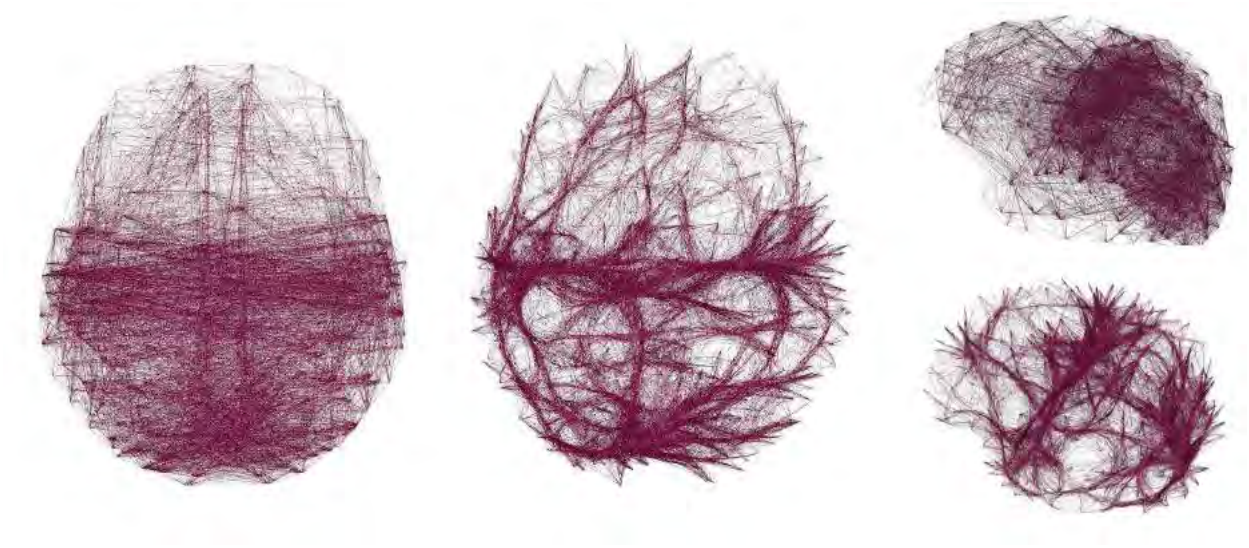

Figure 1: Screenshots of 3D rendered graph drawings showing functional connectivity in a human brain [1] (437 vertices, 8,031 edges, 3D, undirected), from two perspectives each original and bundled with the proposed VBEB. The bundling might, for instance, help to illustrate statistical differences between brain scans of different subjects or at different times, in case of degeneration or regeneration [1]. As the rendering and thus the bundling is natively 3D, the result is freely and interactively accessible for immersive analytics, e.g., in an immersive virtual environment using an HMD or CAVE.

Voxel-based edge bundling through direction-aware kernel smoothing

Daniel Zielasko[a], Xiaoqing Zhao[a], Ali Can Demiralp[a], Torsten W. Kuhlen[a], Benjamin Weyers[b]

[a] *Visual Computing Institute, RWTH Aachen University*
JARA Center for Simulation and Data Science
[b] *Department IV - Human-Computer Interaction, University of Trier*

Abstract

Relational data with a spatial embedding and depicted as node-link diagram is very common, e.g, in neuroscience, and edge bundling is one way to increase its readability or reveal hidden structures. This article presents a 3D extension to kernel density estimation-based edge bundling that is meant to be used in an interactive immersive analysis setting. This extension adds awareness of the edges' direction when using kernel smoothing and thus implicitly supports both directed and undirected graphs. The method generates explicit bundles of edges, which can be analyzed and visualized individually and as sufficient as possible for a given application context, while it scales linearly with the input size.

Keywords: graphs and networks, edge bundling, virtual reality, kernel smoothing

1. Introduction

Relational data with a spatial embedding have applications in various fields, e.g., neuroscience in macro [2, 3, 4] and micro scales [5], astrophysics, molecular chemistry [6], or material flows in factory planning [7], to name a few. For analysis, this data is often represented as a 3D node-link diagram. Furthermore, there is some evidence that even large networks without a natural spatial embedding can benefit from a layout and

exploration in 3D [8]. However, like their 2D counterparts, 3D node-link diagrams suffer from visual clutter and edge bundling is a well-studied field of research tackling this and reveal possible hidden structures (see Figure 1). Multiple fast and efficient algorithms have been developed and presented in scientific publications [9]. However, most of the existing methods support only 2D (see Table 1) visualization, while it is often assumed that the extension to 3D is trivial, which is often not the case. For instance, the great *highway metaphor* for anti-parallel bundles presented by Selassie et al. [10] works well in 2D but cannot be transferred into 3D easily. There is no trivial solution how to position edge bundles with opposite directions next to each other in a 3D space as there is no obvious correct direction to place them in and the direction should be consistent to avoid, e.g., winding spirals and new clutter. But this should just serve as a simple example of challenges that can occur when trying to transfer a 2D algorithm into 3D space. Another more technical example is that 3D graph visualizations benefit most from exploration in a stereo setting [11, 12], be it an HMD, CAVE or fish tank VR system, which raises additional issues, such as caring about correct non-orthogonal projection [13]. Then, the analysis of 3D data works best when it is freely explored as it facilitates the creation of a spatial mental model. But a constantly refreshing 2D bundling of the projected image of a 3D graph drawing is not consistent over changing viewpoints and thus not sufficient here as well. In stereo, this already breaks for a static image, as the two eyes already require a different viewpoint. In summary, 3D node-link diagrams require a fast 3D edge bundling, to which we want to contribute to with this work. We will refer to this requirement as **dim** in the following.

Beside the dimensionality, a feasible edge bundling algorithm should consider the edge direction. Selassie et al. [10] addressed this by a directional extension to the original force directed edge bundling algorithm (FDEB) presented by Holten et al. [14], which is to bundle directed graphs properly (**dir**). Many real-world examples describe relations that are directional and while it might be sufficient to ignore this in a specific analysis task, it might not be in another case, even with the same data set. Therefore, the method in this work is focused explicitly to support both.

While edge bundling may lead to new insights itself, it first and foremost reduces the amount of information simultaneously visualized [15]. In the best case, the bundling reduces clutter or noise so that the users may gain more in-depth insight into what is relevant. Nevertheless, their needs may change in the next phase of the analysis. Therefore, not only the strength of bundling should be adaptable, but the user might also need a deeper understanding of the bundles, e.g., which edges participate in a bundle and to which amount. This is especially true for weighted graphs, which additionally raise the need to draw weighted bundles. The interactive adaption of bundles as well as the consideration of weighted edges can be seamlessly integrated when the actual bundles are explicitly calculated as part of the data structure [13] rather than implicitly generated in the rendering by, e.g., moving lines or pixels closer together in image space [16], which results in a loss of the bundling information for later (re-)use and adaption. Thus, an edge bundling algorithm should be able to calculate explicit bundles (**trace**).

Bundling is usually not the primary purpose of an application; it is one tool among others. This raises two further requirements. First, it should be sufficiently efficient for an interactive application and, since there might not exist an algorithm constant in time, it should scale at most linearly with the graph's input size (**scale**). Additionally, a fast algorithm is one requirement for the interactive bundling of time-varying graphs. Second, there is no generically best or correct bundling as it depends on the application context and/or aesthetics. Thus, an algorithm needs specific and understandable *degrees of freedom* to be adapted to a particular application context but simultaneously not more than necessary, such that the user does not get lost in the configuration of the algorithm (**dof**). Several existing algorithms need a broad set of parameters, which are often very technical and decoupled from the application domain. The latter could lead to problems in the interpretation of bundled graphs. Furthermore, not every combination of parameters necessarily serves a valid or aesthetic result, and the user has to experiment with the combinations. The algorithm presented in this article does not solve this issue as well; however, we are able to deliver it with a set of default parameter values, which might work for general graphs.

The main contribution of this article is the presentation of an algorithm that extends graph bundling by kernel density estimation [16] to address the identified requirements for bundling edges in 3D and both, directed and undirected graphs. It generates explicit bundles of edges, while it scales linearly with the input size (number of edges). Finally, the method is easy to understand, implement, modify, and parameterize.

The rest of the article is structured as follows. In Section 2, we discuss related work and its relation to the proposed requirements. The actual algorithm and its pipeline are described in Section 3 in detail. Then, its visual results and runtime performance are presented in Section 4. Afterward, we discuss these results, compare them to related work, show possible limitations, and future work in Section 5. Finally, the article is concluded in Section 6.

2. Related Work

There are various ways to classify edge bundling algorithms [9]. One property is the character of the bundling operation, which is either explicit or implicit. Explicit methods do not directly bundle a graph visualization but a so-called intermediate structure, derived from the actual graph. A lot of the explicit algorithms require special graph characteristics, such as a hierarchy [17, 18]. However, explicit methods are working on general graphs as well [19, 20]. It is important to note that this concept is different from the explicit bundling we mentioned earlier as one of the identified requirements (**trace**). The latter refers to traceability that characterizes whether it is possible to relate a drawing or rather the drawn bundles (as data model of bundles) to the original graph (data structure), i.e., it is possible to map each edge of a graph to the bundled drawing, such that edges are related to bundles and vice versa. Explicit algorithms usually obtain this connection, but it is desirable to keep the connection also in implicit ones, as these do not suffer from problems with non-optimal intermediate structures [9]. Counterexamples are image-based algorithms [21, 22], which are implicit, as this class of algorithms works on a rasterized image and thus has usually no explicit mapping of the graph's data structure to the rendered (bundled) node-link diagram. However, they are very fast and scale with the input size of the graph (**scale**). Furthermore, they have shown to handle directed graphs as well (**dir**) [23]. Peysakhovich et al. [24] introduced an attribute-based version (ADEB), which compensates for some of the disadvantages to not being traceable, but still, it is neither possible to operate on the resulting graph simplification nor to semantically investigate the bundles as they only exist implicitly in the image space. As image-based algorithms always work in the 2D image plane, they cannot handle 3D graph visualizations (**dim**). Algorithms that are capable of 3D graph representations were first designed for subsets of graphs as trees [25, 18] or used a projection onto the surface of a sphere [26]. Böttger et al. [1] first proposed an approach based on Force-Directed Edge Bundling (FDEB) [14] and Kernel Density Estimation Edge Bundling (KDEEB) [16] that can handle 3D graph visualizations without such restrictions. Nevertheless, their approach neither considers directed edges (**dir**) nor does it compute explicit bundles (**trace**), which is also the case for Hurter et al.'s functional decomposition method [27]. The last three requirements (**dim, trace, dir**) are covered by 3dFDEB presented by Zielasko et al. [13], which ported FDEB to 3D and introduced an edge clustering step to handle its runtime complexity. Furthermore, it can bundle attribute-driven by just extending or modifying the feature vector of each edge that is used to perform the edge classification. However, even when arguably very rare [13], the theoretical runtime complexity of 3dFDEB is still $O(n^2)$ (where n is the number of edges), such as FDEB's runtime (**scale**). Furthermore, 3dFDEB violates the reduced parameter requirement (**dof**), similar to many of the algorithms mentioned above. While the number of actual parameters of the latter is small, it is far more important that they are not transparent to the user and even worse, for a force-based algorithm a *non-optimal* set of parameters can cause a muddle.

In conclusion, various algorithms exist, each filling a certain gap, but still, there is a lack especially of techniques that apply to 3D graph drawings without losing flexibility in the generated graph visualization (see Table 1). In the following Section 3, a new method that aims at fulfilling all these requirements is presented.

3. Method

In this article, we use a similar graph definition as employed in [13], where a graph G is an ordered tuple (V, E), $V \subset \mathbb{R}^2$ or $\mathbb{R}^3$ is a finite set of vertices and $E \subseteq V \times V$ is a finite set of edges. An edge e is an ordered tuple (u, v), with $u, v \in V$. For a given edge $e = (u, v)$, we define $\vec{e} = v - u$. Furthermore, let be $n = |E|$.

	directed graphs	3D	speed rel. to CUBu	number & complexity of parameters	edges are traceable
FDEB [14]	✓with [10]		10,000	high	✓
KDEEB [16]			100	moderate	
ADEB [24]	✓		100	moderate	
CUBu [21]	✓		1	moderate	
3DHEB [23]	✓		100	moderate	
Böttger et al. [1]		✓	100*	moderate	
3dFDEB [13]	✓	✓	100*	very high	✓
VBEB	✓	✓	100	moderate	✓

Table 1: Summary of how relevant related work and the proposed method (VBEB) do score concerning the named requirements: Are they able to process and visualize directed and 3D graphs? Then, we normalized and rounded the runtime of all approaches to the fastest (CUBu), regarding the rough computation times [21] of the standard US migration graph (see Figure 6). The result shows magnitudes. The runtime of the two algorithms marked with an asterisk was not measured on this graph and is just a sympathetic estimate based on their similarity to other methods. Finally, the number and complexity of parameters are rated and stated if traceability is given.

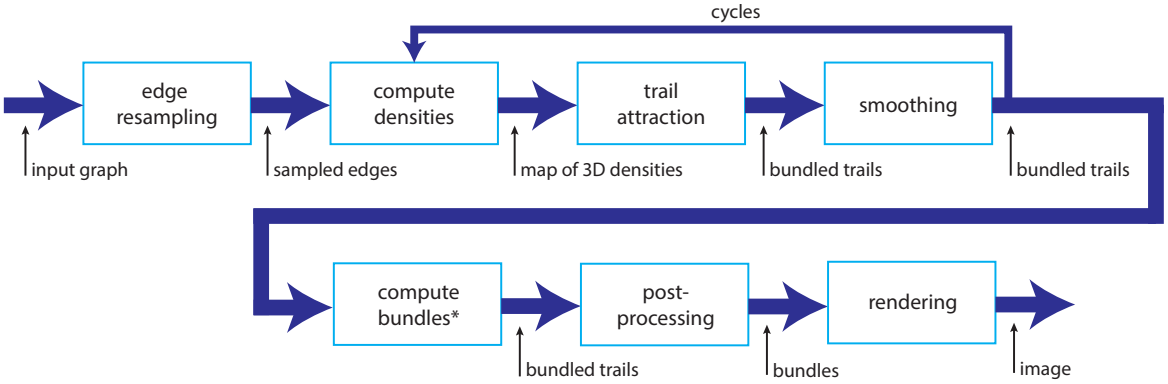

Figure 2: Voxel-based edge bundling pipeline, starting with the resampling of a graph drawing to voxel space, then computing a directional density map, attracting and smoothing the resampled edges along this field, then optionally compute explicit bundles (therefore marked with an asterisk), and finally do some post-processing and rendering on the bundled graph drawing.

The proposed method, voxel-based edge bundling (VBEB), is based on KDEEB [16], which iteratively generates a map of edge densities by utilizing kernel smoothing and then moves edges along the highest density values together to bundles. Kernel estimation is used on a voxel grid in many other methods that are not graph related, e.g., to identify clusters of crime [28]. In the following, we want to give a broad overview of the pipeline that is used, as shown in Figure 2. In the first step, the drawing of the given graph is resampled to voxel space (w.l.o.g. we consider $V \in \mathbb{R}^3$ in the following (**dim**)). This is described in detail in Section 3.1.

Second, and different to KDEEB, a directional density map is calculated and used for the kernel smoothing, in addition to the obvious replacement of the 2D kernel to a 3D one (see Section 3.2). Third, the edge trails are moved along the gradients of locally calculated density maps considering their direction (see Section 3.3) in combination with the previously calculated global directional density map. The latter process is streamlined from the concepts introduced by ADEB [24] and 3D density histograms for criteria-driven edge bundling (3DHEB) [23]. Both also take the edge direction into account, but only sample the edge direction space and need additional iterations in the size of the number of samples, which is not the case for the method that we propose in the following.

Next, the result of the bundling is smoothed (see Section 3.4), and the process is reiterated from the second step on. Finally, explicit bundles can be calculated (see Section 3.5); the result is post-processed and drawn (see Section 3.6). At the end of this Section (see Section 3.7), we analyze the runtime behavior of the presented algorithm.

3.1. Resampling

Initially, the node-link diagram of a given graph is resampled to a voxel grid. In the following the *drawing* of an edge stays related to the graph, i.e., every edge e is mapped to an ordered tuple of edge segmentation points $T_e = (t_e^{(0)}, \ldots, t_e^{(k)})$, with $k \in \mathbb{N}$ and $t_e^{(i)} \in \mathbb{R}^3$. The size of each voxel is determined by the average edge length and the constant λ, which specifies into how many segmentation points $t_e^{(i)}$ an edge with average length should be split, such that

$$voxel\ width = \frac{1}{\lambda |E|} \sum_{e \in E} \|\vec{e}\|$$

Every tuple of segmentation points T_e *knows* about its originating edge as this enables access to the underlying data structure, e.g., to name the participating edges and more meta information when a bundle is selected in the drawing. The edge resampling step happens once during the whole procedure. There is no further resampling after the segmentation points got moved (see Section 3.3) and thus the edge segments potentially are no longer uniform in length. Whenever two segmentation points of the same edge are moved to the same voxel they get merged.

In the following, the normalized direction of an edge segmentation point $t_e^{(i)}$ be:

$$dir(t_e^{(i)}) = \frac{t_e^{(i+1)} - t_e^{(i-1)}}{\|t_e^{(i+1)} - t_e^{(i-1)}\|}$$

At the boundaries, the direction is not calculated by the line through the predecessor and successor, but the segmentation point itself and its successor/predecessor.

3.2. Directional Density Map

In the next step, a density $\phi(x)$ is calculated using kernel density estimation [29] for each voxel x, analog to KDEEB with $x \in \mathbb{R}^2$ [16] and $x \in \mathbb{R}^3$ presented by Böttger et al. [3]. Given the kernel bandwidth $h > 0$, this computes the kernel function $K : \mathbb{R}^3 \to \mathbb{R}$. But contrary to previous approaches, the density is not calculated as a scalar sum of all interfering edges, $\rho : \mathbb{R}^3 \to \mathbb{R}$, but instead considering the direction of the corresponding edge segmentation points present in this voxel, such that $\phi : \mathbb{R}^3 \to \mathbb{R}^3$:

$$\phi(x) = \sum_{e \in E} \sum_{y \in T_e} \left(K(\frac{x - y}{h}) \cdot dir(y) \right)$$

This means that an edge or its segments only add to the density in the direction they run to in this point, which results in a map of directional densities (see Figure 3 for a 2D example). Differently said, we define a density map for each dimension, and edge segments are proportionally adding density to the dimensions they touch. We think the term density still is the right one to use, such as in related work, however this requires to accept that edge vectors pointing to a negative direction, add negative density and thus also a voxel's density vector can hold negative elements and finally, the scalar density in a voxel for a given direction (see Section 3.3) can be negative, as well. For example, a negative density in the x-axis of a voxel does mean that more edges are pointing to $-x$ than x; thus it becomes a hill for all edges pointing to negative x and a valley below sea level for edges pointing to positive x.

In this way, e.g., perpendicular edges do not artificially create hills in the density map at their crossing points (c.f. Figure 3), neither do anti-parallel edges. Otherwise, these points would especially attract edge paths in the next steps, which we want and do avoid with the usage of the directional density map (c.f. Figure 4). In case of an undirected graph, we decide among the two possible directions (u, v) and (v, u), such that all edge vectors lie within the positive Z hemisphere.

The result of the current step is a directional vector field of densities, which is used in the next step to reroute the edge trails. For the kernel function K, various kernels are applicable [29], but for us, a Gaussian kernel showed the best results.

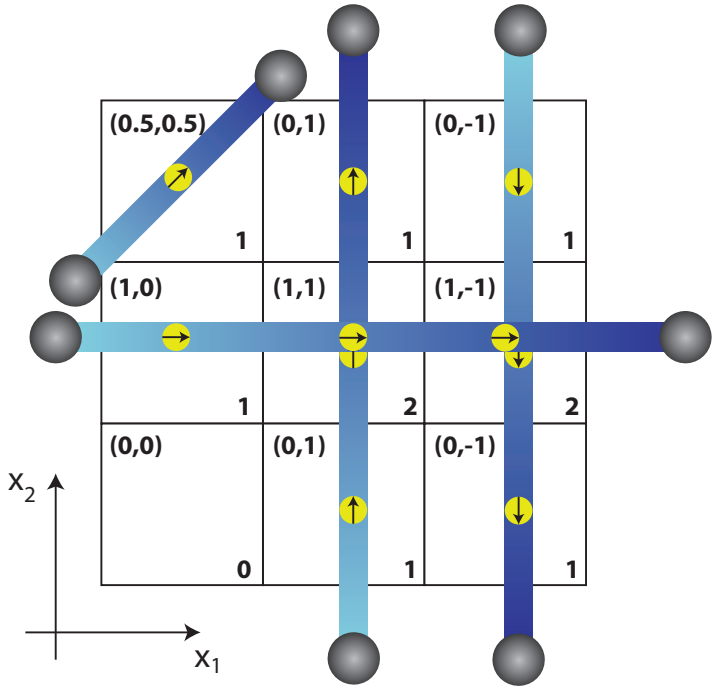

Figure 3: 2D example of a pixel-wise directional density map that we are calculating, see the vectors in the upper left corner of each box (pixel), and the density map computed by the related work, see the values in the lower right corner of each box (pixel), with an identity function as kernel for the given directed edges. The edges are running from light blue to dark blue, while the edge segmentation points are depicted as yellow circles with their direction marked with an arrow.

3.3. Attract the Trails

In the next step of the pipeline (Figure 2), the edge segmentation points are moved i times along the density gradient to adjacent voxels. When a point has reached a peak, it is not further moved. A voxel is treated as adjacent when it shares at least one edge with the corresponding voxel. However, as we are not using a scalar density field any longer, there is also no longer a gradient of density. But given the current edge segmentation point, it is possible to calculate a scalar map of densities with respect to the point's direction, in the following called *local* density map, based on the *globally* defined directional density map (see Section 3.2). Therefore, the vector dot product between the given segmentation point's direction $dir(t_e^{(i)})$ and the directional density $\phi(x)$ is calculated.

$$\rho_{local}(x, t_e^{(i)}) = \langle dir(t_e^{(i)}), \phi(x) \rangle$$

This happens for each adjacent voxel x and thus results in a local and scalar density map, such as in related work, however, this map is only valid for one direction, which is the one of the current segmentation point $t_e^{(i)}$. Then, the segmentation point is moved along the steepest gradient that is greater than zero, when there is one. The globally calculated directional density map and the direction-aware reduction to a local scalar density field by the vector dot product now finally ensures that segmentation points (edges) are only moved to other segmentation points with a similar direction and thus not moved to voxels with a lot of perpendicular or almost perpendicular directed edges, as well as to the opposite direction of anti-parallel edges (cf. Figure 4). It would also be possible to leave out the initial generation of a global directional density map and instead calculate a direction-aware scalar density field for each segmentation point from scratch when it is needed, but this is not efficient, as in worst case for every edge segmentation point all other edge segmentation points have to be visited. Then, note that the application of the proposed directional density map brings the same benefits to a 2D bundling, but the case of perpendicular edges is not interesting, as there is no direction in

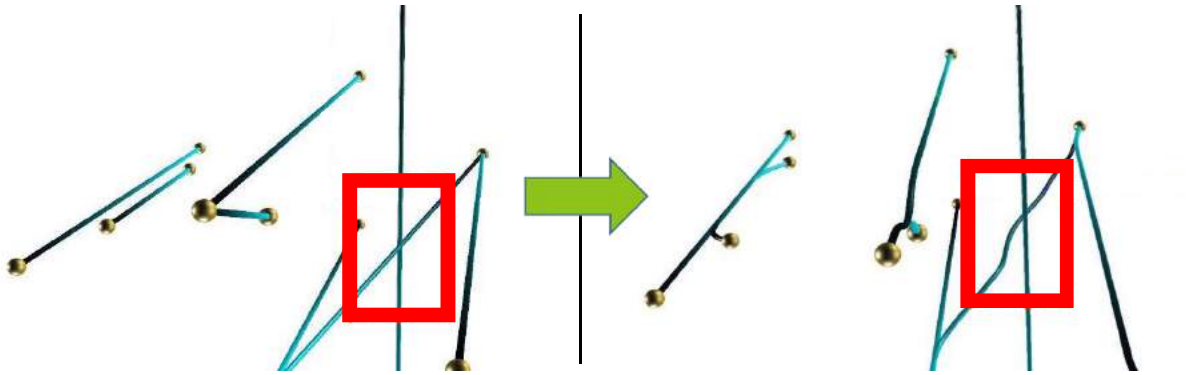

Figure 4: Perpendicular and anti-parallel edge trails are implicitly pushed away rather than attracted. The illustration shows a graph drawing of a 3D directed graph before the trail attraction (left) and the right after it (right). The evading edge is attracted by its left neighbor that is going in the same direction and not by the point of crossing with the perpendicular edge, which would have a higher density in other approaches.

which an edge could move to avoid crossing a perpendicular one. However, a perpendicular edge is still not artificially pushed to the crossing point and instead only influenced by edges that take a similar direction. Furthermore, the mentioned effects can be optionally strengthened by introducing hard thresholds. After attracting the edge segmentation points along the gradients, the edge trails are smoothed (see Section 3.4).

3.4. Smoothing

In the last step, the segmentation points of each edge were moved along the density field without considering that an edge should smoothly merge into a bundle, i.e., points at the start and end of an edge are potentially moved as far as the ones in the middle. Additionally, other local effects can lead to sharp bends. Therefore, the edge trails are smoothed by repositioning every segmentation point $t_e^{(i)}$, such that

$$t_e'^{(i)} = (t_e^{(i-1)} + t_e^{(i)} + t_e^{(i+1)})/3$$

This smoothing step is performed by most of the current edge bundling algorithms and guarantees, that edges smoothly fade in and out. It replaced the use of a force system [14] or a more complex procedure of moving segmentation points in the step before [3], which are aiming for the same. As a next step, the directional density map is updated (Section 3.2), the trails attracted (Section 3.3), and the current step repeated. This, so-called cycle is performed c times. As result, there is a bundling of the graph drawing and all trails are traceable to their edges and vice versa. The graph can be drawn after any cycle (see Figure 5), in between the cycles or parallel to the calculation of the next. This is an advantage when the graph is very large and the calculations take seconds or more, the user can already inspect the first intermediate results. However, the bundling is just implicit and it often might be beneficial to calculate explicit bundles. Whenever this is necessary, an optional bundle creation can immediately be attached to the pipeline (see Section 3.5). Otherwise, the post-processing is performed directly after all cycles were performed (see Section 3.6).

3.5. Explicit Bundles

At this point, the present data representation of the graph drawing is in all important points identical with 3dFDEB [13], and the method used there to create explicit bundles can also be used here. As 3dFDEB is not grid-based, the first step of the procedure is to cluster all edge segmentation points and receive points that are aligned to an implicit grid afterward. However, as all edge points are already implicitly aligned to the underlying voxel grid in our case, this step can be skipped. Therefore, we directly start with the second step, i.e., collect identical segments, which each consists out of two consecutive edge segmentation points, in different edge trails and join them to bundles. These segments are collected by a clustering method, e.g.,

DBSCAN [30], in a 6-dimensional space (3D starting point, 3D endpoint). Having access to explicit bundles allows not only different rendering styles (cf. [13]), e.g., a bundle thickness related to the accumulated weight of the containing edges, but also allows computational analysis on the achieved graph simplification (**trace**).

3.6. Post-Processing

Finally, the post-processing is performed in the same way as in Section 3.4, i.e., the trails and bundles are processed by a smoothing step, but now without the underlying voxel grid, resampled using Catmull-Rom splines [26], and finally rendered.

3.7. Runtime

The overall runtime complexity is not different from KDEEB [16]. Thus, each proposed step is linearly computable in the number of edges n, and therefore, the overall runtime is in $O(n)$ (**scale**). Some of the steps can even be computed more efficiently if iterating the underlying voxel grid instead, e.g., to update the directional density map, especially when the edge density is very high. However, the number of cycles c and the resolution of the voxel grid, which defines the resolution of the trails, are adding potentially significant factors to this runtime. Furthermore, the kernel size could be varied with the number of iterations as discussed in [21] to further accelerate the computations but was kept constant here (see Section 4).

4. Results

For evaluation, the presented method was implemented in C++, with naïve Intel TBB-based parallelization. One of the goals of the proposed technique was to minimize the number of parameters and make the remaining parameters as easy to understand and define for a user as possible (**dof**). While the presented method does not use radically fewer parameters, it is comparable to other kernel-based methods; we at least tried to find constant mappings to some of those through experimentation. Those mappings were then assigned to all example graphs in the following to show that they might work in general. Firstly, the voxel resolution (cf. Section 3.1) is defined by the average edge length and a constant λ that was set to 20. The effect of varying λ and thus the voxel resolution is depicted in Figure 7. Secondly, the kernel size was always set to 3, which seems very low in comparison to image-based methods, but simultaneously a voxel holds way more than 1 pixel here. Next, a segmentation point is moved i-times along its gradient (cf. Section 3.3) with i set to 4. Especially, the specification of the latter two is discussed in Section 5.2. Finally, the number of cycles c was set to 3. Because the observations we made is that after 3 or 4 cycles there are usually no more significant changes in the visualization (see Figure 5 for an example).

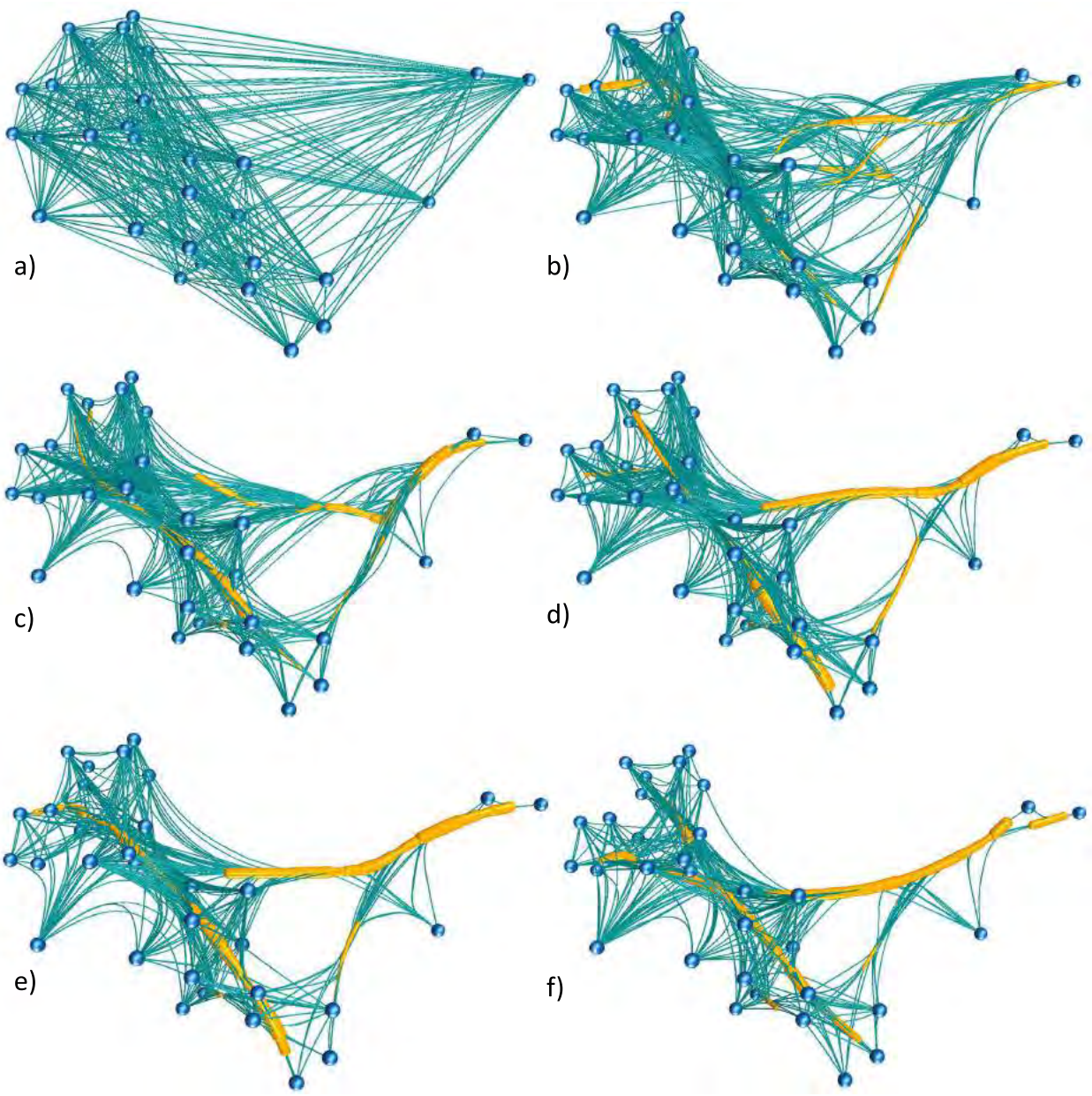

Figure 5: Connection between the resulting graph drawing and the number of cycles processed by VBEB starting before the first cycle a), to the fifth, f). The 3D graph is taken from [13] and represents brain region connectivities. It is treated as undirected for this example. Some bundles are detected and drawn in orange.

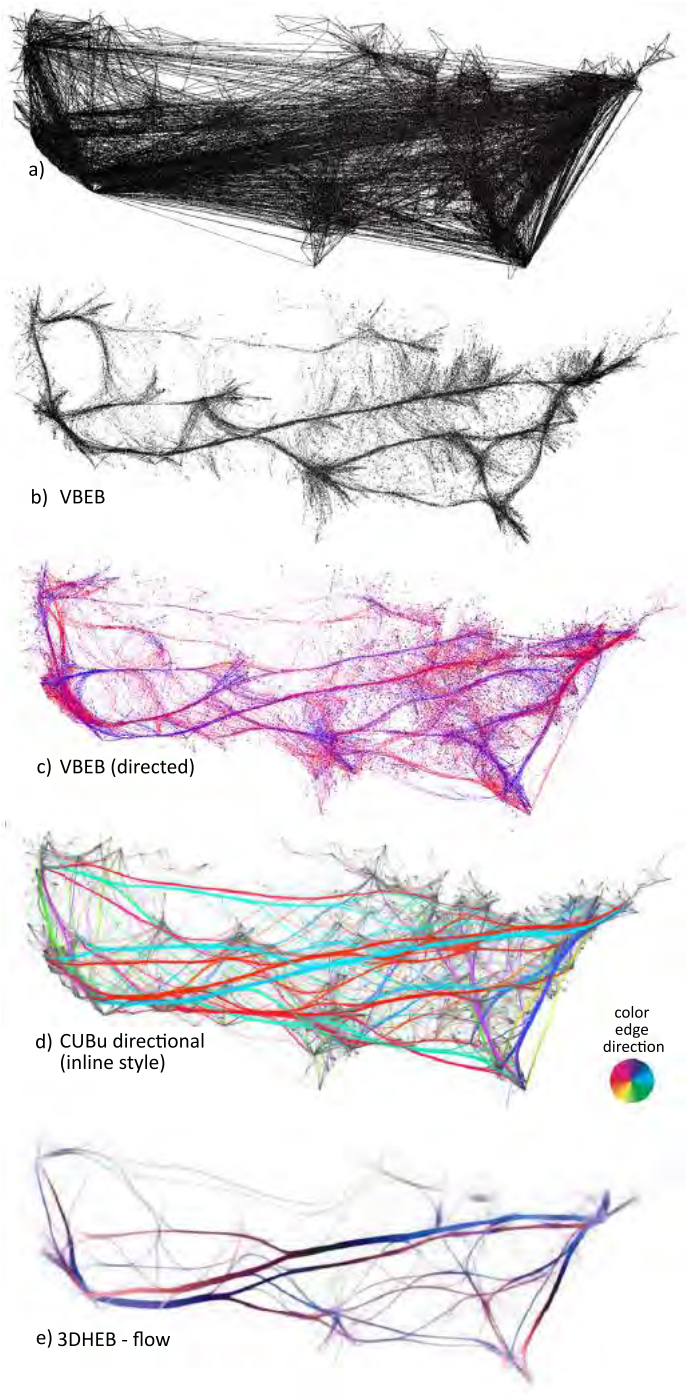

Figure 6: US migration graph (6,517 vertices, 9,780 edges, 2D, directed), a) shows the original, unbundled node-link diagram, b) shows the graph drawing bundled by VBEB when treated as undirected, and c) shows a bundling with directed edges (heading from red to blue). c) and d) are showing results from related work using the same data set where d) shows a directional bundling using CUBu [21], and e) shows a *flow* bundling using 3DHEB [23].

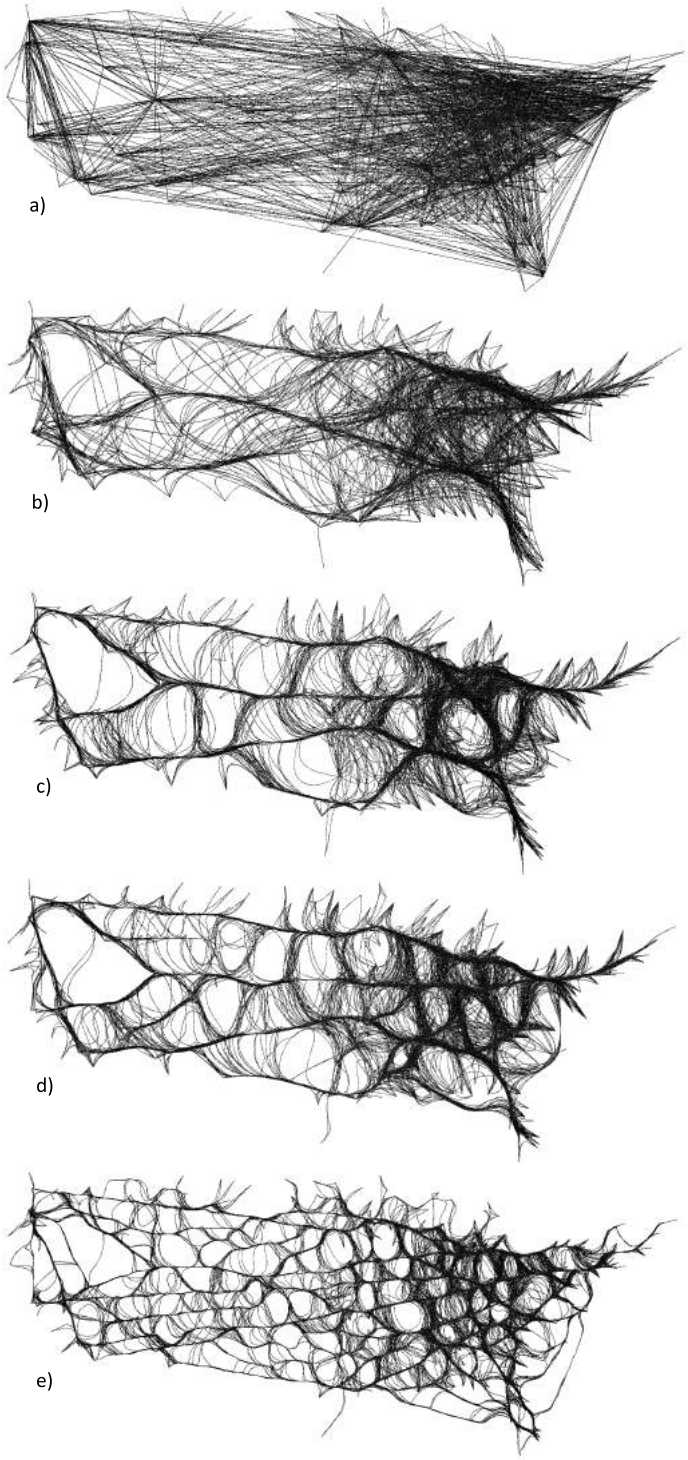

Figure 7: Us airlines graph (235 vertices, 2,101 edges, 2D, undirected) bundled by VBEB with different voxel (cell) resolutions, where a) shows the original, unbundled node-link diagram, b) the graph bundled with 3,772 cells and a total runtime of about $100ms$, c) 14,720 cells, which is the default the algorithms would have calculated ($\sim 700ms$) d) 23,028 cells ($\sim 800ms$) and e) with 59,248 cells ($\sim 5,800ms$)

Table 2: Mean runtime in *ms* and standard deviations obtained from 10 measurements for the different graphs. An x marks when the graph is directed or 3D. If so, the algorithm took that into account. The calculation of explicit bundles is optional and influences the runtime of the post-processing, thus both is given at the outer right of the second table.

graph	vert.	edges	voxels	3D	dir.	pre-processing	core algorithm sum
NEST	32	564	25K	x	x	4.7 (3.9)	69.4 (3.1)
US airlines	235	2,101	15K			10.6 (2.5)	295.9 (21.5)
funcBrain	437	8,031	72K	x		27.2 (6.5)	674.0 (18.3)
US migration	6,517	9,780	8K		x	26.5 (9.2)	473.7 (6.8)
funcBrain full	437	95,266	51K	x		288.1 (40.4)	10,043.6 (22.1)

graph	core algorithm			post	explicit bundles	
	cycle 1	cycle 2	cycle 3	processing	compute	post-prosessing
NEST	22.9 (1.0)	21.7 (1.5)	20.5 (1.2)	9.0 (0.4)	62.3 (2.9)	16.1 (1.0)
US airlines	105.7 (11.1)	95.5 (5.7)	90.4 (4.8)	33.0 (2.3)	361.0 (28.3)	84.9 (4.5)
funcBrain	250.6 (11.2)	214.1 (7.1)	204.7 (5.8)	78.4 (3.6)	426.2 (19.6)	151.7 (8.4)
US migration	172.3 (3.3)	151.3 (2.6)	145.6 (2.1)	61.6 (2.3)	454.3 (41.0)	137.5 (12.3)
funcBrain full	3,384.0 (15.1)	3,361.4 (10.5)	3,291.6 (16.7)	1,195.9 (32.4)	-	-

The algorithm was evaluated for different data sets by means of the resulting graph drawings. The first data set originated from a graph that shows the results of a neural network simulation using NEST, the interconnectivity between different brain regions of a macaque's brain [31]. The graph has a natural 3D embedding, is directed and its edges are weighted (representing the degree of connectivity). A depiction of the bundling is given in Figure 8. Second, two standard 2D data sets US migration[1] (see Figure 6) and US airlines (see Figure 7) were processed. For the later, the figure even shows the impact of different voxel grid resolutions, in comparison to the default one. Both are freely accessible[2]. Last, the drawing of a second graph with a natural 3D embedding depicting the functional connectivity of a human brain [3] was bundled and is shown in Figure 1. For the following runtime measurements, an artificially fully connected version of the latter graph was used additionally. The results of the runtime measurements for the different graphs are shown in Table 2. The measurements were performed on the CPU using an Intel Xeon E5-1650 3.50GHz, with 6 cores running Windows 7 with 16 GB of RAM. All measurements were repeated 10 times, and for all calculations and visualizations, the previously presented parameters were used. Additionally, the mode the algorithm processed the graph drawing in for the measurements is always related to its given properties, i.e., when marked as directed in the table the graph was processed as directed, even if it is also possible to ignore this property as shown in Figure 6.

5. Discussion

First, the visual outputs of the presented algorithm are discussed in Section 5.1. Then, the requirements which we have defined within the introduction are revisited with respect to the proposed approach in Section 5.2. This is followed by a summary of the limitations regarding the visual scaling of the proposed approach, as well as a review of possible starting points for future improvement in Section 5.3.

5.1. Visualizations

The proposed method primarily targets 3D graph drawings, and hence, we first demonstrate two examples from this domain. The first dataset is a graph representing the functional connectivity of the brain presented by Böttger et al. [3]. The visual outputs of VBEB are comparable to the original publication, yet the original runtime was 8s and more, where VBEB takes about 0.7s on a comparable machine. Thus, the performance is faster by more than one order of magnitude. As the two data sets are similar but not identical, although equivalent in size, a side-by-side comparison is not feasible here. The bundling of the fMRI-based (functional magnetic resonance imaging) connectivity might, for instance, help to illustrate statistical differences between brain scans of different subjects or at different times, in case of degeneration or regeneration [1].

The second 3D dataset is generated by a NEST simulation and is seen in Figure 8. Our result is yet again comparable to the one proposed by [13], but is subject to more variation than the first example. The reference

[1] `https://www.census.gov/population/www/cen2000/ctytoctyflow/`

[2] `https://github.com/upphiminn/d3.ForceBundle/tree/master/example/bundling_data`

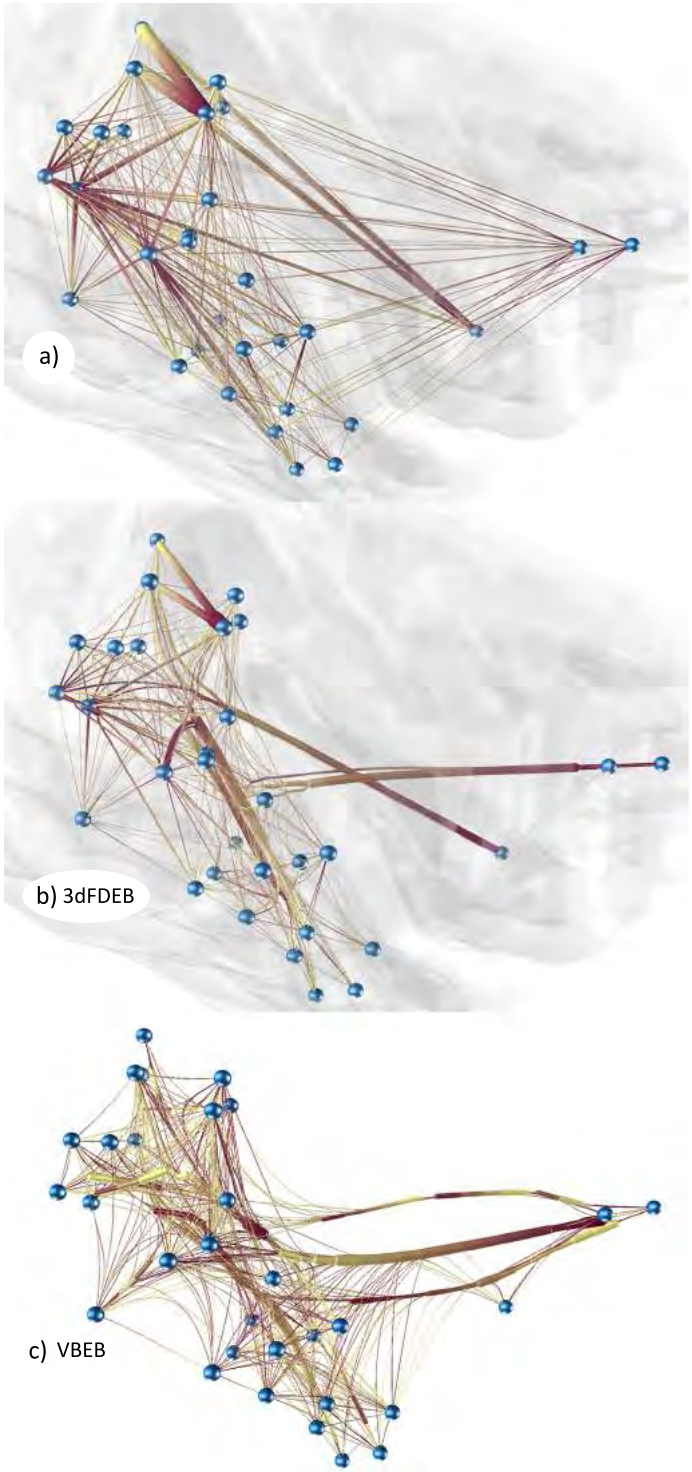

Figure 8: Graph originating from a NEST simulation (32 vertices, 564 edges, 3D, directed), a) shows the original, unbundled node-link diagram, b) shows a bundling from [13], and c) a bundling from the proposed VBEB, both with explicit bundles carrying summed edge weights. The edge direction is encoded in color from light to dark.

result is bundled *stronger* in several locations. While this contributes to the reduction of visual clutter, it also might potentially hide features of interest. In this point, it is important to remember what edge bundling is made for. It is not about getting the least cluttered image of a graph possible, it is about supporting an analyst in getting hidden insights, and therefore, the reduction of clutter might help. For the shown results 3dFDEB took about 1.36s, while VBEB takes about 0.07s on a slightly faster machine (approximately 15% faster). Thus, the performance of VBEB is again faster by at least one order of magnitude than compared to 3dFDEB and assuming that there was picked a good density value for the latter. It is worth noting that every 3D representation of a bundled graph appears significantly more cluttered when projected as a 2D image like here, compared to its natural appearance in a CAD-like application or an immersive virtual environment. This applies to both the reference results as well as ours.

Since the majority of existing edge bundling methods operate on the 2D domain, we also applied our approach to two standard examples within this domain to provide a fair comparison. In Figure 6, our results are directly compared to a selection out of the related work [32, 14, 20, 33, 34, 16, 22, 35], namely CUBu [21] and 3DHEB [23], as both provide state of the art edge-direction-aware bundlings in 2D. First to note is that in the 2D cases, the reference methods are much faster, which is discussed in more detail in Section 5.2 **(scale)**. However, in all depictions the same *structures* in the migration map get visible. The directed result in Figure 6c also shows that anti-parallel edges are placed in bundles side by side. This happens implicitly as a result of how the algorithm treats density. The offset between the anti-parallel bundles is not directly adjustable by a parameter; however, it is still a consequence of the used kernel size and voxel resolution, as the minimal offset corresponds to the distance between the centers of neighbored voxels.

Furthermore, the depictions show that the results of VBEB leave more small scale structures unbundled, which leaves more clutter. To some extent, the reason for this is that the bundling happens direction aware and therefore, not every nearby edge is attracted by a bundle. As discussed before this can have disadvantages or advantages depending on what is more important, less clutter, or the preservation of information (c.f. ambiguity free edge bundling [15]). But additionally, this effect gets stronger as lower the grid resolution is (see Figure 7). Otherwise, when considering higher resolutions, not only the runtime increases but also structures in the global scale vanish. This is a challenge many grid-based methods have to tackle to some extent, and we will discuss some possible solutions in the future work Section 5.3. However, considering an overview like the depicted ones, the user might be only interested in *global* structures anyways.

5.2. Requirements

In the introduction, we identified various requirements that an edge bundling algorithm should meet. Those are revisited in the following. The presented algorithm bundles the given graph drawing in the space it is given and we showed that for different 2D and 3D examples (**dim**), while the actual implementation is optimized for 3D drawings, i.e., all 2D examples are processed on a voxel grid as well, even when that would be not necessary and could be further optimized. Furthermore, the method separates edges that take different directions (**dir**) and thus places anti-parallel edges or trails next to each other (cf. [10] and see Figure 8 and 6), rather than putting them upon each other.

The algorithm operates on the graph data structures itself (**trace**), i.e., all the trails drawn are mapped to the underlying representation of the edge in the graph's data structure. Thus, every explicit bundle is linked to the participating edges. This offers, for instance, in place exchangeability for the visual representation of edges and bundles without any new calculations. Furthermore, it is possible to use the simplified graph structure for analysis and to smoothly un-bundle the drawing.

We have shown in a theoretical analysis that the algorithm scales linearly with the input size of the graph (see Section 3.7, **scale**) and substantiated this with some data points (see Table 2), even if they vary slightly with the given dimensionality and voxel resolution, as our implementation does not always iterate over the edges, but the voxel grid (cf. Section 3.7). However, in comparison to image-based methods, such as CUBu [21], our implementation on the CPU is up to a magnitude slower, i.e., for very large graphs those methods can provide much longer interactive framerates. Nevertheless, as already indicated, this mostly relates to the different architectures rather than the method and whenever it is not suitable to use the GPU or in the case of 3D graph drawings the proposed method is an alternative. Memory consumption is an additional factor in grid-based approaches, especially in 3D, as the required memory rises cubic with the

edge resolution. However, in practice this does not render a problem as the resolution does not increase with the input size of the graph and for the highest resolutions used in all examples less than $200MB$ of RAM in total were used, while the memory consumption already was an actual issue in other methods [3]. In summary, all examples but the artificially grown one, are processed in less than one second by the proposed method. Thus, the presented method might be usable for interactive analysis or even for the visualization of bundled time varying-graphs.

The last requirement considered refers to the number of parameters and their understandability (**dof**). The presented algorithm still needs some parameters, an equal amount to all kernel density-based algorithms, which is thus still a limitation. However, we can suggest a set of default parameter values, which we showed for that the algorithm behaves well for very different graphs with. From the four parameters, the most interesting still to play with is the voxel resolution, as shown in Figure 7. Different resolutions potentially reveal phenomena on different scales in a graph. Furthermore, the number of cycles and iterations is strongly connected and in sum has only to ensure that the edges find their final trails on the *peaks* of the density map, where cycles are computationally costlier than iterations, but potentially also cause the finer results. It might also be possible not to specify the number of cycles and iterations and instead stop the computation as soon as there are no more significant changes. Together both parameters depend on the chosen voxel resolution, which is a possible explanation of why the parameter set worked well for all our examples. However, even with a different resolution, the results are satisfying (see Figure 7). The last parameter is the kernel size. As mentioned before we are working with a small kernel of size 3. We ran experiments with larger kernels but the differences were almost identical. However, different voxel resolutions might also require adaption to the kernel size for some graphs. In summary, there are still some parameters to define, but they are robust, especially in comparison with force-based methods, where a small change in one parameter can result in a mess.

5.3. Limitation & Future Work

The most significant limitation is that the used grid resolution profoundly influences the scale of the bundling, i.e., with a high resolution mainly local effects are visible and with a lower resolution global ones (see Figure 7). Achieving both at the same time is not possible with the current approach. One possibility to resolve this issue would be to bin edges by length and perform a bundling for every cluster and with different resolutions, but draw all in the same place. This comes very close to the histogram concepts from 3DHEB [23]. However, one may observe that the scale of the effects a user wants to explore strongly correlates with the level of zooming into the graph drawing. Thus, a lower resolution for an overview might be sufficient, and it would be interesting to investigate an automatic change of the resolution depending on the zooming level, known as level of detail approach. This might be even more interesting as with increasing zoom-level the graph cutout gets smaller, which allows for a constant number of total grid cells even when the resolution increases and, thus the total runtime would not increase.

6. Conclusion

In this article, we presented a 3D extension to kernel density estimation-based edge bundling. The extension adds awareness of the edges' direction when using kernel smoothing and thus, e.g., implicitly supports both directed and undirected graphs. We evaluated our method on different 2D and 3D graph drawings and compared the results to the ones of state of the art edge bundling approaches. The resulting visualizations are comparable to existing ones, while the runtime could be heavily improved for the 3D cases and is still at interactive framerates for the processed 2D examples, however, slower when compared to GPU-based 2D approaches. In future work, we want the address the named limitations, i.e., especially visual scalability and seamless adjustability through less and understandable control parameter.

Acknowledgments

The authors would like to acknowledge the support by the Helmholtz portfolio theme "Supercomputing and Modeling for the Human Brain", and the support by the Excellence Initiative of the German federal

and state governments, the Jülich Aachen Research Alliance – Center for Simulation and Data Science. This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 785907 (HBP SGA2). Furthermore, the authors would like to thank Joachim Böttger who provided the data set of the functional brain connectivity.

References

[1] J. Böttger, A. Schäfer, G. Lohmann, Three-Dimensional Mean-Shift Edge Bundling for the Visualization of Functional Connectivity in the Brain, IEEE TVCG 20 (3) (2014) 471–480.

[2] C. Nowke, M. Schmidt, S. J. V. Albada, J. M. Eppler, R. Bakker, M. Diesmann, B. Hentschel, T. Kuhlen, VisNEST – Interactive Analysis of Neural Activity Data, Proc. IEEE Symposium on Biological Data Visualization (2013) 65–72.

[3] J. Böttger, R. Schurade, E. Jakobsen, A. Schäfer, D. S. Margulies, Connexel Visualization: A Software Implementation of Glyphs and Edge-Bundling for Dense Connectivity Data Using BrainGL., Frontiers in Neuroscience 8 (2014) 15.

[4] J. J. G. Keiriz, L. Zhan, O. Ajilore, A. D. Leow, A. G. Forbes, Neurocave: A Web-Based Immersive Visualization Platform for Exploring Connectome Datasets, Network Neuroscience (2018) 1–18.

[5] X. D. Arsiwalla, R. Zucca, A. Betella, E. Martinez, D. Dalmazzo, P. Omedas, G. Deco, P. F. M. J. Verschure, Network Dynamics With BrainX3: A Large-Scale Simulation of the Human Brain Network With Real-Time Interaction, Frontiers in Neuroinformatics 9 (2015) 2.

[6] K. Lamberts, S. Porsche, B. Hentschel, T. Kuhlen, U. Englert, An Unusual Linker and an Unexpected Node: CaCl2 Dumbbells Linked by Proline to Form Square Lattice Networks, CrystEngComm 16 (2014) 3305–3311.

[7] S. Gebhardt, S. Pick, H. Voet, J. Utsch, T. Al Khawli, U. Eppelt, R. Reinhard, C. Büscher, B. Hentschel, T. W. Kuhlen, FlapAssist: How the Integration of VR and Visualization Tools Fosters the Factory Planning Process, Proc. IEEE Virtual Reality (2015) 181–182.

[8] C. Ware, P. Mitchell, Visualizing Graphs in Three Dimensions, ACM Transactions on Applied Perception 5 (1) (2008) 1–15.

[9] A. Lhuillier, C. Hurter, A. Telea, State of the Art in Edge and Trail Bundling Techniques, Computer Graphics Forum 36 (3) (2017) 619–645.

[10] D. Selassie, B. Heller, J. Heer, Divided Edge Bundling for Directional Network Data, IEEE TVCG 17 (12) (2011) 2354–63.

[11] D. Zielasko, B. Weyers, M. Bellgardt, S. Pick, A. Meißner, T. Vierjahn, T. W. Kuhlen, Remain Seated: Towards Fully-Immersive Desktop VR, Proc. of IEEE Virtual Reality Workshop on Everyday Virtual Reality (2017) 1–6.

[12] D. Zielasko, M. Bellgardt, A. Meißner, M. Haghgoo, B. Hentschel, B. Weyers, T. W. Kuhlen, buenoSDIAs: Supporting Desktop Immersive Analytics While Actively Preventing Cybersickness, Proc. of IEEE VIS Workshop on Immersive Analytics.

[13] D. Zielasko, B. Weyers, B. Hentschel, T. W. Kuhlen, Interactive 3D Force-Directed Edge Bundling, Computer Graphics Forum 35 (3) (2016) 51–60.

[14] D. Holten, J. J. van Wijk, Force-Directed Edge Bundling for Graph Visualization, Computer Graphics Forum 28 (3) (2009) 983–990.

[15] S.-J. Luo, C.-L. Liu, B.-Y. Chen, K.-L. Ma, Ambiguity-Free Edge-Bundling for Interactive Graph Visualization, IEEE TVCG 18 (5) (2012) 810–21.

[16] C. Hurter, O. Ersoy, A. Telea, Graph Bundling by Kernel Density Estimation, Computer Graphics Forum 31 (3pt1) (2012) 865–874.

[17] D. Holten, Hierarchical Edge Bundles: Visualization of Adjacency Relations in Hierarchical Data., IEEE TVCG 12 (5) (2006) 741–8.

[18] P. Caserta, O. Zendra, D. Bodénes, 3D Hierarchical Edge Bundles to Visualize Relations in a Software City Metaphor, Proc. IEEE International Workshop on Visualizing Software for Understanding and Analysis (2011) 1–8.

[19] K. Buchin, B. Speckmann, K. Verbeek, Angle-Restricted Steiner Arborescences for Flow Map Layout, Algorithmica 72 (2) (2015) 656–685.

[20] A. Lambert, R. Bourqui, D. Auber, Winding Roads: Routing Edges into Bundles, Computer Graphics Forum 29 (3) (2010) 853–862.

[21] M. van der Zwan, V. Codreanu, A. Telea, CUBu: Universal Real-Time Bundling for Large Graphs, IEEE TVCG 22 (12) (2016) 2550–2563.

[22] A. Lhuillier, C. Hurter, A. Telea, FFTEB: Edge Bundling of Huge Graphs by the Fast Fourier Transform, Proc. IEEE Pacific Visualization Symposium (2017) 190–199.

[23] D. C. Moura, 3D Density Histograms for Criteria-driven Edge Bundling, arXiv preprint arXiv:1504.02687.

[24] V. Peysakhovich, C. Hurter, A. Telea, Attribute-Driven Edge Bundling for General Graphs with Applications in Trail Analysis, Proc. IEEE Pacific Visualization Symposium (2015) 39–46.

[25] M. Giereth, H. Bosch, T. Ertl, A 3d Treemap Approach for Analyzing the Classificatory Distribution in Patent Portfolios, Proc. IEEE Symposium on Visual Analytics Science and Technology (2008) 189–190.

[26] A. Lambert, R. Bourqui, D. Auber, 3D Edge Bundling for Geographical Data Visualization, Proc. International Conference Information Visualisation (2010) 329–335.

[27] C. Hurter, S. Puechmorel, F. Nicol, A. Telea, Functional Decomposition for Bundled Simplification of Trail Sets, IEEE TVCG 24 (99) (2017) 1–1.

[28] T. Nakaya, K. Yano, Visualising Crime Clusters in a Space-Time Cube: An Exploratory Data-Analysis Approach Using Space-Time Kernel Density Estimation and Scan Statistics, Transactions in GIS 14 (3) (2010) 223–239.

[29] B. W. Silverman, Density Estimation for Statistics and Data Analysis, Vol. 26, CRC Press, 1986.
[30] M. Ester, H. Kriegel, J. Sander, X. Xu, A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise, Proc. ACM Knowledge Discovery and Data Mining (1996) 226–231.
[31] M. Gewaltig, M. Diesmann, NEST (NEural Simulation Tool), Scholarpedia 2 (4) (2007) 1430.
[32] W. Cui, H. Zhou, H. Qu, P. C. Wong, X. Li, Geometry-Based Edge Clustering for Graph Visualization, IEEE TVCG 14 (6) (2008) 1277–84.
[33] E. R. Gansner, Y. Hu, S. North, C. Scheidegger, Multilevel Agglomerative Edge Bundling for Visualizing Large Graphs, Proc. IEEE Pacific Visualization Symposium (2011) 187–194.
[34] O. Ersoy, C. Hurter, F. V. Paulovich, G. Cantareiro, A. Telea, Skeleton-Based Edge Bundling for Graph Visualization, IEEE TVCG 17 (12) (2011) 2364–73.
[35] A. Graser, J. Schmidt, F. Roth, N. Brändle, Untangling Origin-Destination Flows in Geographic Information Systems, Information Visualization.