region, it sets the flag of the region to be visited. Hence, it will not form a loop path in \( T \). If \( R_t \) is reached, the counter of all possible paths increments one. The modification from the traditional depth-first search algorithm is that after a region is visited, we reset the flag of the current region to be unvisited, making this region available to other paths. Finally, if all other regions are visited, the algorithm ends. The modified depth-first search is illustrated in Algorithm 13.

**Algorithm 13**  
**DEPTH-FIRST SEARCH** *(P: (GLOBAL) input_path_number; K: input_clusters; \( R_c \): input_current_region; \( R_t \): input_target_region; VISITED[]): input_source_flag).*

1. \( P // \) The number of path between \( R_s \) and \( R_t \)
2. \( VISITED[R_c] \leftarrow \text{True} \)
3. **if** \( R_c = R_t \) **then**
4. \( P \leftarrow P + 1 \)
5. **else**
6. **for** each adjacent region \( R_n \) of \( R_c \) **do**
7. **if** \( VISITED[R_n] = \text{False} \) **then** **DEPTH-FIRST SEARCH** *(P, K, R_n, R_t, VISITED[R_n])*
8. **end if**
9. **end for**
10. **end if**
11. \( VISITED[R_c] \leftarrow \text{False} \)

### 5.4 Application Examples

We provide several examples to show how to use the proposed metric and approximation method to evaluate and compare edge bundling algorithms. The edge bundling algorithms we used in this section include the force-directed edge bundling (FDEB) [45], the fast fourier transform edge bundling (FFTEB) [63], and the moving least squares edge bundling
The three methods cover three different edge bundling frameworks: force-directed, kernel estimation, and curve approximation. The three frameworks cover most of the edge bundling methods. Although only three methods are chosen to compare in this chapter, our metric and method can be easily extended to other edge bundling methods. First, we briefly revisit the algorithms of FDEB, FFTEB, and MLSEB, respectively. Second, we conduct a heuristic study to discuss the color and angle thresholds in our path-finding method. Third, we show an example to use the proposed information-theoretic metric and path-finding method by exploring the parameters of the three edge bundling methods on a dataset. We discuss how users of edge bundling applications should choose the appropriate parameters of the edge bundling methods for their applications based on the analysis of the corresponding uncertainty $H(W)$ and coverage $\lambda$. Fourth, we use different datasets to illustrate the similarities and differences among the three different edge bundling methods based on the analysis of the uncertainty $H(W)$ and coverage $\lambda$ and the corresponding visualizations.

In the following results, we render all edge bundling drawings into $800 \times 800$ images. The background color of these drawings is set to be (0, 0, 0, 0) in terms of RGBA value. We use a color-encoded method [110] in all renderings. The color-encoded method encodes the direction and the length of the corresponding edge with HSV value (i.e., hue $H$, saturation $S$, value $V$, and alpha $A$). Using this method, short edges are better visible, and at the same time long edges can also attract attention. Additionally, since the value of $H(W)$ can be large for some large graphs, we multiply $H(W)$ with a factor $\frac{1}{p}$ for a better displaying (i.e., $H(W) = H(W) \times \frac{1}{p}$), where $p$ is the number of all pairs of different nodes in the corresponding graph.
5.4.1 Revisit FDEB, FFTEB, and MLSEB

5.4.1.1 FDEB

FDEB uses a spring model to bundle edges in a graph drawing. It first employs similarity, namely compatibility measures to find the similar edges in a graph based on four spacial features (i.e., direction, length, position, and projection). The result of the compatibility measures is a soft clustering, which means one edge may belong to multiple groups. In every iteration of the bundling process, edges are subdivided into sample points. In the subdivision process, every edge is subdivided into identical number of sample points. A spring model is then applied in the subdivision points of edges to advect the edges in an iterative manner. One edge only interacts with its compatible edges. Hence, edges are bundled in a group manner, as shown in Figure 5.8b. The final image is generated when the spring system reaches an equilibrium state. One unique parameter in FDEB is the compatibility threshold used to determine how similar two edges are. Hence, it also determines the clustering results and the final edge bundling drawing.

5.4.1.2 FFTEB

FFTEB uses fast fourier transform to accelerate large-graph bundling process. The basic idea of FFTEB is from the kernel density estimation edge bundling (KDEEB) [47]. KDEEB uses a uniform sampling method, where edges with different lengths have different numbers of sample points. KDEEB transforms an input graph into a density map using kernel density estimation, and then moves the sample points of edges towards the local density maxima to form bundles. The sample points of a single edge could belong to different bundles since the sample points are advected locally. FFTEB is an enhanced method of KDEEB. It shifts the bundling process from the image space to the spectral (frequency) space, thereby increasing computational speed. They address the performance issue by using GPU processing. An
important parameter used in FFTEB is the window size of the kernel density estimation. The window size of the kernel density estimation determines the location of local density maxima in the drawing.

### 5.4.1.3 MLSEB

MLSEB generates bundles based on moving least squares (MLS) approximation. The basic idea of MLSEB comes from reconstructing a smooth curve from an unorganized point cloud. It first discretizes all edges into a set of sample points. Then, it uses moving least squares approximation to reconstruct a smooth curve based on locally close sample points with a window size. The close sample points of edges will be projected onto the local regression curve iteratively. After each iteration, the sample points of different paths locally gather closer. Using the advected sample points of each path as control points, a B-spline curve is generated for each edge. The curve-like bundles are thus formed. The moving least squares method also gives a result that one edge may belong to different bundles, as shown in Figure 5.8d. MLSEB also uses the same uniform sampling for each edge as FFTEB. The important parameter is the compact support (i.e., the window size) of the moving least squares approximation.

### 5.4.2 Heuristic Study

In this section, we conduct a heuristic study to discuss the color and angle thresholds in our proposed method for different datasets. The datasets we used in this section are the U.S. airlines dataset with 2100 edges and 235 vertices, and the U.S. migrations dataset with 9780 edges and 1700 vertices.

Since we employ the algorithms in Section 5.3.4 to mimic an observer’s perception to track paths between two vertices in a graph drawing, we conduct a heuristic study to discuss
Figure 5.8: Visualizations of different datasets with different methods.
the parameters, the color threshold $C$ and the angle threshold $L$. The rationale of this heuristic study is to choose the optimal thresholds for our evaluation algorithm. In the heuristic study, we compare the coverage and the uncertainty of the U.S. airlines using Figure 5.8b–d, as well as the U.S. migrations data using Figure 5.8f–h with different values of color threshold and angle threshold. The values of color threshold we used are 100, 120, 150, 170, and 190. The values of angle threshold we used are $45^\circ$, $90^\circ$ and $135^\circ$. The compatibility of the FDEB drawings in Figure 5.8b,f, is 0.7. The kernel size of the FFTEB drawings in Figure 5.8c,g,i is 5%. The compact support of the MLSEB drawings in Figure 5.8d,h,j is 5%. In Table 5.1, we list the coverage and uncertainty values for the corresponding combination of color and angle thresholds. $c_{100,a45}$ in the table means $C = 100$ and $L = 45^\circ$, etc. In Table 5.1, we can see that, as $C$ and $L$ increase, the coverage and uncertainty increase, which matches our expectation that higher $C$ and $L$ will give a higher coverage and uncertainty. One important point is that, if we use the same color threshold and different angle thresholds, the corresponding gradients of coverage and uncertainty between $L = 90^\circ$ and $L = 135^\circ$ are significantly smaller than the gradients of $L = 45^\circ$ and $L = 90^\circ$ for all results. For example, for FFTEB, the coverage and uncertainty differences between $c_{100,a45}$ and $c_{100,a90}$ are significantly smaller than the differences between $c_{100,a90}$ and $c_{100,a135}$. We find that this phenomenon exists for all edge bundling methods we tested. Hence, we conclude that $90^\circ$ can be used as an appropriate angle threshold. Choosing $90^\circ$ as the angle threshold can ensure the path in the growing region will not turn abruptly. One example between small and large angle thresholds are shown in Figure 5.6b.1,c.1. On the other hand, for the color threshold, we can know that, when the color threshold reaches 190, the coverage of all methods become almost 1. Therefore, in the following comparison, we list different color thresholds smaller than 190 with the same angle threshold $L = 90^\circ$. Since we do not have an optimal color threshold, we compute the values of coverage and uncertainty using different color thresholds. We compute the average sum of all the color thresholds of the
corresponding coverage and uncertainty in order to obtain succinct statistic results.

Table 5.1: A heuristic study to choose color and angle thresholds.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>FDEB</th>
<th>FFTEB</th>
<th>MLSEB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>H(W)</td>
<td>λ</td>
<td>H(W)</td>
</tr>
<tr>
<td>c100, φ45</td>
<td>0.661</td>
<td>0.795</td>
<td>0.791</td>
</tr>
<tr>
<td>c100, φ90</td>
<td>0.751</td>
<td>0.817</td>
<td>0.856</td>
</tr>
<tr>
<td>c100, φ135</td>
<td>0.793</td>
<td>0.830</td>
<td>0.870</td>
</tr>
<tr>
<td>c120, φ45</td>
<td>0.838</td>
<td>0.944</td>
<td>0.846</td>
</tr>
<tr>
<td>c120, φ90</td>
<td>0.907</td>
<td>0.972</td>
<td>0.888</td>
</tr>
<tr>
<td>c120, φ135</td>
<td>0.945</td>
<td>0.975</td>
<td>0.893</td>
</tr>
<tr>
<td>c150, φ45</td>
<td>0.893</td>
<td>0.988</td>
<td>0.873</td>
</tr>
<tr>
<td>c150, φ90</td>
<td>0.926</td>
<td>0.997</td>
<td>0.911</td>
</tr>
<tr>
<td>c150, φ135</td>
<td>0.961</td>
<td>0.997</td>
<td>0.928</td>
</tr>
<tr>
<td>c170, φ45</td>
<td>0.909</td>
<td>0.996</td>
<td>0.883</td>
</tr>
<tr>
<td>c170, φ90</td>
<td>0.934</td>
<td>0.998</td>
<td>0.940</td>
</tr>
<tr>
<td>c170, φ135</td>
<td>0.965</td>
<td>0.998</td>
<td>0.952</td>
</tr>
<tr>
<td>c190, φ45</td>
<td>0.922</td>
<td>0.999</td>
<td>0.938</td>
</tr>
<tr>
<td>c190, φ90</td>
<td>0.945</td>
<td>0.999</td>
<td>0.954</td>
</tr>
<tr>
<td>c190, φ135</td>
<td>0.976</td>
<td>0.999</td>
<td>0.967</td>
</tr>
</tbody>
</table>

5.4.3 Comparison I

In this section, we evaluate the coverage and uncertainty of the three methods with different parameters. We use an example to show how to use the proposed metric and approximation method to choose an appropriate parameter for a specific edge bundling application. The dataset we use in this section is the U.S. airlines. To explore the parameter combinations
of edge bundling methods is a tremendous work since many parameters and factors can be
taken into account, such as the color-encoded method, the number of sampling points of
every edge, the unique parameter of a method, and so on. These parameter combinations
vary from case to case. It is not feasible to list many parameter combinations. Hence, we
rather use an simple instance to illustrate how users can employ our proposed metric and
path-finding method to choose parameters for edge bundling applications. Recall that, in
Section 5.4.1, we mentioned the unique and important parameters for the three methods
respectively, which are the compatibility threshold for FDEB, the kernel size for FFTEB,
and the compact support for MLSEB. We use a bivariate analysis to analyze the results of the
above methods with their different corresponding parameters. We focus on analyzing these
parameters in this chapter. Our analysis can be easily extended to more complex parameter
configurations for other edge bundling methods.

First, we discuss the parameters and configurations used in this comparison. We use four
different compatibility thresholds for FDEB. Remember that the compatibility threshold
of FDEB corresponds to the soft clustering result that determines the final graph layout. In
Figure 5.9a–d, we set the compatibility thresholds to be 0.8, 0.7, 0.6, and 0.5, respectively.
For FFTEB, the most important parameter is the kernel size of the density estimation, which
determines how the sample points of edges coverage. We set the kernel size of FFTEB to be
1%, 3%, 5%, and 10% of the image size, as shown in Figure 5.10. For MLSEB, the most
significant parameter is the compact support of the moving least squares approximation. It
is essentially a window size to estimate the weights of neighboring sample points that are
used to approximate a local regression curve. We set the compact support of MLSEB to be
1%, 3%, 5% and 10% of the image size, which are shown in Figure 5.11.

The angle and color thresholds we used in this comparison are \( L = 90 \) and \( C = 100, 110, 120, 130, 140, 150, 160, 170, 180, \) and 190, respectively. As mentioned before, to
show a succinct result, we compute the average sum of all the color and angle threshold
Figure 5.9: FDEB drawings of the U.S. airlines using different compatibility threshold values: (a) 0.8; (b) 0.7; (c) 0.6; and (d) 0.5.

combinations of the corresponding coverage and uncertainty respectively. Figure 5.12 shows a scatter plot of coverage versus uncertainty of different parameters of the three methods. We first analyze the impact of compatibility threshold to the FDEB drawing. The label $c = 0.6$ means the compatibility is set to be 0.6 for FDEB, etc. The compatibility threshold determines the clustering result. As the compatibility threshold decreases, the similarity estimation is more loose, which results in more edges are bundled together. We can observe this trend in Figure 5.9. In Figure 5.12, the coverage and conditional entropy increase monotonically with the increase of the compatibility value, which matches our expectation that if more edges are bundled, it becomes less uncertain in perception but hard to reconstruct the original graph structure through perception. This phenomenon also happens to MLSEB in this dataset. As the compact support (i.e., the window size) increases, edges are merged into larger bundles. Meanwhile, the coverage and conditional entropy
also decrease accordingly. We also look at the kernel size of FFTEB. As the kernel size increases, the edges are bundled tighter, and more edges gather into larger bundles. Greater kernel size results in less coverage and conditional entropy. However, we are surprised to see that, when $k = 3\%$ and $k = 5\%$, the conditional entropy of the two results are almost the same, but $k = 3\%$ shows a significantly greater coverage. By comparing Figure 5.10b,c, we find that the visual result of $k = 3\%$ is very similar to the result of $k = 5\%$. As mentioned in Section 5.3.3, we should achieve a high coverage and low conditional entropy in an edge bundling drawing. Hence, Figure 5.10b is better than Figure 5.10c in terms of the proposed coverage and conditional entropy.

Analyzing the coverage and conditional entropy can help users choose optimal parameters for the edge bundling applications. We can set a 2D interval to select the optimal parameter for our desire edge bundling methods. Since we intend to achieve high coverage
and low conditional entropy in an edge bundling drawing, we can define a 2D interval by setting a lower bound for coverage and an upper bound for conditional entropy respectively. Take Figure 5.12 as an example; we can set the lower bounds of coverage to be 0.85 and the upper bound of conditional entropy to be 0.8, which means we want the method can achieve at least 85% coverage and at most 80% conditional entropy. The 2D interval gives three instances that are a FDEB with a 0.6 compatibility threshold, a FFTEB with a 3% kernel size, and an MLSEB with a 5% compact support. In this analysis, users can determine a customized 2D interval to select the appropriate parameters for their desire edge bundling applications. This analysis can also be easily extended to cases with more complex parameter combinations.

Figure 5.11: MLSEB drawings of the U.S. airlines using different compact support values: (a) 1% compact support; (b) 3% compact support; (c) 5% compact support; and (d) 10% compact support.
5.4.4 Comparison II

The purpose of this section is not to merely compare the quality of the three methods using different datasets. We have introduced a bivariate analysis to analyze and compare the three methods with different parameters for one specific application. Users and readers should follow the analysis method in Section 5.4.3 to test different parameter combinations for different datasets. In this section, we discuss the three edge bundling methods in depth. The main focus of this section is on analyzing their similarity and difference based on the visual result, coverage, and conditional entropy across different datasets. We use three datasets: (1) U.S. airlines (2100 edges and 235 nodes); (2) U.S. migrations (9780 edges and 1700 nodes); and (3) large U.S. migrations (545,881 edges and 3075 nodes). We use segment-based node–link diagram, FDEB, FFTEB and MLSEB to visualize the three
datasets, and compute the corresponding coverage and conditional entropy values. Due to the limitation of the implementation of FDEB, we cannot construct an edge bundling drawing for large U.S. migration with FDEB. Remember that we used angle threshold $L = 90$ and color threshold $C = 100, 110, 120, 130, 140, 150, 160, 170, 180,$ and $190$ in our path-finding algorithm. Moreover, we compute the average sum of all angle and color thresholds for coverage and conditional entropy, respectively. The additional images we used in this section are shown in Figure 5.8. We plot the corresponding scatter plot of $H(W)$ and $\lambda$ in Figure 5.13.

First, we analyze the difference between the node–link diagram drawing and the three edge bundling methods. The node–link diagrams in Figure 5.8a,e have relatively more visual clutter than other edge bundling methods because of edge crossings and total area used. The visual clutter can be quantitatively interpreted by our proposed conditional entropy $H(W)$. The coverage and conditional entropy of the node–link diagram drawings fully reflect this phenomenon. According to the result of Figure 5.13, the node–link diagram has overall the highest coverage; however, its uncertainty $H(W)$ is significantly larger than the other three edge bundling methods, which matches our expectation. The coverage of the node–link diagram is saturated but the visual result is of high uncertain, which can be interpreted by $H(W)$. Hence, we have an impression that, although node–link diagram gives the highest coverage rate, it is also the most uncertain visualization compared to the other three edge bundling methods.

Second, we observe that some results of Figures 5.10 and 5.11 are visually similar. However, these results are very different from FDEB in Figure 5.9. As we revisited the FDEB in Section 5.4.1, the final result of FDEB is determined by the edge clustering result. Since only compatible edges will be bundled together, the clustering result determines the potential position where the edges are in the final drawing. The clustering result is based on the edge level. In FFTEB, the sample points of edges are moved to the local density
maxima according to the gradients of the local density estimation. It uses a kernel size to determine the effect of the local density estimation. In MLSEB, bundled curves are locally approximated by the local compact support in a regression manner. To construct such a bundled curve, it also needs to use a compact support (window size) to estimate the weights and positions of the local neighboring sample points. Indeed, both FFTEB and MLSEB use a window size (i.e., kernel and compact support, respectively) in their bundling processes. The estimation process of both methods is essentially density estimation. Hence, the clustering or merging process of FFTEB and MLSEB is based on the sample point level. An interesting observation is that, in U.S. airlines and U.S. migrations datasets, FDEB has the highest coverage and conditional entropy value, and the coverage and conditional entropy are significantly greater than the other two edge bundling methods. Another interesting observation is that, when the compatibility threshold is high ($c = 0.8$), the result in Figure 5.9a resembles Figure 5.8a. Hence, it yields great values of coverage and conditional entropy. Based on the above observation, we can conclude that edge clustering methods prone to generate a more visual uncertain edge bundling result. Although it reduces small-angle edge crossings, it cannot reduce the edge crossings among edge clusters. This is why FDEB results visually have more edge crossings than the other two methods, which can be reflected by the condition entropy $H(W)$ in Figure 5.13.
As the density estimations of FFTEB and MLSEB are similar, some results of FFTEB and MLSEB are similar as well (e.g., Figure 5.10a versus Figure 5.11a, and Figure 5.10b versus Figure 5.11b). FFTEB and MLSEB both produce web-like bundle effect that is completely different from FDEB. What differentiates them is the advection methods of the sample points. FFTEB uses the gradients of the density estimation as the advection vector while MLSEB uses regression method to project the sample points to new positions. When the kernel size and compact support are both below 5%, their results are almost identical. However, when the kernel size and compact support reach 5% or greater, their results become different. Figure 5.8c,d,g–j illustrates this phenomenon. When we use a very large dataset, the large U.S. migrations (545,881 edges and 3075 nodes), their results become completely different, as shown in Figure 5.8i,j. The resulting scatter plot in Figure 5.13 shows that FFTEB and MLSEB have their own advantages and disadvantages for U.S. airlines and U.S. migrations. However, MLSEB outperforms FFTEB in the very large dataset with respect to
both coverage and $H(W)$.

5.5 Conclusion

I have presented a new information-theoretic metric for evaluating the overall uncertainty of edge bundling visualizations. We used the theory of the mutual information to assess the final results of edge bundling methods. Based on the information-theoretic pipeline of Chen et al. [15, 14], we have discussed the benefits and drawbacks of edge bundling visualizations. We defined and quantified the uncertainty $H(X|Y)$ of the visual description of the relation between two vertices. We attributed $H(X|Y)$ to the actual number of paths between the two vertices in the corresponding drawing. $H(X|Y)$ basically measures the length of the bits necessary to describe the relation given the visual description. We then argued that the average of total sum of $H(X|Y)$ of all pairs of vertices can be used to evaluate the uncertainty of edge bundling visualizations, i.e., $H(W)$. The key idea of our evaluation is that the lower the value of $H(W)$ produced by an edge bundling method, the better the method is. We show an example for users to choose optimal parameters for their desired edge bundling applications. We used this metric to compare three different edge bundling methods, the force-directed edge bundling (FDEB), the fast fourier transform edge bundling (FFTEB), and the moving least squares edge bundling (MLSEB) in depth, which has not been fully discussed in the existing literature. We found that $H(W)$ can correctly reflect the degree of ambiguity of edge bundling algorithms. Note that our framework only focuses on the vis-encoder process of Chen et al.’s pipeline. We admit that some factors in the vis-channel and vis-decoder process, such as display resolution, view angle, interaction, and cognition would affect the final observer’s comprehension. We focus on the vis-encoder process because it is easier to explain the concept of the uncertainty caused by edge bundling visualizations. Although we did not take these factors into account, our methodology is easy
to extend in the vis-channel and vis-decoder process. In addition to the path-finding method, we will try to use other methods to approximate the value of coverage and conditional entropy of edge bundling methods. We also would like to include user studies in our future work. A user study can testify and give more accurate values of coverage and conditional entropy from a user perception perspective. In addition, general graph visualization can use our information-theoretic metric to quantify the quality of their visualization results. Our proposed metric can also be used to generate novel graph visualizations and be applied to real-world applications.
Chapter 6

A Deep Learning Framework for Edge Bundling

6.1 Introduction

The family of edge bundling techniques have provided rich graph layouts to effectively describe relational structures from a graph. The different layouts are generated by different statistical or computational methods (e.g., spring-force model, Kernel Density Estimation (KDE) model and Moving Least Squares (MLS)). Each of these methods yields different results or styles of graph layouts. However, it lacks a unified framework that can generate the layout (style) of edge bundling methods. Developing a unified layout (style) generator is a challenging task. On one side, few unified frameworks of graph visualization have been presented in the literature. On the other side, developing a unified framework that can mimic the computational processes of different methods is a formidable task. In this chapter, I present a deep learning framework for generating results of the image-based edge bundling methods.

Deep learning techniques are promising to discover intriguing patterns from data in
artificial intelligence applications. Deep learning techniques have been introduced and achieved tremendous successes in many scientific and applied fields, such as image processing, pattern recognition, computer vision, volumetric rendering, and natural language processing. Deep learning has been used as a unified framework in image segmentation [66], multi-task learning [17], multi-label classification [116], and biometric verification [104]. Deep learning techniques have attracted the visualization community in recent years. Since deep learning techniques often include complex multi-layer architectures and a considerable number of parameters, visualizations can help developers and researchers inspect and understand the multi-layered structure and backpropagation process of deep learning networks. Much visualization work focused on visualizing models, parameters, and processes in deep learning networks. Yet, little attention has been put into using deep learning techniques to generate style-wise graph visualizations.

In this chapter, I present a deep learning framework for generating results of the image-based edge bundling methods. The proposed framework relies on the Convolutional Neural Network (CNN). I name the framework as CNN-bundling. My framework provides a general deep model to generate similar results from two different classes of edge bundling visualizations. First, I revisit three different edge bundling methods and their techniques from Section 5.4.1. Second, according to the generality of the edge bundling methods, I conclude that grouping similar edges is technically advecting the sample points of the edges locally. Hence, the objective of our learning model is to infer the sample points such that the locations of inferred points will be as close to the sample points after advection as possible. Third, I use synthetic graphs, and generate the corresponding edge bundling results of a specific edge bundling technique. The generated results are used as training datasets. Through training ten-thousands of results of a specific edge bundling visualization, our deep framework can learn the style of the edge bundling technique, and generate similar results compared to the target edge bundling technique. I demonstrate the effectiveness of
our framework using comparisons between the results generated from our deep learning framework and the results generated from three different edge bundling methods. The edge bundling techniques I intend to train are KDE-based edge bundling and MLS-based edge bundling.

### 6.2 Related Work

Since edge bundling methods have been reviewed in previous chapters, I refer readers to Section 2.2 for revisiting the edge bundling methods. In this work, my method tends to address the generation of different styles of two different types of edge bundling methods using a deep learning technique. In this section, I will first revisit the image-based edge bundling methods, which include KDE-based and MLS-based edge bundling methods. I will also discuss some representative work in general graph uncluttering methods and deep learning fields. For a more diverse graph visualization review, I refer readers to the following work. Comprehensive surveys have been conducted by Herman et al. [43] and Vehlow et al. [112] for graph visualization. Large graph visualizations and dynamic graphs have been reviewed by Landesberger et al. [114] and Beck et al. [8] respectively.

Image-based techniques used a density assessment to guide bundling processes [11, 28, 47, 63, 84, 110]. A skeleton-based edge bundling is given in SBEB [28]. They combine a distance field and an edge clustering method to create a 2D skeleton that is suitable for converging the edges. The following methods are generally based on *Kernel Density Estimation*. Kernel density estimation edge bundling (KDEEB, Figure 2.2.(f)) [47] first introduced this idea. The basic pipeline is to first transform an input graph into a density map using kernel density estimation, and then move the sample points of edges towards the local density maxima to form bundles. Peysakhovich et al. [84] extended KDEEB using edge attributes to distinguish bundles. CUDA Universal Bundling (CUBu) [110] used GPU
acceleration to enable interactively bundling a graph with a million edges. Fast Fourier Transform Edge Bundling (FFTEB) [63] improved the scalability of density estimation by transforming the density space to the frequency space. The MLS-based method in Chapter 4 is also an image-based technique since it uses the same uniform sampling method to sample the edges of a graph, and it also requires a local density assessment to regressively fit local curves.

Edge bundling is one of the graph uncluttering methods. Therefore, our method is highly related to graph uncluttering algorithms that include node overlap removal and edge uncluttering. The graph uncluttering problem has been addressed by a few researchers. Chuang et al. [68] used a potential field to remove node overlaps. Lyons et al. [71] proposed a method to maintain the mental map of the layout by distributing the nodes of a graph evenly. Force and spring models are often used in graph uncluttering and repositioning. The initial work is done by Fruchterman et al. [34]. Li et al. [65] used a spring algorithm to remove node overlaps. Huang et al. [46] used a force model to separate nodes which are close. Gansner et al. [37] used a combined force and Voronoi diagram to generate an overlap-free layout. Marriott et al. [74] and Dwyer et al. [26] leveraged constraint optimization in overlap removal.

Deep learning techniques have been introduced to various scientific and applied fields. Deep learning techniques have been used as a unified framework in image segmentation [66], multi-task learning [17], multi-label classification [116], and biometric verification [104]. My work mainly uses CNN architecture to extract local features of the distribution of sample points of a graph. Krizhevsky et al. [55] have used CNN technique in image classification and recognition, and achieved tremendous success. The following studies [16, 25, 38, 103] have shown that CNN of deep learning can be adaptable to specific tasks in pattern recognition, object detection, and scene recognition. Researchers have made some efforts in addressing the estimation of density estimation using neural network [3, 67].
6.3 Problem Formalization

Essentially, edge bundling techniques advect the sample points of edges of a graph to cluster edges in order to reduce edge clutter in terms of visual results. Similar to the previous chapter, I refer to sample points of edges as sites. The operations of edge bundling techniques are based on advecting each site to a new position where the new position is decided according to the corresponding underlying algorithm. As mentioned in the previous chapters, Mean Shift is used in KDE-based edge bundling methods, Moving Least Squares Projection is used in MLSEB, and Spring Force is used in FDEB. Mean Shift advects sites to the local density maxima, and Moving Least Squares Projection first fits a regression curve locally and then projects the sites to the regression curve. The Spring Force model of FDEB moves sites with combined forces. The combined forces applied on each site are generated by a force-interactive model that depends on the distance from the site to the nearby compatible sites. Note that in FDEB, not all the nearby sites should be considered to be interactive with the site. Only the nearby sites that are compatible to this site are taken into account, which is illustrated in Section 5.4.1. Hence, before advecting each site, a local information of neighbour sites must be collected for each site. The Spring Force model, Mean Shift and Moving Least Squares Projection algorithms were utilized to accomplish the position assignment of a point cloud based on the information from the local neighbour sites of each site. According to Equation 2.1, I formalize the operation of edge bundling algorithms as follows: given a point cloud $s = \{(x_1, y_1), (x_2, y_2), (x_3, y_3), \ldots, (x_n, y_n)\}$ sampled from the edges of a graph, edge bundling is to find a position assignment of the sites, in which each site $(x_i, y_i)$ in the assignment is moved to a new position, such that connecting the corresponding sites with new positions of each edge reduces the edge clutter for the graph visualization. Different algorithms impose different bias towards the final positions for the point cloud. Hence, different edge bundling techniques generate different layout styles for a
Figure 6.1: The pipeline of our CNN framework for edge bundling.

6.3.1 Model Description

I introduce a deep CNN framework to edge bundling methods. The overview of our CNN framework is presented in Figure 6.1. Since this framework uses CNN architecture, I name this framework CNN-bundling. The reason why I choose a deep CNN framework over other deep learning framework is that the implementation of KDE-based and MLS edge bundling is essentially an image-based method. It requires each site to calculate the local distribution of neighbour sites (as discussed in Section 5.4.1), which can be implemented using convolutional methods (e.g., CUBu [110]). In deep learning applications, CNN framework is often used to extract the local important feature and information for forward propagation in image classification. Hence, CNN framework is adequate for training the two edge bundling methods. The input of our model is the movement between the current and next stage of each site, i.e., the initial position of each site and the new position of each site after advection. The output I want to extract from this model is the feature vectors that can infer the position of the current stage of each site to a predicted position \( \mathcal{P}(x_i, y_i) \), which is often named inference, and \( P \) is the function of the forward propagation of the deep
training model. Suppose I have the current position \((x_i, y_i)\) of a site and its corresponding new position \((\bar{x}_i, \bar{y}_i)\), which is generated by the edge bundling algorithms (e.g., Force model, Mean Shift, or Moving Least Squares). I refer \((\bar{x}_i, \bar{y}_i)\) to be the groundtruth position in the following content. My model consists of one input layer, four convolutional layers, four fully connected layers, and one loss function. The overview of our architecture is shown in Figure 6.1. The input layer is essentially the vector representation of the current position \((x_i, y_i)\). The hidden layers contain two components. The first one is multi-convolutional layers, the other is a fully connected neural network. The multilayer of CNN is to find the local feature for each site, which is essentially the density estimation of the sites in the local area. The fully connected layers use the local feature generated from CNN to predict the coordinates of the sites in the next stage. Each hidden layer between the input and output layers is a feature vector. The values of feature vectors are initialized to random values. The output layer, which is the last layer of the fully connected layers, outputs a vector that is the predicted position through forward propagation. The groundtruth \((\bar{x}_i, \bar{y}_i)\) is used as a supervising correction, such that the gradient can be computed via a loss function, and it is fed backward through the network to update the feature vectors. The feature vector is obtained after a number of iterations, which is usually referred to as epochs in Artificial Intelligence applications. Finally, I extract the feature vectors for each site and apply the trained feature vectors to predict the new position of a site. Using \(\mathcal{P}(x_i, y_i)\) in the visual result, I can have the inferred edge bundling result of a specific edge bundling method. In the rest of this section, I will discuss the specific implementation of the input layer, the hidden layer, and the output layer.

First, similar to Section 3.4.2, I accumulate sites on an image buffer. The width \(w\) and the height \(h\) of the buffer are set to be the resolution of the display. The accumulation can be done by using direct rendering. After rendering, the buffer gives a \(w \times h\) histogram of 3-tuples that contain the \(x_i\) and \(y_i\) coordinates of the site and the number of sites falling
into that pixel. The 3-tuple histogram is then a \( w \times h \times 3 \) matrix. This is the data in the input layer. Second, I use a convolutional neural network combined with a fully connected neural network as the hidden layers in our deep learning architecture. My architecture of hidden layers contains four convolutional layers followed by four fully connected layers. Each layer (including both convolutional and fully connected layers) has a ReLU activation function. The kernel size of each convolutional layer is \( w_n \times h_n \). The exacted features in each layer are set to be \( k_0, k_1, k_2, \) and \( k_3 \). The convolutional layers output a \( w_n \times h_n \times k_3 \) matrix. As this matrix is the output of the four convolutional layers, the \( k_3 \) features are extracted, which should contain a good representation of the local distribution of sites based on \( x_i, y_i \) and the accumulated number of sites. By combining \( x_i \) and \( y_i \) of the site with the \( k_3 \) features falling into the \((x_i, y_i)\) pixel, I can get a \( k_3 + 2 \) features from the site. I then feed the \( k_3 + 2 \) features into a fully connected neural network and the output layer generates a vector \( \text{OUT}_i = \mathcal{P}(x_i, y_i) \), which is the predicted coordinate of the original site. Suppose I have the output \((\overline{x}_i, \overline{y}_i)\) from an existing algorithm as supervising groundtruth, the loss function can be thus formalized as:

\[
\mathcal{L} = \| \mathcal{P}(x_i, y_i) - (\overline{x}_i, \overline{y}_i) \|
\]  

(6.1)

I use stochastic gradient descent (SGD) to minimize the loss function through multiple epochs. The feature vectors are updated through multiple epochs based on the gradient with respect to the loss function \( \mathcal{L} \). That is, I want to reduce the discrepancy between the predicted positions of sites and the ground truth as much as possible. If the discrepancy can be reduced small enough, I consider my model can predict the position of sites given a certain edge bundling algorithm. I use multiple synthetic graphs to train the feature vectors and use data from the real world data to infer the edge bundling results. Given \( m \) synthetic graphs \( \{G_1, G_2, ..., G_i, ..., G_m\} \), I consider the edges of each graph can be sampled into
\[ s_i = \{ (x^i_1, y^i_1), (x^i_2, y^i_2), (x^i_j, y^i_j), \ldots, (x^i_n, y^i_n) \} \]. I consider the sites in a graph as a batch. My model only trained one batch at a time. In each epoch, I iterate through all batches to update the feature vectors. I set the kernel size to be 6% of the resolution size. The feature size \( k_0, k_1, k_2, \) and \( k_3 \) in the convolutional layer are set to be 20, 30, 40, and 50, respectively. The sizes of the fully connected layers are set to be 100, 60, 20, and 2, respectively. Figure 6.1 illustrates these settings. In this chapter, I use a heuristic study to decide these parameters, and use Tensorflow [1] as our deep learning programming library.

### 6.4 Synthetic Dataset

In this work, I use a synthetic dataset as the training dataset, and use a real-world dataset as the test dataset to generate the inferred result. The reason why I use a synthetic dataset in the training process is that using real-world datasets as training data is not practical. This is because, in a regular training process, it may require a considerable number of data. Typically, Krizhevsky [55] used the CNN architecture that requires at least one million images as train datasets. On the other hand, I want the inferred results to cover as many conditions as possible. These conditions may contain different edge placements in the original node-link diagram. In order to avoid overfitting, our training dataset should contain as many different edge placements as possible. In real-world applications, it is very hard to collect a sufficiently large amount of graph data that cover all possible conditions. Hence, I decided to use synthetic graphs as training datasets. The synthetic dataset I use in this chapter contains graphs in which the vertices of edges are randomly generated. I think when the number of edges of a graph is large enough, the graph is reasonable to use as a training dataset. The inference will be compared to the results generated from the existing state-of-the-art edge bundling methods in order to demonstrate our work in Section 6.6. Our synthetic dataset contains ten thousand graphs. The two vertices of an edge are randomly
placed in a [20, 80] space. To achieve a general model and avoid overfitting, I use graphs with different edges. The graphs with 1000 to 4000 edges and the graphs with 4000 to 8000 edges accounted for 40% of the total graphs each. The remaining 20% are the graphs that have 8000 to 16000 edges.

6.5 Training and Testing

In one training epoch, one batch contains a number of sites. I use the sites in a graph as a batch. One training epoch just uses the current position in the input layer and the groundtruth position as correction in the output layer. The output layer is updated through forward propagation and the feature vectors are updated via backpropagation. The feature vectors are the output that will be used in the test phase. Each training epoch is to learn the movement of one iteration of a specific edge bundling process. In the test phase, I use the same number of inference steps as the iteration number of the corresponding edge bundling methods to infer the predicted positions because I want to generate the similar edge bundling result. That is, if an edge bundling process iterates ten times to move the site of a graph for the final result, I will conduct the same number of times of forward propagation to infer the predicted positions in the reference phase. The number of iterations used in MLSEB is 10 while in CUBu, we use 20 iterations to get the bundling result from the synthetic graphs. To generate a perfect model is difficult and may be very time-consuming. Hence, I set a threshold of mean squared error generated from the loss function. When the mean squared error is below the threshold, CNN-bundling stops training. The threshold is set to be 0.06, and the drawings of the graphs are all normalized to a 0 to 100 space.
Figure 6.2: The visualization results of (a) US airlines using CNN-bundling, (b) US airlines using KDE-based edge bundling, (c) US migrations using CNN-bundling, and (d) US migrations using KDE-based edge bundling.

Figure 6.3: The visualization results of the large US migrations using (a) CNN-bundling and (b) CUBu.
6.6 Results

In this section, I compare the inferred results of CNN-bundling and the results generated by the edge bundling methods. Specifically, the edge bundling methods compared in this section are the results generated from the original CUBu and MLSEB. First, I compare the visual results. Figure 6.2 and Figure 6.3 show the comparisons of the results generated from our CNN framework and CUBu using the graphs of U.S. airlines, U.S. migrations, and France airlines, respectively. The feature vectors in this comparison are trained from ten thousand synthetic graphs using the edge bundling transition of CUBu. The kernel of our CNN framework in the feature vectors is 6% of the resolution size. My framework generates a very similar visualization compared to the result of the original CUBu. I also use the MLS-based method to train the feature vectors with the same CNN framework and the synthetic graphs. Figure 6.2 and Figure 6.3 show the comparisons of the results generated from CNN-bundling and MLSEB using U.S. airlines, U.S. migrations, and France airlines, respectively. These figures demonstrate that CNN-bundling can generate similar results of different edge bundling methods.

Next, I compare the inferred results and the groundtruth results statistically. I first compare how close the inferred points are to the sample points after iterative advections for each edge bundling technique. A straightforward way is to compare the Euclidean distance between an inferred point and the corresponding point in groundtruth, i.e., error. I use the mean squared error to measure the difference between the inferred edge bundling result and the groundtruth result of a graph. I calculate the mean squared error of all sites in a Euclidean space that is normalized to 0 to 100. The mean squared error is presented in a percentage manner. Moreover, I use the information-theoretic metric presented in Chapter 5 to measure and compare the results. Table 6.1 and Table 6.2 show that the statical information of the inferred edge bundling results and the groundtruth of the corresponding
(a) US airlines using CNN-bundling  
(b) US airlines using MLS-based edge bundling.  
(c) US migrations using CNN-bundling  
(d) US migrations using MLS-based edge bundling

Figure 6.4: The visualization results of (a) US airlines using CNN-bundling, (b) US airlines using CUBu, (c) US migrations using CNN-bundling, and (d) US migrations using CUBu.

As Table 6.1 and Table 6.2 show, the results generated from CNN-bundling have larger coverage and $H(W)$ values in most of the cases, since Figure 6.2, 6.3, 6.4, and 6.5 show that the groundtruth edge bundling results have a tighter bundling effect. My framework is a generic edge bundling generator that can generate different styles of CUBu and MLSEB using the same configuration in the code.

Table 6.1: Comparison between CUBu and CNN-bundling

| Graph    | U.S. airlines | | U.S. migrations | | France airlines | |
|---------|---------------|---|-----------------|---|-----------------|
|         | Mean error    | Coverage | $H(W)$ | Mean error    | Coverage | $H(W)$ | Mean error    | Coverage | $H(W)$ |
| CUBu    | N/A           | 73.9%     | 0.663  | N/A           | 59.2%     | 0.532  | N/A           | 58.1%     | 0.645  |
| CNN-bundling | 1.08%    | 71.0%     | 0.701  | 0.890%        | 60.9%     | 0.500  | 1.79%        | 63.3%     | 0.712  |
Figure 6.5: The visualization results of France airlines using (a) CNN-bundling and (b) MLSEB.

Table 6.2: Comparison between MLSEB and CNN-bundling

<table>
<thead>
<tr>
<th>Graph</th>
<th>U.S. airlines</th>
<th>U.S. migrations</th>
<th>France airlines</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean error</td>
<td>Coverage</td>
<td>H(W)</td>
</tr>
<tr>
<td>MLSEB</td>
<td>N/A</td>
<td>80.1%</td>
<td>0.846</td>
</tr>
<tr>
<td>CNN-bundling</td>
<td>1.806%</td>
<td>83.6%</td>
<td>0.855</td>
</tr>
</tbody>
</table>

6.7 Conclusion

In this chapter, I leverage a CNN framework to learn the edge bundling results from a set of synthetic graphs. I formalize the problem of our learning model for edge bundling to be a point movement problem where the inferred points should be as close to the advected points of the real edge bundling techniques in every iteration as possible. I design a set of CNN layers and a loss function to learn the advection of points of a corresponding edge
bundling technique. My results show that the CNN framework is able to generate similar results compared to the real edge bundling methods. My approach showed that I provide a good initial work to investigate the deep learning model for graph visualization. Though the running time of our deep learning framework is not superior to the original GPU-based implementation of the compared edge bundling methods, I believe that our promising result will inspire researchers to investigate deep learning approaches for graph visualization, and possibly improve the performance and effectiveness in the future. I also believe that our theoretical model and experimental results can contribute to a comprehensive picture of the applications of machine learning for graph visualization. In the future, I would like to improve the performance and accuracy of our deep learning framework. I believe that generating a good edge bundle result with our deep learning framework is not confined only to CNN. Other models would be useful, and more edge bundling methods can be unified in the future. Last but not least, our deep learning framework can be very easily extended to a broader range of graph visualization (for example, node uncluttering algorithms) in the near future.
Chapter 7

Conclusion and Future Work

In this dissertation, I have holistically addressed the key challenges in edge bundling methods, and presented several novel algorithms for edge bundling communities. My contribution in this field is fourfold:

First, I tackle edge bundling from a system perspective. I exploit standard graphics hardware functions and libraries, and present a novel framework that enables edge bundling on web-based platforms in Chapter 3. I adopt a force-directed edge bundling method and an image-based edge bundling method in my framework, combining a parallel texture-based processing scheme to accelerate the bundling process. My framework enables interactivity on web-based platforms. In this framework, I take advantage of the texture-based parallel processing of WebGL, which is supported in most of the ubiquitous smart devices. My work optimizes an end-to-end (from bundling to rendering) process that enables an effective and interactive large graph visualization on web-based platforms. My web-based implementation achieves a great performance improvement compared to the fastest CUDA-based edge bundling implementation on a large graph with half of a million edges. I also demonstrate the performance of my framework by comparing the existing web-based and CUDA-based edge-bundling methods using different real-world graphs and standard graphics cards on
different devices.

Second, I further improve edge bundling by considering the aspects of algorithm complexity and visualization quality. I introduce a new edge bundling method that leverages a Moving Least Squares (MLS) approximation. My edge bundling method is named Moving Least Squares Edge Bundling (MLSEB). In Chapter 4, I illustrate that MLSEB improves graph layout quality based on a preliminary quantitative metric. MLSEB also simplifies the computational steps of edge bundling methods. My approach generates bundle effects by iteratively projecting each sample point of an edge to its local regression curve based on its neighborhood’s density. Such a local regression method can produce a better convergence result using fewer iterations. Complicated processing steps, such as resampling and smoothing, are thus avoided. Compared to other edge bundling methods, my method provides a better quality based on a preliminary quality metric. MLSEB achieves the same quantitative quality value with less computational steps compared to the state-of-the-art edge bundling algorithm. MLSEB is implemented used GPUs. The performance result also shows that my CUDA-based implementation is at the same order of magnitude of the fastest edge bundling method in terms of computational speed.

Third, I propose an information-theoretic framework to evaluate the visual results of edge bundling techniques in Chapter 5. I first illustrate the advantage of edge bundling visualizations for large graphs, and pinpoint the ambiguity resulting from drawing results. Second, we define and quantify the amount of information delivered by edge bundling visualization from the underlying network using information theory. I present a new algorithm to evaluate the resulting layouts of edge bundling using the amount of the mutual information between a raw network dataset and its edge bundling visualization. The information-theoretic metric relies on the number of visual paths between two nodes in a graph. Third, I use a computational method to approximate the number of paths between two nodes in a graph drawing image. Comparing different parameter combinations of a specific method using the proposed metric
helps find out the optimal parameter combination of a specific edge bundling method for an edge bundling application. Comparison examples based on the proposed framework between different edge bundling techniques are also presented. Domain experts can use our metric to find out the best edge bundling method for their applications.

Fourth, I present a generic deep learning framework to generate different styles of edge bundles using a deep learning technique in Chapter 6. Our deep learning technique can mimic the styles and layouts of a specific edge bundling methods. My deep learning framework mainly uses a Convolutional Neural Network (CNN) architecture to learn and mimic the local information estimation of edge bundling methods. Using the feature vectors trained from the results of a specific edge bundling method, we can infer the final layout result of the corresponding edge bundling method. Our framework is a unified edge bundling layout generator that can adapt the image-based edge bundling methods. Our framework can also inspire researchers to investigate how to use deep learning techniques to generate effective graph visualization results, such that the mature edge bundling field can be pushed forward.

In the future, I will explore deep learning methods to infer the results of graph visualizations for general graph layouts. Apart from bundling methods, this research can be also applicable and extendable for addressing node uncluttering in graph visualization. I will also extend my research to community detection for extremely large graphs. It is promising to leverage the results of community detection to create novel effective visualization for extremely large graphs.
Bibliography


