Low-Stretch Trees for Network Visualization

by

Rebecca Linda McKnight

B.Sc., The University of Victoria, 2012

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL
STUDIES
(Computer Science)

The University of British Columbia
(Vancouver)

August 2015

Abstract

Low-stretch trees are spanning trees which provide approximate distance preservation for edges in the original graph by minimizing stretch. We explore the application of these trees to network visualization. In particular, we present a novel edge bundling technique, LSTB, that computes edge bundles explicitly and efficiently and does not rely on fixed vertex positions. This approach is in contrast to previous methods, which require the user to provide a layout of the input graph. We introduce an abstract framework for edge bundling methods, which provides a unifying formalization of bundling terminology and techniques, as well as a classification of such methods. Based on this framework, LSTB provides algorithmic support for sophisticated visual encodings, including dynamic layout adjustment and interactive bundle querying.

In addition, we explore the use of the multiplicative weights update method to compute a distribution over low-stretch trees in order to achieve low stretch for all edges in expectation, rather than on average. We present the results of using this distribution in place of a single low-stretch tree as a routing graph for LSTB. While the distribution provides better stretch guarantees, we find that from a visual perspective a single low-stretch tree provides a better routing graph for the LSTB edge bundling application.

Preface

This thesis is the expanded version of currently unpublished work done with Nicholas J. A. Harvey and Tamara Munzner. The ideas and algorithms presented here were developed jointly by us. I performed all implementation and results gathering presented in Chapter 6. Chapter 5 and Chapter 7 were written solely by me.

Figures 1.2, 6.1a, 6.1c, 6.1e, and 6.1g used with permission.

Table of Contents

List of Tables

List of Figures

Acknowledgments

I would like to thank my research supervisor, Nick Harvey, for his help over the past two years. His teaching, both in and out of the classroom, has been invaluable to me. I am so grateful for the time and effort he put into guiding me through this process.

I would also like to thank Tamara Munzner for her input and insight into the visual aspects of this work. Her keen eye for beauty and design was an invaluable resource and essential to the success of this thesis.

I thank Alex Telea for providing data for comparison and Quirijn Bouts for providing Figure 6.1e.

I am also deeply grateful to my husband, Christopher, and to my family for their love and support throughout my degree. Their faith and encouragement give me motivation to succeed.

Chapter 1

Introduction

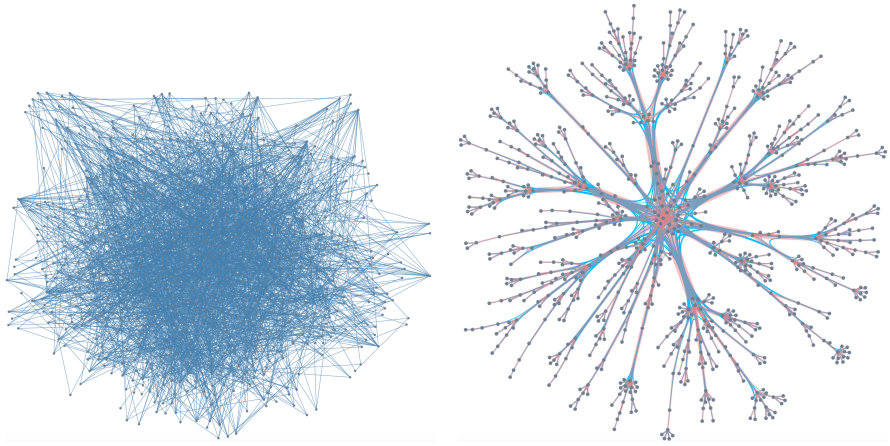
Graphs are common in many applications, such as: computational biology, computer networking, and natural language processing. Two key areas that study graphs and their representations are graph theory and information visualization. **Graph theory** studies properties and applications of graphs. Common problems in graph theory include constructing graphs with particular properties, computing subgraphs with a particular structure, and finding routes through a graph. **Information visualization** ("vis") creates "visual representations of datasets designed to help people carry out tasks more effectively" [23]. In particular, **network visualization** focuses on visualizing graph data.

Finding a visually effective embedding of a given graph is often difficult, however, unless the data set is very small or the graph admits a nice property, such as planarity. In order to improve the visual representation of a graph by reducing visual clutter, two approaches might be considered: adjusting vertex positions and adjusting edge positions. The latter is known as **edge bundling**, and has been studied extensively in the information visualization literature. Edge bundling methods visually group edges into bundles in order to minimize edge clutter. We introduce a new classification of such methods as either **layout-based** or **layout-free**. Layout-based methods require the input graph to have a pre-computed layout: that is, geometric positions for all vertices. These methods use information from the layout, such as edge proximity, to perform bundling. For example, geo-spatial data includes a specific embedding which can be used to inform bundling. However,

a particular layout may not match a user's intent or needs, or they may not know how to choose an appropriate layout. In addition, pre-computing a layout may be expensive. In such situations, layout-based edge bundling will likely not be useful.

In contrast, layout-free edge bundling methods do not use any information regarding vertex positions in order to perform bundling. That is, the bundling computed by such methods is independent of layout. Therefore, any graph layout can be used. An example of this is shown in Figure 1.2, where different tree layouts are used to draw the same software package interaction data. However, the bundling in both cases is the same: the same edges are bundled together regardless of adjusted vertex positions. This example is taken from Hierarchical Edge Bundles (HEB) [17], which is the only pre-existing layout-free edge bundling method we are aware of. HEB takes a compound graph as input: that is, a graph together with a specific hierarchical relationship defined by a rooted spanning tree on the vertices. The hierarchy is used as a routing graph, where graph edges are routed through the hierarchy in order to form bundles.

This method of using a routing graph to perform bundling is also common among layout-based methods [10, 11, 21, 26]. However, some of the routing graphs used are dense, which leads to inefficiencies when finding routes. From this per-



(a) Input graph. **(b)** LSTB bundled graph.

Figure 1.1: Comparison between the input graph and the output of LSTB on the Poker data set.

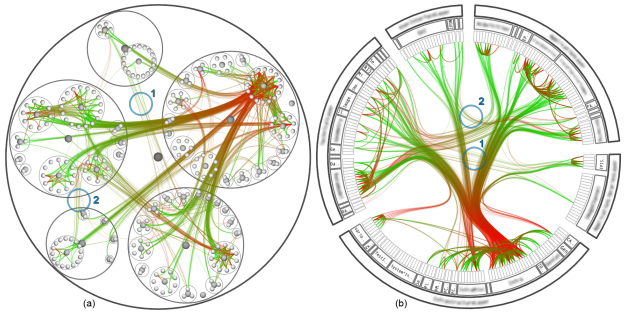


Figure 1.2: © 2006 IEEE Comparison between two bundled layouts of software package interaction data from Hierarchical Edge Bundles [17]. In each case, although the vertex positions differ, the bundles are the same.

spective, HEB is ideal, since it uses a spanning tree to route edges. HEB requires this hierarchy as input, though, which renders the method unusable in cases where no hierarchy is known. For example, a protein interaction graph has no inherent layout and no underlying tree structure; therefore, neither layout-based methods nor HEB are sufficient.

To address these problems, we introduce a new edge bundling system: LSTB. Figure 1.1 shows an example of the input and output of our method. Our system is comprised of two steps: the edge bundling algorithm and the visual encoding step. Figure 1.3 shows the system workflow.

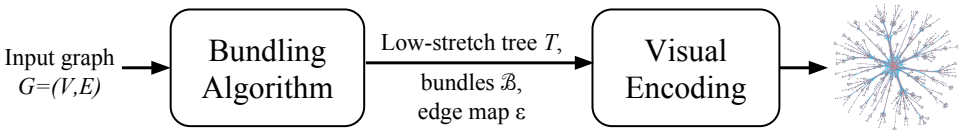


Figure 1.3: LSTB system workflow.

In the first step, our layout-free bundling algorithm computes a particular type of spanning tree, a *low-stretch* tree [3], to use for edge routing. Given a graph $G = (V, E)$ and a spanning tree $T = (V, E_T)$, the **stretch** [3] is the ratio between path length in the tree and path length in the original graph. A **low-stretch tree** is a spanning tree that approximately minimizes the stretch of edges on average. Such

a tree provides the distance preservation that we desire in a good routing graph.

For graph G, the stretch of an edge $e = (u, v) \in E$ in tree T is defined as:

$$s_T(e) = d_T(u, v)$$

where $d_T(u, v)$ is the path length from u to v in T. The overall stretch of G is defined as:

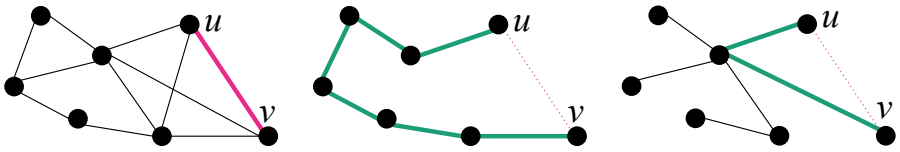
$$s_T(G) = \frac{1}{|E|} \sum_{e \in E} s_T(e)$$

Figure 1.4 gives an example of two spanning trees: one with poor (large) stretch, and one with good (small) stretch.

One of our main contributions is to introduce the use of low-stretch trees to the problem of edge bundling. Low-stretch trees became objects of interest in the graph embeddings literature several decades ago, but they are also the key objects underlying several recent breakthroughs in spectral graph algorithms, such as maximum flow in near-linear time [28].

The power of low-stretch trees can be seen through a simple example. Consider an n-by-n grid graph, or mesh. Arbitrary spanning trees generally do a poor job of encapsulating the structure of this graph, as shown in Figure 1.5a. However, Alon et al. [3] show that low-stretch trees succeed at capturing this structure. This is illustrated in Figure 1.5b.

Most low-stretch tree construction algorithms, however, only guarantee the preservation of distances on average. In contrast, we explore the use of a distribution over low-stretch trees to achieve low stretch for edges in expectation. Such

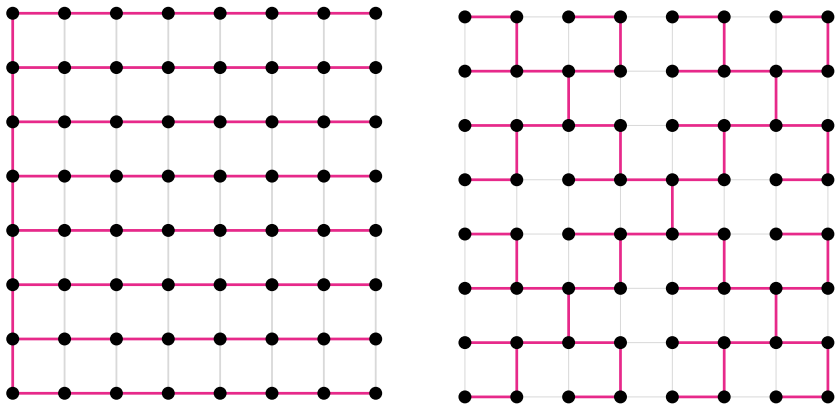


(a) Original graph G, with comparison edge $e = (u, v)$ highlighted

(b) Tree T with $s_T(e) = 6$, $s_T(G) = 28/11$

(c) Tree T with $s_T(e) = 2$, $s_T(G) = 17/11$

Figure 1.4: Comparison between trees with poor (high) and good (low) stretch, respectively.



(a) Minimum spanning tree.

(b) Low-stretch spanning tree, as shown in Alon et al. [3].

Figure 1.5: Comparison between a minimum spanning tree and a low-stretch spanning tree for an 8-by-8 grid graph.

a distribution can be obtained via a reduction to a zero-sum game, where the payoff values are stretches of edges from the input graph in spanning trees. One player wishes to find an edge with maximum stretch, whereas the other wishes to find a tree which minimizes the stretch of that edge. By Von Neumann's Minimax Theorem, there exists a distribution which gives each player the best possible payoff given their opponent's strategy. We use the multiplicative weights update method [6] to approximate such a distribution. In contrast to using a single tree, we also explore using this distribution over low-stretch trees for our edge bundling algorithm.

Many existing edge bundling methods perform bundling visually without real consideration of the bundles themselves: the bundles are either not explicitly computed, or not explicitly shown to the user. In contrast, our bundling algorithm outputs the bundles directly. In order to explain this concept clearly, we introduce an abstract framework for edge bundling which includes a formal definition of a bundle. This definition provides the data abstraction for our visualization design, according to the nested model [23].

The visual encoding step of our system is responsible for drawing the bundled graph. This step corresponds to the visual encoding and algorithm levels of the

nested model for visualization design [23]. Since our bundling algorithm is layout-free, we can use any existing vertex layout method. We also explore several ways of visually encoding bundles; the techniques we use are pre-existing, but their application to edge bundles is new. In addition, we incorporate interactive features such as dynamic layout adjustment and bundle query on hover.

Depending on the task, one bundling algorithm may be more suitable than another. However, the data in question poses restrictions on which methods are even feasible. Table 1.1 explains these restrictions in conjunction with our layout-based and layout-free classifications by comparing method usage depending on whether the user has a hierarchy or layout for the input graph. As seen in this table, LSTB can be used in each scenario, and clearly fills a gap in the current edge bundling literature.

In summary, we report on the following contributions: LSTB, a new layout-free edge bundling technique that computes bundles explicitly; an abstract framework for edge bundling that clearly defines terminology and techniques used in such methods; the utilization of the multiplicative weights update method to compute a distribution over low-stretch trees; and the introduction of low-stretch trees to the edge bundling literature.

	Has Hierarchy	*No Hierarchy*
Has Layout	$H_{\checkmark}L_{\checkmark}$: Any method	$H_{\times}L_{\checkmark}$: Any layout-based method, or LSTB
No Layout	$H_{\checkmark}L_{\times}$: HEB [17], or LSTB	$H_{\times}L_{\times}$: **LSTB**

Table 1.1: A comparison of edge bundling method usage, broken down based on whether the graph has a layout or hierarchy. Our method, LSTB, is applicable in all four cases.

Chapter 2

Literature Review

We discuss prior work related to low-stretch trees, edge bundling, and other visualization techniques. Low-stretch tree computation methods will be reviewed as well as motivations and popular applications of these objects. We also give an overview of important work on edge bundling, particularly the methods that inspired LSTB. In addition, we discuss previous work on several visualization techniques that we use to display the results of our bundling algorithm.

2.1 Low-Stretch Trees

Low-stretch trees were first introduced by Alon, Karp, Peleg and West in 1995 [3] for use in the k-server problem, a key problem in online algorithms. They formulate a zero-sum game for the problem, where the *tree player* picks a spanning tree of the graph and the *edge player* picks an edge in the graph; the payoff is then the stretch of the chosen edge in the chosen tree. This framework is described in detail in Section A.1. By von Neumann's Minimax Theorem [29], for any fixed strategy x by the edge player there exists a strategy y for the tree player consisting of a single tree which achieves $\min_y x^\intercal M y$, where M is the payoff matrix of the game. Alon et al. assume a uniform strategy for the edge player, and provide an algorithm that constructs a single tree as a fixed strategy for the tree player which provides stretch $\exp(O(\sqrt{\log n \log\log n})) = O(n^{0.01})$ for edges on average.

In 2001, Boman and Hendrickson [8] discovered low-stretch trees could be

used as preconditioners for solving linear systems where the matrix is the Laplacian of a graph. The work of Spielman and Teng, and others [28] dramatically extended the work of Boman and Hendrickson by giving provably near-linear time algorithms for solving Laplacian linear systems. These fast Laplacian solvers are used in problems such as max flow and bipartite matching [28]. This breakthrough prompted a renewed interest in developing algorithms for computing low-stretch trees.

Elkin, Emek, Spielman and Teng [12] introduced a star-decomposition technique which their algorithm uses to compute a spanning tree with average stretch $O(\log^2 n \log\log n)$. Abraham, Bartal and Neiman [2] extend this technique to compute a distribution $\mathcal{T}$ over spanning trees in order to obtain good expected stretch: $\mathbb{E}_{T\in\mathcal{T}}[s_T(e)] = \tilde{O}(\log n)$. This distribution consists of all spanning trees induced by their hierarchical star partition algorithm with non-zero probability. Based on this result, they obtain an average stretch guarantee of $O(\log n \log\log n(\log\log\log n)^3)$. A more recent algorithm for computing low-stretch trees from Abraham and Neiman uses similar ideas to develop a petal-decomposition technique which improves average stretch to $O(\log n \log\log n)$ [1]. The goals of these algorithms, however, are to improve stretch guarantees. Therefore, most of these algorithms are very theoretical and are not practical to implement.

2.2 Edge Bundling

One of the first papers on edge bundling was Hierarchical Edge Bundling (HEB) [17]. As mentioned previously, this algorithm takes as input a compound graph and uses the hierarchy to route the additional edges. Existing tree drawing methods are used to lay out the hierarchy, and splines are used to draw the routed edges through the tree. While this method does not explicitly compute bundles before laying out the tree, the bundles are layout-independent and therefore this method is layout-free. Interestingly, all subsequent work has been exclusively layout-based, to the best of our knowledge.

Some early work on edge bundling focused on routing-based techniques, where bundles are formed by edges being routed along a mesh or grid. Geometry-Based Edge Clustering [11] generates a control mesh and computes control points on the

mesh which edges are routed through. Winding Roads [21] computes a hybrid quad-tree/Voronoi diagram according to vertex positions of the input graph, and then routes the edges along this grid. Force-Directed Edge Bundling (FDEB) [18], another early method, takes a different approach. This algorithm subdivides edges and exerts spring forces on the subdivision points in order to attract edges towards one another. Unfortunately, this iterative procedure tends to be quite slow, and involves many parameters.

More recent methods employ unique and varied techniques. Skeleton-Based Edge Bundling (SBEB) [13] clusters similar edges together and generates a skeleton from their medial axes, then attracts edges to the skeleton to produce bundles. Kernel Density Estimation Edge Bundling (KDEEB) [19] computes a density map of the input graph's layout using kernel density estimation, and then applies image sharpening to merge local high points in order to form bundles. Multilevel Agglomerative Edge Bundling (MINGLE) [14] constructs a proximity graph for edges, then bundles edges according to this graph with the objective of minimizing ink used. This method does identify bundles explicitly, but does not exploit them when drawing the resulting bundled graph. However, this method is the fastest known at this time.

Several recent methods have employed spanners as routing graphs [10, 25, 26]. A **spanner** is a subgraph that approximately preserves edge distances. These methods compute a spanner of the visibility graph and use it to route edges. Unfortunately, spanners have known poor guarantees in terms of sparsity: to preserve distance with factor $O(t)$, $\Omega(n^{1+1/t})$ edges are needed [4]. Furthermore, these methods are computationally expensive: the most recent method requires $O(n^2 \log^2 n)$ time to compute a spanner [10]. We will use a routing graph which can be quickly computed, and provides both sparsity and approximate distance preservation on average.

2.3 Network Visualization

The computation of the routing graph in our bundling algorithm relies on an iterative coarsening process that is similar in spirit to the coarsening steps used in many multi-level graph layout systems such as FM^3 [15]. However, low-stretch trees

come with provable bounds on quality, which make them ideal routing graphs (see Section 4.1.1 for more details). In our visual encoding step, we employ existing graph visualization techniques. We use a tree as the backbone for graph layout; this approach is used by many previous systems, including SPF [5]. We create perceptual layers using colour and opacity both statically and dynamically, with support for interactive highlighting of bundles or edges on hover, as in previous systems such as Constellation [24].

Chapter 3

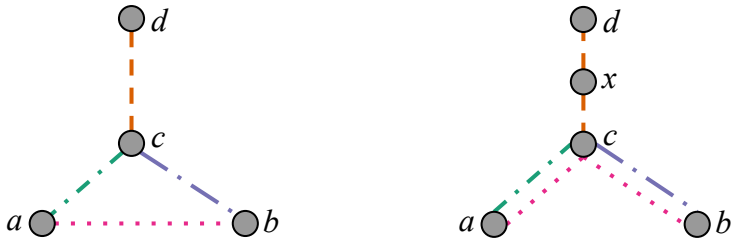
Abstract Framework for Edge Bundling

One contribution of this paper is to formalize an abstract framework for edge bundling that answers the question: what is a bundle? We wish to solidify the intuitive notion of a grouping of edges in order to explicitly compute and visually represent edge bundles. Our framework will use this definition to describe the requisites of a bundling algorithm.

3.1 Framework Details

We define the process of **edge segmentation** as a key component of all bundling methods. Let $G = (V, E)$ be a graph with $n = |V|$ vertices and $m = |E|$ edges. An example graph is shown in Figure 3.1a. We wish to subdivide each edge $e = (u, v) \in E$ into segments, such that each edge can be represented by an ordered list of segments. A **segment** $s = (x, y)$ is an edge which cannot be subdivided further. A segment's endpoints may be: an endpoint of the original edge e, another vertex from V, or a *dummy* vertex. Dummy vertices may be introduced during segmentation to subdivide an edge. This case is shown for edge (c, d) in Figure 3.1, where x is a dummy vertex. Alternatively, edges may be routed through existing vertices in the graph. This case is shown for edge (a, b) in Figure 3.1, where the segmentation routes through c. Also, an edge may consist only of a single segment.

These cases are shown for edges (a,c) and (b,c) in Figure 3.1. The decision of how to segment edges depends on the bundling algorithm. Once all edges have been segmented, we have a mapping $\mathcal{E}$ from each edge $e \in E$ to its corresponding ordered list of segments.



(a) Input: original graph $G = (V,E)$, where edges are differentiated by colour and dash style.

(b) Output: segmented graph $M = (V \cup \{x\}, S)$, with corresponding colour and dash styles.

Figure 3.1: Illustration of the input and output of edge segmentation.

Let S be the multiset of all segments, and U be the set of all dummy vertices. The result of the segmentation process is a multigraph $M = (V \cup U, S)$, as shown in Figure 3.1b. A **bundle** is then a set of two or more segments, $B \subset S$, $|B| \geq 2$. Let $\mathcal{B}$ denote the set of all bundles. Segments in the same bundle are referred to as **bundle neighbours**, although they need not share any endpoints from V in order to be bundled together. An edge $e \in E$ *belongs* to any bundle that contains one of its segments. A segment may be contained in at most one bundle. A particular bundling algorithm must decide how to assign segments to bundles.

We define an **edge bundling algorithm** as an algorithm which takes as input a graph $G = (V,E)$ and outputs a set of bundles $\mathcal{B}$, as well as a mapping $\mathcal{E}$ between edges $e \in E$ and ordered lists of segments. Based on this definition, an algorithm must determine how to perform edge segmentation, as well as how to assign segments to bundles. While most existing edge bundling methods implement a system where the output is a graph drawing, we believe the concepts from this framework apply nonetheless. This framework provides a context from which all bundling methods can be discussed.

3.2 Application to Existing Methods

One of our goals for this framework is to provide a formalization of bundling techniques such that these methods can be compared in a uniform way. We will exemplify this by applying the framework to several of the existing methods introduced in Chapter 2. For each method, we will discuss the edge segmentation process and how bundles are formed.

3.2.1 Force-Directed Edge Bundling

FDEB [18] models edges as springs which attract one another in order to form bundles. The algorithm proceeds in cycles, where each cycle consists of a number of iterations. Edge segmentation is performed using dummy vertices. Edges are subdivided uniformly by dummy vertices, where the number of such vertices introduced starts at one and doubles each cycle. Consecutive vertices along the same edge are attracted to one another via a spring force. Edge compatibility measures are used to determine whether two edges should interact, where pairs of edges with compatibility above a certain threshold are considered to be interacting edges. For such pairs of edges, an electrostatic force is used to attract corresponding pairs of dummy vertices, which is how edges are bundled together.

This method implements a system which outputs the graph drawing directly. Although edges are drawn together in bundles, the bundles are not explicitly computed, so it is not known whether two edges are bundled together or not. While the edge compatibility measure indicates whether two edges exert force on each other, they may not appear visually in the same bundle.

3.2.2 Multilevel Agglomerative Edge Bundling

MINGLE [14] takes as input a set of edges with fixed endpoint positions. Each edge is considered to be a four-dimensional vector based on the location of its endpoints. An edge proximity graph is constructed by finding the k closest neighbours of an edge according to Euclidean distance in the 4D space; the vertices of this graph are edges of the original graph. An iterative coarsening process is then used to form bundles, where vertices in the proximity graph are bundled together and coalesced if their bundling minimizes ink used to draw the edges.

In this case, edges are bundled without segmentation. Bundles are computed explicitly, since edges are directly assigned to bundles based on their ink-saving compatibility. However, these bundles are only used to guide edge drawing, rather than for additional visual feedback or interactivity. When drawing bundled edges, edges are segmented using two dummy vertices as meeting points. For a set of bundled edges, the corresponding dummy vertices of each edge will be drawn in the same location, so the edges share a common segment. The edges then fan out from these meeting points to their respective endpoints.

3.2.3 Clustered Edge Routing

Clustered Edge Routing [10] performs edge bundling using spanner-based routing. First, links are clustered using a well-separated pair decomposition which is consistent with the compatibility measures of Holten and van Wijk [18]. This gives an explicit bundling, since each well-separated pair of point sets (A, B) induces a cluster containing edges which have one endpoint in A and the other in B. Next, a visibility graph is computed such that vertices are connected by an edge if they are visible to each other. Dummy vertices are also added to route around any obstacles, which include vertices of the original graph. A spanner of this graph is computed in order to improve sparsity; this technique is discussed further in Section 2.2. This spanner is then used to route edges, where edges which are bundle neighbours must share a common sub-path in the graph. This bundles the edges visually.

As in the previous case, edges are bundled without segmentation and the bundles are only used for drawing. When drawing, each edge is segmented by the dummy vertices on its route through the routing graph. Several additional techniques are used to reduce visual clutter further, including merging obstacles, edge ordering and crossing minimization.

Chapter 4

LSTB

Our layout-free bundling method takes as input a connected graph $G = (V, E)$. The first step of the system is the bundling algorithm, which computes the routing tree $T = (V, E_T)$ and the set of bundles $\mathcal{B}$. The visual encoding step then draws the bundled graph, which includes interactive elements for data exploration.

4.1 Bundling Algorithm

Our novel edge bundling algorithm computes bundles directly without relying on the geometry of vertices or edges. This layout-free approach routes edges through a tree to determine both edge segmentation and bundle membership. The algorithm consists of two steps: routing graph generation, where the spanning tree is computed; and edge routing, where edges are segmented and bundles are formed.

4.1.1 Routing Graph Generation

Bouts and Speckmann [10] quantify several desirable properties of a routing graph, including two that are layout-independent: sparsity, and that the shortest path between two vertices in the routing graph should not be much longer than their direct connection in the graph. Since we are computing a spanning tree, the first property is achieved by definition. Therefore, we focus on the second property, which amounts to a guarantee that the spanning tree preserves edge distances from the original graph. This is equivalent to a low-stretch guarantee, as introduced in

Chapter 1.

We will use a low-stretch tree as our routing graph for bundling. In order to compute such a tree, we use the algorithm from Alon et al. [3]. This algorithm performs an iterative coarsening process in order to compute the low-stretch tree (LST). This process is explained in detail in Section 5.1.

This algorithm runs in time $O(m \log n)$, and the tree is guaranteed to have low average stretch: $\exp(O(\sqrt{\log n \log \log n})) = O(n^{0.01})$. As discussed in Section 2.1, other methods with better theoretical guarantees are known [2, 12], but their algorithms are not practical to implement. Regardless, any method can be substituted to compute the LST in this step, provided there are guarantees on the stretch of the resulting tree.

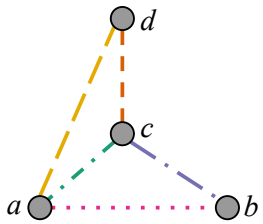
4.1.2 Edge Routing and Bundling

Our routing graph is a low-stretch tree $T = (V, E_T)$. Let the **remainder** edges be those not in the spanning tree: $E_R = E \setminus E_T$. We will segment the remainder edges by routing them through T. Tree edges E_T will be represented by single segments, as (a, c) and (b, c) are in Figure 3.1b.

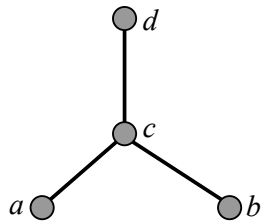
For any two vertices in a connected graph, there is a unique path between them in any spanning tree of the graph. For any edge $e = (u, v) \in E_R$, the unique u-v path through T gives its segmentation. That is, for every edge (x, y) on the u-v path in T, a new segment will be added and the ordered list of these segments will represent e. In terms of our framework, we are segmenting through existing vertices in V, rather than adding dummy vertices. This case is shown for the (a, b) edge in Figure 3.1.

As discussed in Chapter 3, the output of this segmentation process is a multigraph that is supported on the same set of edges as T. We define our bundles as sets of segments that have the same endpoints. Therefore we will have at most $n - 1$ bundles, since $|E_T| = n - 1$, and for any segment (x, y) it must also be true that $(x, y) \in E_T$. Figure 4.1 illustrates this process.

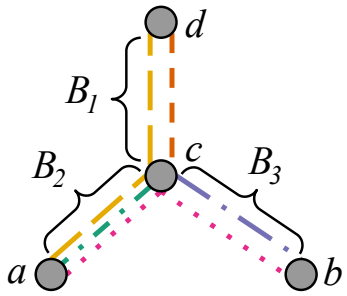
To perform segmentation and bundling efficiently, we need a fast subroutine to handle queries for u-v paths in our spanning tree T. A naïve approach is to simply perform breadth-first search, but this requires $\Theta(|V|)$ time per query. Since



(a) Original graph G, where colour and dash style are used to differentiate edges.



(b) Routing tree T.



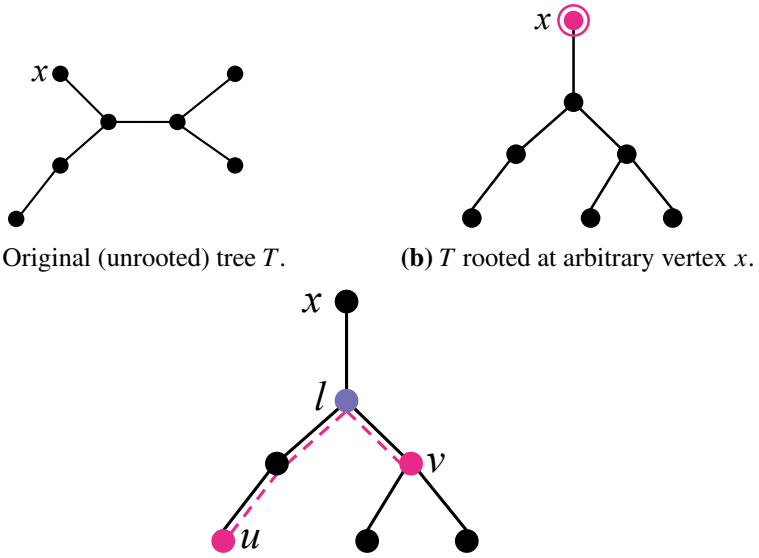
(c) Result of the bundling algorithm of LSTB. Segments have the same colour and dash style as their corresponding edge in G, and bundles B_1, B_2, B_3 are indicated.

Figure 4.1: Illustration of the bundling algorithm of LSTB. A low-stretch tree $T = (V, E_T)$ is computed for use as a routing graph. Segmentation is then performed by routing edges through this tree, using the routing algorithm illustrated in Figure 4.2. Bundles are then formed from groups of segments sharing the same endpoints as edges in E_T.

$\Omega(|E_R|)$ queries are needed, this approach requires $\Omega(|V| \cdot |E_R|)$ time. Instead, we will use a two-phase approach that pre-processes the graph in linear time, but can then perform queries in time proportional to the length of the returned path (which is optimal, as the path itself is returned). Performing such a query for every edge in E_R requires time $O(s_T(G) \cdot |E_R|)$, which is small since it is a low-stretch tree.

This subroutine can be easily implemented as follows. The pre-processing step simply roots the tree at an arbitrary vertex and directs all edges towards the root. Then, to find a u-v path in T, step *in parallel* from each of u and v towards the root, marking the nodes along the way. The first marked vertex encountered on either

path is the lowest common ancestor ℓ, and the time to identify it is proportional to the length of the u-v path. The path through the tree is then the concatenation of paths u-ℓ and ℓ-v, and the segmentation of (u, v) is this path. This process is illustrated in Figure 4.2.



(a) Original (unrooted) tree T.

(b) T rooted at arbitrary vertex x.

(c) To find the path from u to v in T, we find their lowest common ancestor, ℓ, by climbing up the tree. The path from u to v is then the concatenation of paths u-ℓ and ℓ-v.

Figure 4.2: Illustration of the routing algorithm of LSTB from Section 4.1.2. The algorithm roots tree $T = (V, E_T)$ at an arbitrary vertex in order to speed up queries, which then only need to find the lowest common ancestor between two vertices.

Once the u-v paths have been computed for each $(u, v) \in E_R$, bundles are formed such that all segments that have the same endpoints are bundle neighbours. The output of our bundling algorithm is then: T, our low-stretch tree; $\mathcal{B}$, our set of bundles; and $\mathcal{E}$, our map from edges in E to their segmentation.

4.2 Visual Encoding

We wish to take advantage of the key assets of our bundling algorithm when designing the visual encodings for LSTB: namely, that it is layout-free and that it

computes bundles directly. We will do so by allowing arbitrary layout adjustment, as well as adding interactive bundle queries. The figures in this section show the Poker graph (see Table 6.1 for more details).

4.2.1 Vertex Layout

Since the bundles we compute are independent of layout, we can position the vertices in any manner we choose. We lay out the routing graph $T = (V, E_T)$ using the standard force-directed graph layout built into D3 [9]. Since our routing graph is a tree, we could also take advantage of tree layouts, such as the Reingold-Tilford algorithm [27]. Figure 4.3 shows a comparison of these layouts.

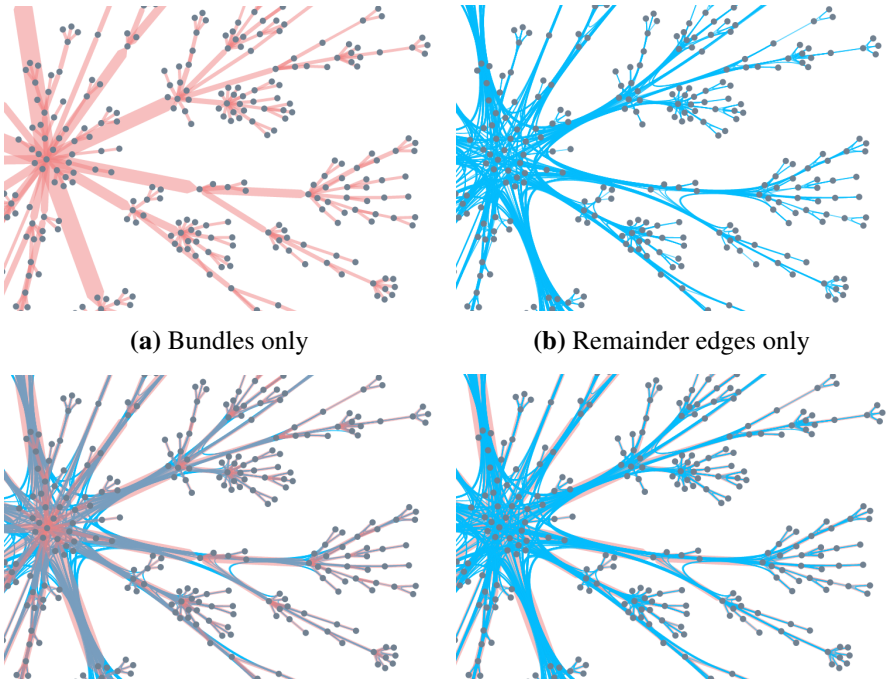
(a) Force-directed layout **(b)** Radial Reingold-Tilford layout

Figure 4.3: Comparison between vertex layout methods for the routing graph T.

Additionally, vertices may be arranged according to a user-defined layout, but imposing this layout on our tree backbone may not produce an effective visualization. Figure 6.1h shows such an example for comparison purposes. We also allow dynamic layout adjustment: vertices can be dragged in order to adjust their position, as discussed further in Section 4.2.3.

4.2.2 Edge Drawing

Previous edge bundling methods draw edges individually, so bundles are shown as closely located or overlapping edges. However, we wish to also take advantage of



(a) Bundles only

(b) Remainder edges only

(c) Bundles foreground

(d) Remainder edges foreground

Figure 4.4: Examples of visual encoding options for LSTB.

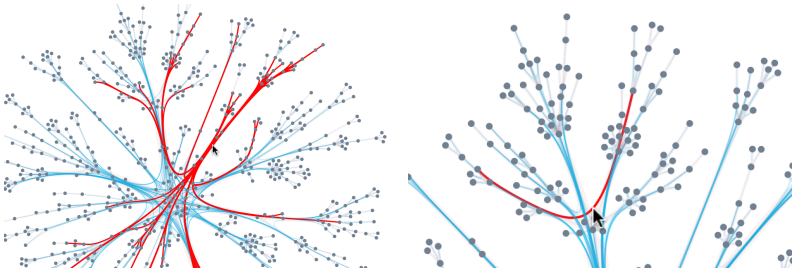
the fact that our algorithm outputs bundles, and that all segments in a bundle share the same endpoints. We will do so by drawing bundles explicitly, both independently from and in combination with individual drawing of remainder edges.

Figure 4.4 shows the different visual encoding options for bundles and edges. Bundles are depicted by straight lines with tapered endpoints and varying thickness, as shown in Figure 4.4a. The thickness of bundles varies in proportion with bundle size: that is, the number of segments in a bundle. Individual edges are drawn as splines, as shown in Figure 4.4b. The control points for the spline of edge (u, v) are the endpoints of its segments, which correspond to the vertices in the u-v path in T. This approach is similar to the method of HEB [17].

Bundles and edges can also be drawn simultaneously, where distinct perceptual layers are created by adjusting opacity. Figures 4.4c and 4.4d show the difference between having bundles and remainder edges as the foreground layer.

4.2.3 Interactivity

Another contribution of our bundling method is increased support for user interaction. Vertex re-positioning by dragging is possible without re-running the edge bundling algorithm because the algorithm is layout-free and bundles are computed independently of layout, in contrast to previous methods. Different hovering techniques are also possible, since bundles are computed explicitly and edge membership in bundles is known. Figure 4.5a shows how when a bundle is hovered over, all edges belonging to that bundle are highlighted. Figure 4.5b shows how single edges are individually highlighted on hover.



(a) Hovering over a bundle highlights edges in that bundle

(b) Hovering over an individual edge highlights that edge

Figure 4.5: Comparison between bundle querying and edge querying on hover.

Chapter 5

Low-Stretch Trees

LSTB uses low-stretch trees as routing graphs for edge bundling. The history behind these trees is discussed in Section 2.1. This chapter will go into more detail on the construction algorithm of Alon et al. in addition to exploring the computation of distributions over low-stretch trees in order to achieve low stretch for edges in expectation.

In this chapter, we will consider weighted graphs $G = (V, E, w)$ where $w : E \to \mathbb{R}_{\geq 0}$. We will update our previous definition of stretch as follows, where $T = (V, E_T, z)$ is a spanning tree of G with weights $z : E_T \to \mathbb{R}_{\geq 0}$:

$$s_T(e) = \frac{d_T(u, v)}{w(e)} \quad \text{for } e = (u, v)$$

Here, $d_T(u, v)$ is the z-weighted distance from u to v along the unique path between them in T.

5.1 Low-Stretch Tree Construction

LSTB computes low-stretch trees using the method of Alon, Karp, Peleg and West [3]. Their algorithm, given in Algorithm 1, takes as input a weighted n-vertex multigraph G. The algorithm starts by breaking the edges into weight classes, and then proceeds in rounds. At each step of the algorithm, the vertices of the graph are partitioned into clusters such that each cluster has low topological diameter. A shortest-paths spanning tree is then computed for each cluster, and the edges

from these trees are added to the (initially empty) low-stretch tree. Each cluster is then contracted into a meta-vertex, and edges are added to represent connections between vertices in different clusters. The algorithm then iterates on this new multigraph.

The key properties of the clusters are that each cluster has a spanning tree of radius $\leq y^{j+1}$ at round j, and in every non-empty edge class E_i for $1 \leq i \leq j$, the fraction of inter-cluster edges is at most $1/x$, where the optimal value for x is shown to be $\exp(\sqrt{\log n \log\log n})$. This algorithm is guaranteed to find a low-stretch tree with stretch $\exp(O(\sqrt{\log n \log\log n})) = O(n^{0.01})$ [3].

In their paper, Alon et al. do not give runtime guarantees for their algorithm. However, an informal analysis shows that the runtime is $O(m \log n)$ by looking at the number of iterations performed. At each step, the number of edges shrinks by a factor of x, where $x \approx \log n$. Therefore the number of iterations is $\log_x m \leq \log_{\log n} n^2 = O(\log n)$. At each round we do $O(m)$ work, so the total runtime is indeed $O(m \log n)$.

5.1.1 Parameters

The low-stretch tree algorithm of Alon et al. depends on two key parameters, x and y. These parameters are set at the beginning of the algorithm, where $x = \exp(\sqrt{\log n \log\log n})$ and $y = 27x \ln n \cdot \left\lceil \frac{\ln n}{\ln x} \right\rceil$. Alon et al. set the values for these parameters according to their theoretical guarantees for the stretch of the resulting tree. The x parameter is responsible for controlling the expansion of each cluster. The cluster starts at a root node, then expands outward to its neighbours, working level-by-level until the next level contains fewer edges than an x-fraction of the existing edges in the cluster. The y parameter is used for weighted graphs to segregate edges into weight classes E_i. This way, the algorithm considers lighter edges first in order to attain the radius bound given earlier.

Algorithm 1 AKPW algorithm to compute low-stretch tree T in multigraph G [3]

function LOWSTRETCHTREE(G)

input: Graph $G = (V, E, w)$

output: Low-stretch tree $T = (V, E_T, w)$

// Set parameters

$x = \exp(\sqrt{\log n \log\log n})$, $\rho = \left\lceil \frac{3 \ln n}{\ln x} \right\rceil$, $\mu = 9\rho \ln n$, $y = x\mu$

$i_{\max} = \lfloor \log_y \max_{(u,v)\in E} w(u,v) \rfloor + 1$

$E_i = \{(u,v) \in E : w(u,v) \in [y^{i-1}, y^i)$ for $i = 1, \ldots, i_{\max}$

// Perform iterative rounds

$j = 1$

$T = (V, E_T = \emptyset)$

while $\bigcup_i E_i \neq \emptyset$ **do**

$C =$ CLUSTER(G_j, j, E_i's$, x, y$)

for all $c \in C$ **do**

$G_c =$ INDUCEDSUBGRAPH(G_j, c)

$T_c =$ SHORTESTPATHSPANNINGTREE(G_c) *// Uses Dijkstra's alg*

for all $(u,v) \in E_{T_c}$ **do**

$E_T = E_T \cup \{(u,v)\}$

$V_{G_{j+1}} = \{v_c : c \in C\}$ *// Contract each cluster into a single vertex*

$E_{G_{j+1}} = \emptyset$

for all $(u,v) \in E_{G_j}$ **do**

if $u \in c_i$ **and** $v \in c_j$ **and** $i \neq j$ **then**

$E_{G_{j+1}} = E_{G_{j+1}} \cup \{(v_{c_i}, v_{c_j})\}$ *// Add inter-cluster edges*

else

$i = \lfloor \log_y w(u,v) \rfloor + 1$

$E_i = E_i \setminus (u,v)$ *// Remove intra-cluster edges*

$G_{j+1} = (V_{G_{j+1}}, E_{G_{j+1}})$

$j = j + 1$

return $T = (V, E_T, w)$

Algorithm 1 AKPW algorithm to compute low-stretch tree T in multigraph G [3]

function CLUSTER(G, E_i's, j, x, y)

input: Graph $G = (V, E, w)$, edge buckets E_i, and parameters j, x, y

output: Clustering $C = \{c_1, \ldots, c_K\}$ where $\bigcup_{k \in \{1,\ldots,K\}} c_k = V$

$C = \emptyset$

$V_{\text{rem}} = V$

while $V_{\text{rem}} \neq \emptyset$ **do**

Choose $u \in V_{\text{rem}}$ arbitrarily

$V(0) = \{u\}$

$E_i(0) = \{\}$

$\ell = 0$

repeat

$\ell = \ell + 1$

$V(\ell) = \{v \in V_{\text{rem}} : |\text{SHORTESTUNWEIGHTEDPATH}(G, u, v)| = \ell\}$

$E_i(\ell) = \{(v_1, v_2) \in E_i : v_1 \in V(\ell), v_2 \in V(\ell) \cup V(\ell - 1)\}$

until $\forall 1 \leq i \leq j,\ |E_i(\ell)| \leq \frac{1}{x}|E_i(1) \cup E_i(2) \cup \ldots \cup E_i(\ell - 1)|$

$c = V(1) \cup V(2) \cup \ldots \cup V(\ell - 1)$

$C = C \cup \{c\}$

$V_{\text{rem}} = V_{\text{rem}} \setminus c$

return C

5.2 Distribution over Low-Stretch Trees

Section 2.1 discusses several methods for computing low-stretch trees. In general, these methods compute a single tree and provide low stretch on average over all edges in the original graph. The method of Abraham et al., however, gives theoretical results based on finding a distribution over low-stretch trees [2]. In doing so, they provide low stretch for each edge in expectation. Unfortunately, their algorithm is theoretical in nature and not practical to implement. We will present a practical solution for finding a distribution over low-stretch trees.

We will use a reduction to a zero-sum game, given in Section A.1. We wish to find a distribution D over trees T such that $\mathbb{E}_{T \sim D}[s_T(e)]$ is small for all $e \in E$. In

the zero-sum game framework, for some strategy (i.e. probability distribution) over trees y, we have $\mathbb{E}_{T\sim y}[s_T(e_i)] = \sum_{T\in y} \Pr[T] \cdot s_T(e_i) = (My)_i$ for any $e_i \in E$. Therefore, our goal is to find $\min_y \max_x x^\intercal My$, which is exactly the goal of the tree player in the game. Our distribution over trees is thus given by y^*, the optimal distribution from von Neumann's Minimax Theorem [29]. The multiplicative weights update method [6] is an algorithm for approximating such distributions.

The Multiplicative Weights Update Method (MWUM) is an algorithmic framework which optimizes decision-making by adjusting weights on decisions. Each decision has some associated cost, which is revealed only after the decision has been made. The algorithm proceeds in rounds, updating the weights depending on the consequences of the decision made in that round. As the algorithm progresses, the weights are distributed to reflect the knowledge of which decisions are better or worse in terms of cost. The expected cost of the algorithm can be shown to be not much worse than making a fixed decision throughout; this holds true for the best overall decision as well [6].

Arora et al. [6] provide a tailored version of MWUM for solving zero-sum games approximately. In this case, the algorithm's rounds correspond to simulated rounds of the game and weight updates correspond to changes in a player's strategy. This method can be used to find strategies $\hat{x}$ and $\hat{y}$ such that, for error δ,

$$\min_y \hat{x}^\intercal My \geq \lambda^* - \delta \quad \text{and} \quad \max_x x^\intercal M\hat{y} \leq \lambda^* + \delta \tag{5.1}$$

Therefore, regardless of their opponent's strategy, a player following such a strategy is guaranteed payoff at most δ from the optimal. In this case, we wish to find $\hat{y}$, since this will be our distribution over trees. In order to compute the trees in this distribution, we will use the method of Alon et al. [3].

5.2.1 Multiplicative Weights Update Method

Algorithm 2 adapts the method from Arora et al. to solve our zero-sum game reduction, with the goal of finding a distribution over trees. If Alice is our edge player and Bob is our tree player, we want Bob's distribution over trees to achieve low stretch for any distribution over edges that Alice may choose. At each round, the algorithm adjusts Alice's weights to focus on edges which had high stretch in the

previous round, so that the resulting distribution over trees will be robust.

The algorithm performs ρ weight update rounds, where the weights γ are over the edges $(e_1, \ldots, e_m)$. Note that the graph may have edge weights w as well, but these should not be confused with the multiplicative weights γ of the algorithm. At the start of round t, the weights γ are normalized to form the distribution $x^{(t)}$, which is Alice's current strategy. Then, the TREEORACLE method returns a low-stretch tree with respect to the distribution $x^{(t)}$ (this will be discussed further in Section 5.2.2). In doing so, Bob is choosing the best pure strategy given Alice's distribution. The edge weights are then updated, based on whether an edge had high or low stretch in the tree, resulting in better or worse payoffs for Alice, respectively. The process is then repeated in the next round. Once all rounds are complete, Alice's final strategy is computed as the average of each round's distribution $x^{(t)}$. Bob's final strategy is computed as the average over the fixed distributions from each round, giving us a distribution over low-stretch trees.

Some distinctions should be noted. Firstly, the trees are computed on demand, since we don't know which trees will be in the distribution before executing the method. Therefore, the matrix M is not explicitly computed; rather, we keep track of the trees as they are computed, and calculate the stretch as needed. Also, since the same tree may be generated in multiple rounds, we only have an upper bound, ρ, on the number of trees in the distribution. In the case of tree repetition, however, we will simply have some probabilities equal to zero. In addition, Arora et al. restrict payoff values to the range $[0, 1]$. Therefore, at each round, stretch values are normalized according to the maximum stretch of that round.

5.2.2 Tree Oracle

In order for this method to succeed, we need a way to compute a tree which has low stretch with respect to a distribution over edges. If we think in regards to the payoff matrix scenario, the TREEORACLE method must find some tree T_j such that $(x^\intercal M)_j$ is minimized, where x is Alice's distribution over edges:

$$\arg\min_j (x^\intercal M)_j = \arg\min_j \sum_i x_i M_{i,j} = \arg\min_j \sum_i x_i s_{T_j}(e_i)$$

Algorithm 2 Multiplicative Weights Update Method for Low-Stretch Trees

function MWUM-LST(δ, G)

input: Desired error δ, graph $G = (V, E = (e_1, \ldots, e_m), w)$

output: Distributions $\hat{x} \in \mathbb{R}^m$ and $\hat{y} \in \mathbb{R}^\rho$, and list of trees $\mathbb{T}$

// Set parameters

$\epsilon = \delta/2,\ \rho = \lceil \ln(m)/\epsilon^2 \rceil$

$\mathbb{T} = [\ ]$

$\gamma_i^{(1)} = 1$ for $i = 1, \ldots, m$

// Perform iterative rounds

for $t = 1, \ldots, \rho$ **do**

$x^{(t)} = \gamma^{(t)} / \sum_{i=1}^m \gamma_i^{(t)}$

$T^{(t)} = \text{TREEORACLE}(x^{(t)}, G)$

if $T^{(t)} \notin \mathbb{T}$ **then**

$k = |\mathbb{T}| + 1$

$\mathbb{T}[k] = T^{(t)}$

$y_i^{(t)} = \begin{cases} 0 & \mathbb{T}[i] \neq T^{(t)} \\ 1 & \mathbb{T}[i] = T^{(t)} \end{cases}$ for $i = 1, \ldots, \rho$

$s_{\max} = \max_i s_{T^{(t)}}(e_i)$

$s_i = s_{T^{(t)}}(e_i)/s_{\max}$ for $i = 1, \ldots, m$ *// Normalize to fit range* $[0, 1]$

$\gamma_i^{(t+1)} = \gamma_i^{(t)} \cdot (1 + \epsilon \cdot s_i)$ for $i = 1, \ldots, m$

$\hat{x} = \sum_{t=1}^\rho x^{(t)}/\rho,\ \hat{y} = \sum_{t=1}^\rho y^{(t)}/\rho$

return $\hat{x}, \hat{y}, \mathbb{T}$

That is, if we consider Alice's distribution to be an importance weighting of the edges in the graph G, then Bob will want to choose a tree that minimizes the stretch of edges according to their importance. Therefore, we define the following weighted stretch value for graph $G = (V, E, w)$ and tree $T = (V, E_T, z)$:

$$s_T(G, x) = \sum_{e_i \in E} x_i \cdot s_T(e_i) = \sum_{e_i = (u,v) \in E} x_i \cdot \frac{d_T(u, v)}{w(e_i)}$$

where x is a distribution over edges in G, so $\sum_i x_i = 1, x_i \geq 0\ \forall i$. In the uniform case, we have $x_i = \frac{1}{|E|}\ \ \forall e_i \in E$, which gives $s_T(G)$, as expected.

Therefore, we need an algorithm which computes a low-weighted-stretch tree according to the value $s_T(G, x)$ for graph $G = (V, E, w)$. We know the algorithm

of Alon et al. finds a low-stretch tree $T = (V, E_T, w)$ according to the value

$$s_T(G) = \sum_{e_i \in E} s_T(e_i) \cdot \frac{1}{|E|} = \sum_{e_i=(u,v)\in E} \frac{d_T(u,v)}{w(e_i)} \cdot \frac{1}{|E|}$$

Therefore, let us re-weight the graph G with weights $z : E \to \mathbb{R}_{\geq 0}$ where

$$z(e_i) = \frac{w(e_i)}{x_i \cdot |E|}$$

Now, the algorithm will compute a tree $T = (V, E_T, z)$ according to the value

$$s_T(G = (V, E, z)) = \sum_{e_i \in E} s_T(e_i) \cdot \frac{1}{|E|} = \sum_{e_i=(u,v)\in E} \frac{d_T(u,v)}{z(e_i)} \cdot \frac{1}{|E|} = \sum_{e_i=(u,v)\in E} x_i \cdot \frac{d_T(u,v)}{w(e_i)}$$

which is equivalent to our weighted stretch value, under T weighted with z. Note also that in the uniform case, this is equivalent to $s_T(G = (V, E, w))$, since $z(e_i) = w(e_i)$ when $x_i = \frac{1}{|E|}$ for all i.

Our oracle, shown in Algorithm 3, will therefore re-weight the graph according to weights z, so that the low-stretch tree $T = (V, E_T, z)$ returned from Algorithm 1 will have low weighted stretch.

Algorithm 3 Compute low-stretch tree WRT distribution over edges

function TREEORACLE(x, G)
  input: Distribution x over edges, graph $G = (V, E = (e_1, \ldots, e_m), w)$
  output: Tree $T = (V, E_T, z)$ in G with low stretch WRT distribution x
  $z(e_i) = w(e_i)/x_i \cdot m$ for $i = 1, \ldots, m$
  T = LOWSTRETCHTREE($G = (V, E, z)$)
  return $T = (V, E_T, z)$

5.2.3 Analysis

We wish to show that the distribution computed by Algorithm 2 provides low stretch in expectation for all edges in E. That is, for all edges $e_i \in E$, $\mathbb{E}_{T\sim D}[s_T(e_i)]$

is small. As discussed earlier, for some e_i, we have:

$$\mathbb{E}_{T\sim D}[s_T(e_i)] = \sum_{T\in D} \Pr[T] \cdot s_T(e_i) = (My)_i$$

In order to show that this is small, we will use the theorem from Arora et al. [6]:

Theorem 5.2.3.1 (adapted from Arora et al. [6])**.** *Given an error parameter* $\delta \in (0,1)$*,* MWUM-LST *finds* $\hat{x}$ *and* $\hat{y}$ *such that:*

$$\min_y \hat{x}^\top M y \geq \lambda^* - \delta \quad \textit{and} \quad \max_x x^\top M \hat{y} \leq \lambda^* + \delta$$

using $O(\ln(m)/\delta^2)$ *calls to* TREEORACLE*, with an additional processing time of* $O(m)$ *per call.*

This theorem follows nicely from the next two results.

Lemma 5.2.3.2 (Harvey [16])**.** *For all i,*

$$\sum_{t=1}^{\rho} \frac{x^{(t)\top} M y^{(t)}}{\rho} \;\geq\; \sum_{t=1}^{\rho} \frac{M_{i,j^{(t)}}}{\rho} - \delta$$

where $j^{(t)}$ *is the index in* $\mathbb{T}$ *of the tree* $T^{(t)}$ *chosen at round t, such that* $y^{(t)}_{j^{(t)}} = 1$*.*

This lemma shows that the algorithm's performance will not be much worse than choosing a fixed action throughout. The key thing to note is that this holds for all possible fixed actions, including the optimal. The proof of this lemma is given in Section A.2.

Corollary 5.2.3.3 (Harvey [16])**.** *For any distribution* $q \in \mathbb{R}^m$*, we have*

$$\sum_{t=1}^{\rho} \frac{q^\top M y^{(t)}}{\rho} - \delta \;\leq\; \sum_{t=1}^{\rho} \frac{x^{(t)\top} M y^{(t)}}{\rho} \;\leq\; \lambda^*$$

The lower bound follows from Lemma 5.2.3.2. Since q is a distribution, we know $\sum_{i=1}^m q_i = 1$. Therefore, we can multiply both sides by the sum, and simplify the RHS to obtain our result. The upper bound comes from the observation that $x^{(t)} M y^{(t)} = \min_y x^{(t)} M y \leq \lambda^*$. More details can be found in the full proof [16].

We are now ready to prove Theorem 5.2.3.1.

Proof (Theorem 5.2.3.1). In particular, let's focus on the right-hand guarantee. The left-hand case is similar [16]. Let $\hat{y} = \sum_{t=1}^{\rho} y^{(t)}/\rho$ be the result returned from the algorithm. Apply Corollary 5.2.3.3 to the distribution $x = \arg\max_x x^\intercal M\hat{y}$. Then we have:

$$x^\intercal M\hat{y} = \sum_{t=1}^{\rho} \frac{x^\intercal M y^{(t)}}{\rho} \leq \lambda^* + \delta$$

In terms of running time, we know that $\rho = \ln(m)/\epsilon^2 = 4\ln(m)/\delta^2$ rounds are performed, and TREEORACLE is called once per round. Within each round, we also update all weights γ_i for $i = 1, \ldots, m$. Therefore, we have $O(\ln(m)/\delta^2)$ calls to TREEORACLE, and perform an additional $O(m)$ work each round. □

Theorem 5.2.3.1 shows that the MWUM-LST algorithm returns a distribution $\hat{y}$ such that

$$\max_x x^\intercal M\hat{y} \leq \lambda^* + \delta$$

where Alon et al. show that $\lambda^* \leq \exp(O(\sqrt{\log n \log\log n}))$, since their tree stretch is upper bounded by that value. This implies that $(M\hat{y})_i$ is small for all i. Therefore, the distribution provides low expected stretch for all edges.

5.2.4 Application to LSTB

In order to utilize this distribution as a routing graph for LSTB, we must determine how to perform routing and how to draw the resulting bundled graph. For routing, we use the simple approach of choosing the shortest path among the trees in the distribution. That is, for some edge $e = (u, v) \in E$, its route through the distribution is the unique path between u and v in T, where $T = \arg\min_{T \in D} d_T(u, v)$. To draw the distribution, we combine the trees into a single routing graph G_R with vertices V and edges $\bigcup_{T=(V,E_T)\in D} E_T$. We lay out G_R using a force-directed layout algorithm. Bundled edges are then drawn as described in Section 4.2.2.

Section 6.3 shows the results of using a distribution over trees for routing in LSTB. For these results, the size of the distribution was fixed by setting the ρ parameter explicitly.

Chapter 6

Results

This chapter presents the results of this work, including visual output and performance of LSTB and experiments with distributions over trees.

6.1 Data and Implementation

The data sets listed in Table 6.1 were chosen based on their use in previous work to enable comparisons to existing methods. The "Case" column indicates which case from Table 1.1 the data falls under.

Data set	$\lvert V\rvert$	$\lvert E\rvert$	**Description**	**Case**
Flare	220	708	Software class hierarchy[1]	$H_{\checkmark}L_{\times}$
US Airlines	235	1297	US airline network[2]	$H_{\times}L_{\checkmark}$
Poker	859	2127	Poker game network[3]	$H_{\times}L_{\times}$
Email	1133	5451	Email interchange network[4]	$H_{\times}L_{\times}$
Yeast	2224	6609	Protein interaction network[4]	$H_{\times}L_{\times}$
wiki-Vote	7066	100736	Wikipedia admin elections[4]	$H_{\times}L_{\times}$

Table 6.1: Graph statistics for data sets used in this paper.

[1] https://gist.github.com/mbostock/1044242
[2] https://github.com/upphiminn/d3.ForceBundle/tree/master/example/bundling_data
[3] Courtesy of A. Telea [19]
[4] http://yifanhu.net/GALLERY/GRAPHS/

LSTB was implemented in Python and JavaScript, using D3 [9] for drawing. Our experiments were run on a MacBook Pro with a 2.6 GHz Intel Core i7 and 16 GB of 1600 MHz DDR3 RAM.

6.2 LSTB Evaluation

The visual results of LSTB are layouts that closely follow the spanning tree backbones. Figure 6.1 compares LSTB to previous edge bundling systems: KDEEB [19], Clustered Edge Routing [10], MINGLE [14], SBEB [13], and HEB [17]. We use the same layouts as in previous work for the layout-based methods, and when none was available we provide an arbitrary layout. For both the wiki-Vote (Figures 6.1a and 6.1b) and Yeast (Figures 6.1c and 6.1d) data sets, the bundled graphs have a tree-like structure, but LSTB makes it easier to identify clusters of vertices and disparities in their sizes. For the Email data, shown in Figures 6.1e and 6.1f, Clustered Edge Routing [10] shows a mesh-like structure in contrast to our tree layout, so choosing the more appropriate method would depend on the task at hand. The US Airlines data has a geographical layout, so the layout-based bundling shown in Figure 6.1g would be suitable for tasks that pertain to spatial position. We show our method of drawing bundles with that geographic layout for comparison purposes in Figure 6.1h, in contrast to the layout computed by LSTB in Figure 6.1i that emphasizes different patterns in the data. Figures 6.1j-l show the Flare data set, which is a software package hierarchy where edges represent imports of one package from another. While the LST used as a routing graph is not the original package hierarchy, we can see from the package names that similar packages are still clustered together, and hovering over edge bundles makes it easy to see which packages call each other.

Our approach achieves competitive performance with the fastest previous method, MINGLE [14]. Table 6.2 gives running times for our method, which range from 0.06 seconds for a 708-edge graph to 8.067 seconds for a 100736-edge graph. On this large graph, MINGLE reports a running time of 18.4 seconds, albeit on an older architecture. We note that while MINGLE is implemented in C using OpenGL on a GPU, our proof-of-concept implementation uses unoptimized scripting languages; further speed improvements would result from a GPU port.

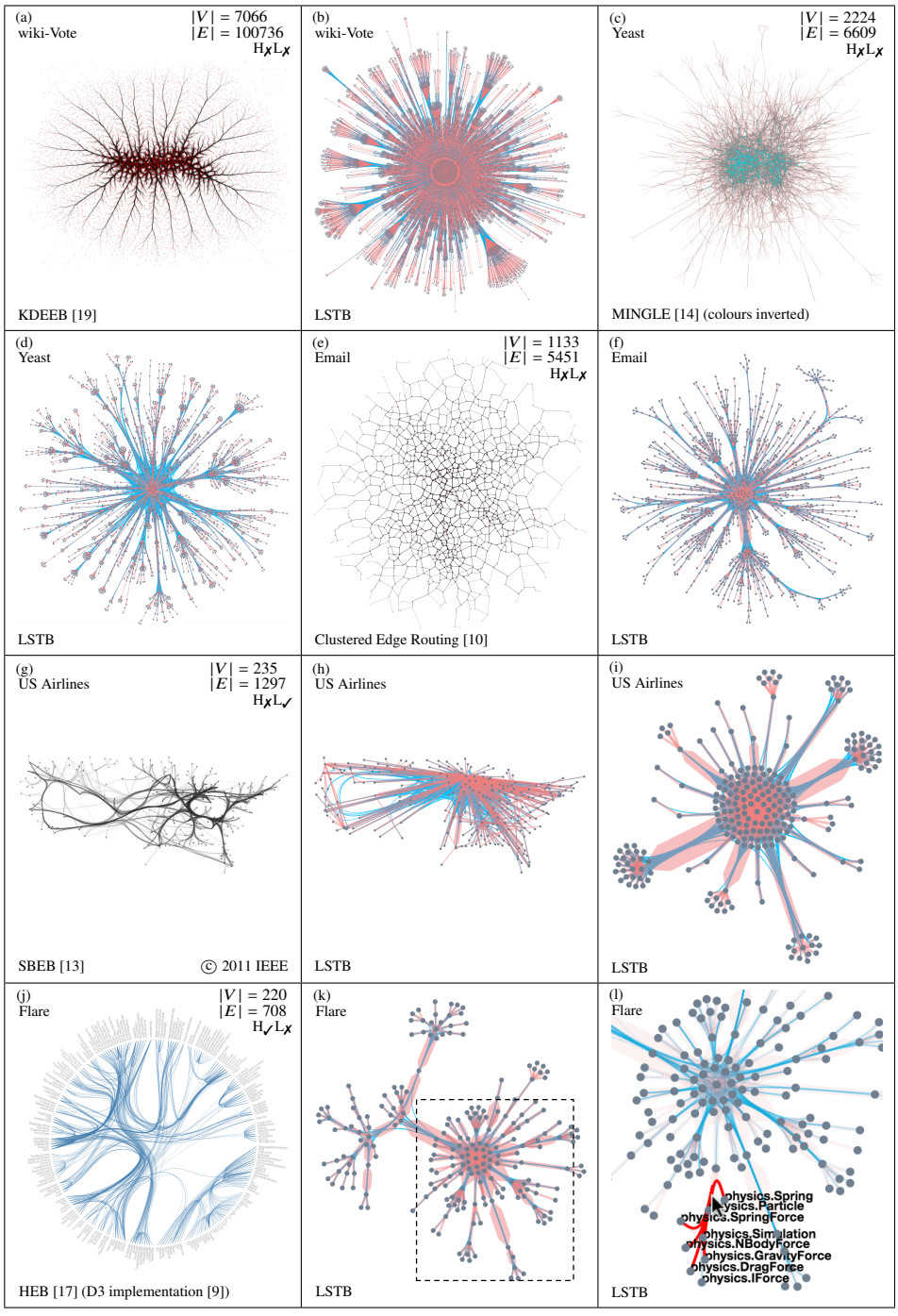


Figure 6.1: Comparison between previous methods and LSTB.

Data set	$\lvert V\rvert$	$\lvert E\rvert$	LST	Bundling	Drawing	Total
Flare	220	708	0.022	0.005	0.032	0.060
US Airlines	235	1297	0.035	0.009	0.042	0.086
Poker	859	2127	0.082	0.026	0.108	0.216
Email	1133	5451	0.180	0.053	0.283	0.516
Yeast	2224	6609	0.247	0.070	0.342	0.659
wiki-Vote	7066	100736	4.275	1.010	2.782	8.067

Table 6.2: Performance statistics of LSTB. Running time is in seconds, and is averaged over 100 runs.

6.3 Distribution over Low-Stretch Trees

The aim of computing a distribution D over low-stretch trees versus a single tree is to obtain low stretch in expectation for each edge in the graph. We can validate the results of our experiments by comparing the maximum expected stretch of any edge over the distribution, $\max_{e \in E} \mathbb{E}_{T \sim D}[s_T(e)]$ to the maximum stretch of any edge in a single tree $\mathcal{T}$, $\max_{e \in E} s_{\mathcal{T}}(e)$. These results are given in Table 6.3. In all distribution computations we use $\delta = 0.5$.

		$\max_{e \in E} \mathbb{E}_{T \sim D}[s_T(e)]$		
Data set	$\max_{e \in E} s_{\mathcal{T}}(e)$	$\lvert D\rvert = 2$	$\lvert D\rvert = 5$	$\lvert D\rvert = 10$
Flare	9.00	7.10	8.36	8.13
US Airlines[5]	34.08	34.30	32.24	25.90
Poker	12.00	12.15	10.82	12.07
Email	8.00	7.97	8.34	9.02
Yeast	10.00	10.08	10.19	11.74
wiki-Vote	6.00	6.03	6.29	6.97

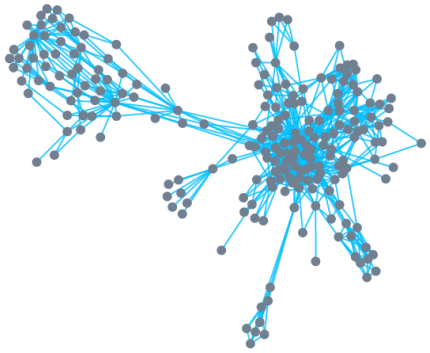
Table 6.3: Maximum edge stretch comparison between a single low-stretch tree $\mathcal{T}$ and a distribution D over low-stretch trees, computed as in Section 5.2 with varying sizes $\lvert D\rvert \in \{2, 5, 10\}$.

From these results we can see that the distribution generally fulfills its purpose of ensuring the expected stretch is low for each edge, versus the single tree. In

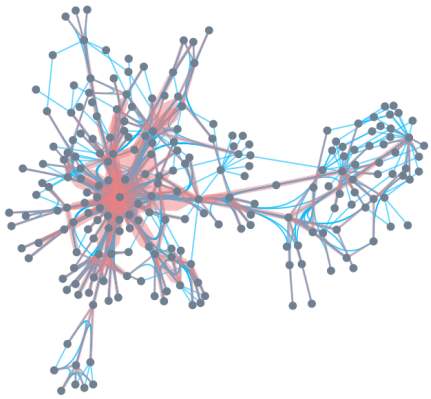
[5]This graph has a non-uniform input weighting.

a few cases, however, the maximum expected stretch goes up compared to the maximum stretch in the single low-stretch tree. Since there is no monotonicity guarantee, the stretch may not go down at each round. Also, since the guarantees given in Section 5.2 are for a distribution of size $\rho = \lceil \ln m/\epsilon^2 \rceil$ and we compute distributions of constant size, we may not obtain the optimal result.

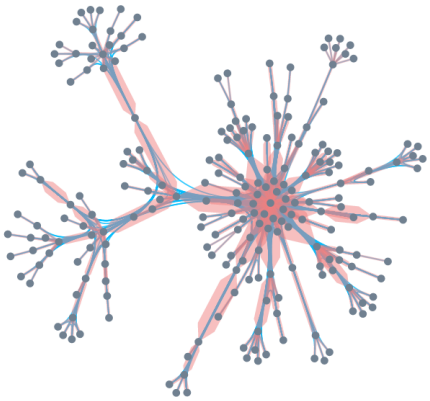
Next we will evaluate the results of applying the distribution to LSTB, as discussed in Section 5.2.4. Figure 6.2 shows the output of LSTB when using a distribution over two low-stretch trees as the routing graph. When remainder edges and bundles are shown, there is a large amount of visual clutter and it is hard to distinguish patterns among edges and bundles. This is particularly evident when comparing this result, shown in Figure 6.2b, to the original output of LSTB with a single tree, shown in Figure 6.2c. These issues are only worsened when larger data sets are used, and for larger distributions. This is discussed in more detail in Section 7.3.



(a) Force layout of unbundled Flare data.



(b) Bundled result using two trees as the routing graph (with 317 edges). While this visualization is still somewhat cluttered, the maximum expected stretch for any edge is 7.1.



(c) Bundled result using one tree (original LSTB output) as the routing graph (with 219 edges). This visualization is less cluttered than (b), but has worse maximum stretch of 9.0.

Figure 6.2: Comparison between visual results obtained on the Flare data set using a distribution over two low-stretch trees (b) and a single low-stretch tree (c) for routing in LSTB.

Chapter 7

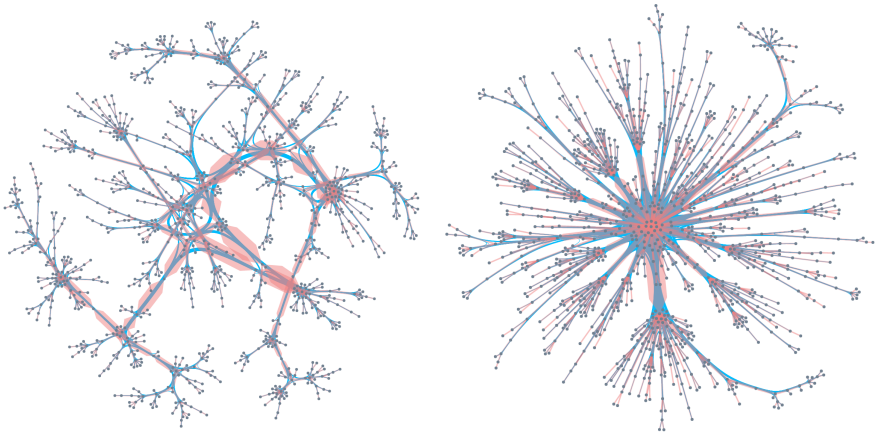
Discussion

Several interesting issues arise when using low-stretch trees for a network visualization application. This chapter will discuss some of them, including the use of trees for routing in edge bundling, the deeper meaning behind "low" stretch, and difficulties in drawing a distribution over trees. We will also evaluate our technique of using low-stretch trees and a distribution over those trees for edge bundling. This chapter concludes with ideas for future work.

7.1 LSTB

The crux of this thesis is the use of low-stretch trees for edge bundling. While routing through a tree results in similar-looking bundled graphs for most data sets, using a tree provides the sparsity needed for an uncluttered bundling. That said, LSTB specifically uses a low-stretch tree to route edges. Section 4.1.1 explains the reasoning behind this decision, but one might also wonder why a different type of spanning tree is not used. For example, a minimum spanning tree might seem appropriate, since it minimizes its total weight. In the unweighted case, any spanning tree is a minimum spanning tree. In either case, however, this tree structure depends very little on the edges that are not included in the tree, despite the fact that these edges are crucial for edge bundling. Edges may therefore be distorted by having endpoints far apart in the tree. To illustrate this, Figure 7.1 shows a comparison between an arbitrary spanning tree and a low-stretch tree for the unweighted

Email data set. In Figure 7.1a, the large bundles along the interior edges imply many edges are being routed from exterior branches through the middle, which means endpoints from edges in the graph are far apart in the tree. This congestion distorts the edges of the original graph.



(a) Arbitrary spanning tree routing graph.

(b) Low-stretch tree routing graph.

Figure 7.1: Comparison between using an arbitrary spanning tree and a low-stretch tree as the routing graph in LSTB on the Email data set.

7.2 Stretch

From the name, we have an intuitive sense of what *stretch* is. From the definition, we know it is the ratio between the distance between edge endpoints in a tree and the weight of that edge in the original graph. However, we are also interested in the meaning of stretch. In particular, what does it mean to have *low* stretch? We have seen two quantities: low stretch for edges on average, and low stretch for every edge. Most low-stretch tree constructions focus on the former [1, 3, 12], while computing a distribution over trees focuses on the latter [2]. As in many situations, there is a trade-off: we can achieve low stretch in expectation for every edge, but we must accept a distribution; on the other hand, we can compute a single tree, but must be satisfied with low average stretch.

In this work, we faced another trade-off in choosing which low-stretch tree

construction to use. We chose the simpler method of Alon et al., but must use the multiplicative weights update method [6] to obtain a distribution with low expected stretch for all edges. Had we chosen the method of Abraham et al., we would be able to sample directly from the distribution, but the implementation would be far more complex (and likely impractical).

In the unweighted (or uniformly weighted) case, stretch is far easier to understand. The "units" are edges: if two vertices were connected in the original graph, then their stretch corresponds to the number of edges separating them in a given tree. In the weighted case, however, things get more complex, particularly when comparing stretches. In such situations, the meaning of stretch depends on the weighting. For geographical data, such as the US Airlines data set (see Table 6.1), edge weights measure physical distance. In other cases, though, the weighting may be less clear. In these situations, stretch results should be thought about from the context of the weighting, rather than edge quantity.

7.3 Distribution over Low-Stretch Trees

Based on the results shown in Section 6.3, our distribution over (few) low-stretch trees performs generally as expected: in most cases, the distribution reduces the maximum stretch of individual edges. However, with a constant number of trees in the distribution and no monotonicity guarantee, this may not always be the case. This caveat should be taken into account when evaluating the use of a distribution of low-stretch trees for routing in LSTB.

Another factor for evaluation is the method by which edges are routed through the distribution. Choosing to route an edge through the tree with the shortest path between its endpoints is simple, but may be computationally expensive. Randomly sampling a tree from the distribution for each edge is fast, but may not provide the best stretch. The most important factor to take into consideration, however, is the visual aspect. Drawing a routing graph which consists of multiple trees presents several difficulties. One such issue is the representation of weights. The trees have a uniform probability of being sampled from the distribution, but the tree edges are weighted. It is unclear what the best way to represent these weights is.

The main problem, though, is visual clutter. In graph theory, any graph with

$O(n)$ edges is considered extremely sparse. In network visualization, however, there is a considerable difference between n edges and $2n$ edges. While increasing the size of the distribution leads to better theoretical results, more edges in the routing graph results in more visual clutter. The graph is more difficult to lay out, and the number of bundles is increased, making them harder to distinguish. These effects are highlighted in Figure 6.2. After taking these issues into consideration, it seems that a single low-stretch tree provides a better routing graph for the LSTB edge bundling application.

7.4 Future Work

We present several ideas for further research in this area. In terms of improving LSTB, different graph layouts and visual encodings could be explored. It would also be interesting to investigate the adaptation of LSTB for layout-based bundling. Ultimately, a united system of both layout-based and layout-free edge bundling should be developed.

Rather than using the method of Alon et al. [3] for computation of low-stretch trees, other methods with better theoretical guarantees could be tried. Along this vein, using the method of Abraham et al. [2] would remove the dependence on the multiplicative weights update method for a distribution over low-stretch trees. In either case, however, the implementation difficulties may outweigh the benefits of these other methods.

Other sparsifiers may also be tried as routing graphs for edge bundling. As mentioned in Section 7.3, there is a notable difference between n and $2n$ edges for visualization. However, new approaches such as Kolla et al. [20] compute *ultrasparsifiers*, which have $n + o(n)$ edges. A famous sparsification result by Batson et al. [7] proved too slow for practical purposes in early experiments, but new work by Lee and Sun [22] improves this with a near-linear time algorithm.

In addition, low-stretch trees could be applied to other areas of network visualization. They are particularly useful for applications which require both the preservation of edge distances and a sparse representation of the original graph.

Chapter 8

Conclusion

We present LSTB, a novel edge bundling technique which is, by our classification, layout-free. While previous bundling methods rely on an input graph layout or explicit hierarchical structure, we use topological features of the graph in order to compute a low-stretch tree which we use to route edges. Our bundling method is fast and simple, and provides algorithmic support for sophisticated visual encodings and interactivity. In addition, our abstract framework for edge bundling presents a formalization of bundling terminology and techniques that allows bundling methods to be compared in a uniform way.

Our application of the multiplicative weights update method to a zero-sum game over edges and trees enables the computation of a distribution over low-stretch trees. This distribution ensures all edges in the original graph have low stretch in expectation. We apply this distribution as a routing graph for LSTB, but our analysis shows that a single low-stretch tree obtains better visual results.

Bibliography

[1] I. Abraham and O. Neiman. Using petal-decompositions to build a low stretch spanning tree. In *Proc. ACM Symp. Theory of Computing (STOC)*, pages 395–406, 2012. ISBN 978-1-4503-1245-5. → pages 8, 39

[2] I. Abraham, Y. Bartal, and O. Neiman. Nearly tight low stretch spanning trees. In *Proc. IEEE Symp. Foundations of Computer Science (FOCS)*, pages 781–790, 2008. → pages 8, 16, 25, 39, 40, 41

[3] N. Alon, R. M. Karp, D. Peleg, and D. West. A graph-theoretic game and its application to the k-server problem. *SIAM Journal on Computing*, 24(1): 78–100, 1995. ISSN 0097-5397. → pages 3, 4, 5, 7, 16, 22, 23, 24, 25, 26, 29, 31, 39, 40, 41

[4] I. Althöfer, G. Das, D. Dobkin, and D. Joseph. Generating sparse spanners for weighted graphs. In *Scandinavian Workshop on Algorithm Theory*, volume 447 of *Lecture Notes in Computer Science*, pages 26–37. Springer Berlin Heidelberg, 1990. ISBN 978-3-540-52846-3. → pages 9

[5] D. Archambault, T. Munzner, and D. Auber. Smashing peacocks further: Drawing quasi-trees from biconnected components. *Visualization and Computer Graphics, IEEE Transactions on*, 12(5):813–820, 2006. → pages 10

[6] S. Arora, E. Hazan, and S. Kale. The multiplicative weights update method: a meta-algorithm and applications. *Theory of Computing*, 8(1):121–164, 2012. → pages 5, 26, 27, 30, 40

[7] J. Batson, D. A. Spielman, and N. Srivastava. Twice-ramanujan sparsifiers. *SIAM Journal on Computing*, 41(6):1704–1721, 2012. → pages 41

[8] E. Boman and B. Hendrickson. On spanning tree preconditioners. *Manuscript, Sandia National Lab*, 2001. → pages 7

[9] M. Bostock, V. Ogievetsky, and J. Heer. D3: Data-driven documents. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)*, 17(12):2301–2309, 2011. → pages 19, 33, 34

[10] Q. W. Bouts and B. Speckmann. Clustered edge routing. In *Proc. IEEE Pacific Visualization Symp. (PacificVis)*, 2015. To appear. → pages 2, 9, 14, 15, 33, 34

[11] W. Cui, H. Zhou, H. Qu, P. C. Wong, and X. Li. Geometry-based edge clustering for graph visualization. *IEEE Trans. Visualization & Comp. Graphics*, 14(6):1277–1284, Nov 2008. ISSN 1077-2626. doi:10.1109/TVCG.2008.135. → pages 2, 8

[12] M. Elkin, Y. Emek, D. A. Spielman, and S.-H. Teng. Lower-stretch spanning trees. *SIAM Journal on Computing*, 38(2):608–628, 2008. → pages 8, 16, 39

[13] O. Ersoy, C. Hurter, F. V. Paulovich, G. Cantareiro, and A. Telea. Skeleton-based edge bundling for graph visualization. *IEEE Trans. Visualization & Comp. Graphics*, 17(12):2364–2373, Dec 2011. ISSN 1077-2626. doi:10.1109/TVCG.2011.233. → pages 9, 33, 34

[14] E. R. Gansner, Y. Hu, S. North, and C. Scheidegger. Multilevel agglomerative edge bundling for visualizing large graphs. In *Proc. IEEE Pacific Visualization Symp. (PacificVis)*, pages 187–194, 2011. → pages 9, 13, 33, 34

[15] S. Hachul and M. Jünger. Drawing large graphs with a potential-field-based multilevel algorithm. In *Graph Drawing*, volume 3383 of *Lecture Notes in Computer Science*, pages 285–295. Springer Berlin Heidelberg, 2005. → pages 9

[16] N. Harvey. Lecture notes in mathematical programming. http://www.math.uwaterloo.ca/~harvey/F10/Lecture12Notes.pdf, 2010. → pages 30, 31, 47

[17] D. Holten. Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. *IEEE Trans. Visualization & Comp. Graphics*, 12(5): 741–748, Sept 2006. ISSN 1077-2626. doi:10.1109/TVCG.2006.147. → pages viii, 2, 3, 6, 8, 20, 33, 34

[18] D. Holten and J. J. van Wijk. Force-directed edge bundling for graph visualization. *Computer Graphics Forum*, 28(3):983–990, 2009. ISSN 1467-8659. → pages 9, 13, 14

[19] C. Hurter, O. Ersoy, and A. Telea. Graph bundling by kernel density estimation. *Computer Graphics Forum*, 31(3pt1):865–874, 2012. → pages 9, 32, 33, 34

[20] A. Kolla, Y. Makarychev, A. Saberi, and S.-H. Teng. Subgraph sparsification and nearly optimal ultrasparsifiers. In *Proc. ACM Symp. Theory of Computing (STOC)*, pages 57–66, New York, NY, USA, 2010. ACM. → pages 41

[21] A. Lambert, R. Bourqui, and D. Auber. Winding roads: Routing edges into bundles. *Computer Graphics Forum*, 29(3):853–862, 2010. → pages 2, 9

[22] Y. T. Lee and H. Sun. Constructing linear-sized spectral sparsification in almost-linear time. In *IEEE Symp. Foundations of Computer Science (FOCS)*, 2015. To appear. → pages 41

[23] T. Munzner. *Visualization Analysis and Design*, chapter 4, pages 67–93. A K Peters Visualization Series. CRC Press, 2014. → pages 1, 5, 6

[24] T. Munzner, F. Guimbretiere, and G. Robertson. Constellation: A visualization tool for linguistic queries from MindNet. In *Proc. IEEE Symp. Information Visualization (InfoVis)*, pages 132–135, 154, 1999. → pages 10

[25] S. Pupyrev, L. Nachmanson, and M. Kaufmann. Improving layered graph layouts with edgebundling. In *Graph Drawing*, volume 6502 of *Lecture Notes in Computer Science*, pages 329–340. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-18468-0. → pages 9

[26] S. Pupyrev, L. Nachmanson, S. Bereg, and A. E. Holroyd. Edge routing with ordered bundles. In *Graph Drawing*, volume 7034 of *Lecture Notes in Computer Science*, pages 136–147. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-25877-0. → pages 2, 9

[27] E. M. Reingold and J. S. Tilford. Tidier drawings of trees. *IEEE Trans. Software Engineering*, SE-7(2):223–228, March 1981. → pages 19

[28] N. K. Vishnoi. $Lx = b$. Foundations and Trends in Theoretical Computer Science. NOW, 2013. → pages 4, 8

[29] J. von Neumann. Zur theorie der gesellschaftsspiele. *Mathematische Annalen*, 100(1):295–320, 1928. → pages 7, 26, 47

Appendix A

Supporting Materials

A.1 Zero-Sum Games and the Minimax Theorem

Consider a game between two players, Alice and Bob. Let M be the payoff matrix such that, on a given turn, if Alice chooses row i and Bob chooses row j, Bob must pay Alice the value $M_{i,j}$. Therefore, Alice wishes to maximize her payoff from M and Bob wishes to minimize it. We will formulate such a zero-sum game for low-stretch trees, where the payoff values will be the stretch of a given edge in a given tree.

For graph $G = (V, E)$, choose an arbitrary ordering of edges $e_1, \ldots, e_m$ where $E = \{e_i : 1 \leq i \leq m\}$. Let the edges $e_i = (u, v)$ of G correspond to the rows of M, and let there be ρ columns, each one corresponding to some (currently undefined) spanning tree T_j. Then the payoff values will be the stretch of a given edge in a given tree, so $M_{i,j} = s_{T_j}(e_i)$. Alice wishes to choose an edge e_i that will have high stretch in whichever tree Bob chooses, and Bob wishes to choose a tree T_j that will have low stretch for whichever edge Alice chooses.

Let $x \in \mathbb{R}^m$ be a distribution over edges, and $y \in \mathbb{R}^\rho$ be a distribution over trees, such that $\sum_{i=1}^{m} x_i = 1$ and $\sum_{j=1}^{k} y_j = 1$, and all x_i and y_j are non-negative. These distributions represent player strategy, where x_i denotes the probability that Alice chooses edge e_i, and y_j denotes the probability that Bob chooses tree T_j. The expected value of Alice's payoff for choosing edge e_i, given Bob is playing by strategy y, is $(My)_i$. Likewise, the expected value of Bob's payment for choosing

tree T_j, given Alice is playing by strategy x, is $(xM)_j$. By von Neumann's Minimax Theorem [29], there exists a distribution x^* over edges and a distribution y^* over trees such that:

$$\max_x \min_y x^\mathsf{T} M y = x^{*\mathsf{T}} M y^* = \min_y \max_x x^\mathsf{T} M y$$

where both players achieve the best possible payoff given the other's strategy. We denote the value of the game as $\lambda^* = x^{*\mathsf{T}} M y^*$.

A.2 Proof of Lemma 5.2.3.2

Adapted from Harvey [16]. Let $\Gamma^{(t)} = \sum_{i=1}^m \gamma_i^{(t)}$. Consider the algorithm's state at the start of round $t+1$.

$$\Gamma^{(t+1)} = \sum_{i=1}^m \gamma_i^{(t+1)} = \sum_{i=1}^m \gamma_i^{(t)} \cdot (1 + \epsilon \cdot s_i)$$

Recall that s_i is the normalized stretch of edge e_i in tree $T^{(t)}$. That is, $s_i = s_{T^{(t)}}(e_i)/s_{\max}$. Since we assume M is normalized, this is equivalent to $M_{i,j^{(t)}}$, where $j^{(t)}$ is the index in $\mathbb{T}$ of $T^{(t)}$ such that $y_{j^{(t)}}^{(t)} = 1$. Therefore, we also have $M_{i,j^{(t)}} = (My^{(t)})_i$. Therefore, we obtain:

$$\begin{aligned}
\Gamma^{(t+1)} &= \sum_{i=1}^m \gamma_i^{(t)} \cdot (1 + \epsilon \cdot (My^{(t)})_i) \\
&= \sum_{i=1}^m \gamma_i^{(t)} + \epsilon \gamma_i^{(t)} (My^{(t)})_i \\
&= \sum_{i=1}^m \gamma_i^{(t)} + \epsilon \sum_{i=1}^m \gamma_i^{(t)} (My^{(t)})_i \\
&= \Gamma^{(t)} + \epsilon \Gamma^{(t)} \sum_{i=1}^m x_i^{(t)} (My^{(t)})_i
\end{aligned}$$

where the last step uses that $x^{(t)} = \gamma^{(t)}/\Gamma^{(t)}$. Simplifying this expression gives us:

$$\Gamma^{(t+1)} = \Gamma^{(t)}(1 + \epsilon \cdot x^{(t)} M y^{(t)})$$

Using that $(1 + y) \leq e^y \ \forall y$, this becomes:

$$\Gamma^{(t+1)} \leq \Gamma^{(t)} \exp(\epsilon \cdot x^{(t)} M y^{(t)})$$

Now, consider the end of the algorithm, after ρ rounds have been performed. We know that the base case of the recurrence is $\Gamma^{(1)} = m$. Therefore, we obtain:

$$\begin{aligned}
\Gamma^{(\rho+1)} &\leq \Gamma^{(\rho)} \exp(\epsilon \cdot x^{(\rho)} M y^{(\rho)}) \\
&\leq \Gamma^{(1)} \prod_{t=1}^{\rho} \exp(\epsilon \cdot x^{(t)} M y^{(t)}) \\
&= m \cdot \exp\left(\epsilon \cdot \sum_{t=1}^{\rho} x^{(t)} M y^{(t)}\right)
\end{aligned}$$

where the product has been pulled inside the exponential. This gives us an upper bound on $\Gamma^{(\rho+1)}$; next, we will find a lower bound.

Note that any $\gamma_i^{(t+1)}$ is a lower bound for $\Gamma^{(t+1)}$ for all $i = 1, \ldots, m$ and $t = 1, \ldots, \rho$ since the weights are non-negative. Therefore,

$$\begin{aligned}
\Gamma^{(t+1)} &\geq \gamma_i^{(t+1)} \\
&\geq \gamma_i^{(t)} \cdot (1 + \epsilon \cdot M_{i,j^{(t)}})
\end{aligned}$$

Consider the result after ρ iterations. Here, our base case is $\gamma_i^{(1)} = 1$. Therefore,

$$\begin{aligned}
\Gamma^{(\rho+1)} &\geq \gamma_i^{(\rho+1)} \\
&\geq \prod_{t=1}^{\rho} (1 + \epsilon \cdot M_{i,j^{(t)}}) \\
&\geq \prod_{t=1}^{\rho} (1 + \epsilon)^{M_{i,j^{(t)}}}
\end{aligned}$$

since $(1 + \epsilon x) \geq (1 + \epsilon)^x$ for $x \in [0, 1]$ and $\epsilon \geq 0$. Putting the upper and lower

bounds together gives:

$$m \cdot \exp\left(\epsilon \cdot \sum_{t=1}^{\rho} x^{(t)} M y^{(t)}\right) \geq \Gamma^{(\rho+1)} \geq \prod_{t=1}^{\rho} (1+\epsilon)^{M_{i,j^{(t)}}}$$

Taking the natural logarithm of both sides gives:

$$\begin{aligned}
\ln m + \epsilon \cdot \sum_{t=1}^{\rho} x^{(t)} M y^{(t)} &\geq \sum_{t=1}^{\rho} M_{i,j^{(t)}} \cdot \ln(1+\epsilon) \\
\sum_{t=1}^{\rho} x^{(t)} M y^{(t)} &\geq \frac{1}{\epsilon}\left(\ln(1+\epsilon) \sum_{t=1}^{\rho} M_{i,j^{(t)}} - \ln m\right) \\
&\geq (1-\epsilon) \sum_{t=1}^{\rho} M_{i,j^{(t)}} - \frac{\ln m}{\epsilon} \\
&\geq \sum_{t=1}^{\rho} M_{i,j^{(t)}} - \rho\epsilon - \frac{\ln m}{\epsilon} \\
&\geq \sum_{t=1}^{\rho} M_{i,j^{(t)}} - 2\rho\epsilon
\end{aligned}$$

using the fact that $\ln(1+\epsilon) \geq \epsilon - \epsilon^2$, and $\sum_{t=1}^{\rho} M_{i,j^{(t)}} \leq \rho$. Dividing by ρ, we obtain our result:

$$\begin{aligned}
\sum_{t=1}^{\rho} \frac{x^{(t)} M y^{(t)}}{\rho} &\geq \sum_{t=1}^{\rho} \frac{M_{i,j^{(t)}}}{\rho} - 2\epsilon \\
&\geq \sum_{t=1}^{\rho} \frac{M_{i,j^{(t)}}}{\rho} - \delta
\end{aligned}$$

□