Smooth Bundling of Large Streaming and Sequence Graphs

C. Hurter [*]
ENAC/University of Toulouse, France

O. Ersoy [†]
University of Groningen, the Netherlands

A. Telea [‡]
University of Groningen, the Netherlands

Abstract

Dynamic graphs are increasingly pervasive in modern information systems. However, understanding how a graph changes in time is difficult. We present here two techniques for simplified visualization of dynamic graphs using edge bundles. The first technique uses a recent image-based graph bundling method to create smoothly changing bundles from streaming graphs. The second technique incorporates additional edge-correspondence data and is thereby suited to visualize discrete graph sequences. We illustrate our methods with examples from real-world large dynamic graph datasets.

Index Terms: I.3.3 [Picture/Image Generation]: Line and curve generation—

1 Introduction

Graph visualization supports various comprehension tasks such as understanding connectivity patterns, finding frequently-taken communication paths, and assessing the overall interaction structure in relational datasets [41]. Visualizing large graphs is challenging, due to inherent clutter, crossing, and overdraw problems [41, 8, 38]. Edge bundling methods have gained strong attention recently as a way to depict the overall connectivity pattern of large graphs by trading clutter for overdraw [25, 8, 38, 16, 22, 12, 18].

Dynamic graphs pose their own understanding challenges. The data volumes are far larger than for static graphs. Users are interested in spotting *changes* in the overall graph structure, while maintaining limited clutter. Bundling methods, a promising option to compactly depict dynamic graph changes, have however been mainly used for static graphs.

In this paper, we present two types of techniques for visualizing dynamic graphs using edge bundles. The first technique considers *streaming graphs*, *i.e.* temporally ordered, unstructured, edge-sequences with start and end lifetime moments. For this use-case, we extend a recent fast and clutter-free static-graph bundling method. The second technique considers *graph sequences*, *i.e.* a discrete set of graphs between which higher-level correspondences can be inferred. For this use-case, we exploit additional edge-correspondence information to further highlight events of interest such as the appearance, change, and disappearance of edge groups, and show results based on different underlying static bundling algorithms. We present efficient GPU implementations of both our techniques which scale to large dynamic graphs, ensure spatial and temporal continuity (*i.e.* preserve the user's mental map), and are simple to implement. We demonstrate our techniques on real-world dynamic graphs from the air-traffic and software engineering application domains.

The structure of this paper is as follows. Section 2 presents related work on dynamic graph and bundled graph visualization. Section 3 details our visualization method for streaming graphs. Section 4 presents our visualization method for graph sequences. Section 5 discusses the two methods in terms of desirable features. Section 6 concludes the paper.

[*]e-mail:christophe.hurter@enac.fr
[†]e-mail:o.ersoy@rug.nl
[‡]e-mail:a.c.telea@rug.nl

2 Related Work

2.1 Preliminaries

Dynamic graphs can be organized in two categories, as follows. *Streaming graphs* are defined as graphs $G = (V, E)$ on a vertex-set V and edge-set E, where edges

$$e \in E = \{n_{start}(e) \in V, n_{end}(e) \in V, t_{start} \in \mathbb{R}, t_{end} \in \mathbb{R}\} \quad (1)$$

are defined by their start and end nodes n_{start} and n_{end}, and lifetime $[t_{start}, t_{end} > t_{start}]$. A weak form of Eqn. 1 can be used to model streaming graphs where only an ordering of the t_{start} and t_{end} values is specified, rather than absolute values. Streaming graphs occur naturally in cases when an entire graph is not known in advance, *e.g.* events collected from live data sources [1].

Graph sequences are defined as ordered sets of graphs $G^i = (V^i, E^i)$ which typically capture snapshots of the structure of a system at N moments $1 \leq i \leq N$ in time. We further call a graph G^i in such a sequence a *keyframe*. In contrast to streams, edges are explicitly grouped in keyframes, and additional semantics can be associated with each such keyframe. Following this, sequences may contain so-called correspondences

$$c : E^i \rightarrow \{\{e_{corr} \in E^{i+1}\}, \varnothing\} \quad (2)$$

Here, $c(e \in E^i)$ yields an edge $e_{corr} \in E^{i+1}$ which logically corresponds to e (if such an edge exists), or the empty set (if no such edge exists). Correspondences model edge-pairs in consecutive keyframes that are related from an application perspective, *e.g.* caller-callee relations between the *same* function definitions in consecutive revisions of a software system.

2.2 Dynamic graph visualization

Visualizing changing graphs has a long history. Methods can be divided into two classes, as follows.

Unfolding the time dimension along a spatial one, *e.g.* using the "small multiples" approach [4], has led to many dynamic graph visualizations. In graph drawing, specific solutions are known for planar straight-line graphs [3]. In software visualization, TimelineTrees [5], TimeRadarTrees [6], and TimeArcTrees [37], and CodeFlows [37] lay out a graph along a 1D space, *e.g.* circle or line, and juxtapose several such instances on an orthogonal axis to show the graph evolution. Although reducing clutter by not using a node-link drawing metaphor, such methods are visually not highly scalable, nor are they very intuitive, especially for long time series containing complex event dynamics.

Producing an *animation* of the graph's evolution is a second way to understand dynamic graphs. Several techniques generate incremental node-link drawings that show the graph evolution by optimizing a cost function that combines classical static-graph-drawing aesthetic criteria with maximizing the layout stability of unchanging graph parts [15, 13, 14, 20]. Animation can be preferable to small-multiples in conveying dynamic patterns, especially for long repetitive time series [39]. Such methods, however, may suffer from visual clutter, due to the underlying node-link metaphor.

2.3 Bundled edge graph visualization

Edge bundling techniques trade clutter for overdraw by routing related edges along similar paths. Clutter causes and reduction strategies are discussed in [11]. Bundling can be seen as sharpening the edge spatial density, by making it high along bundles and low elsewhere [22]. Bundling improves readability for finding node-groups related to each other by edge-groups (bundles) which are separated by white space [16, 38], *i.e.* produces images where high-level graph structures should be easy to follow, while details (individual edges) are emphasized less.

Dickerson *et al.* merge edges by reducing non-planar graphs to planar ones [9]. Holten bundled edges in compound graphs by routing edges along the hierarchy layout using B-splines [18]. Gansner and Koren bundle edges in a circular node layout similar to [18] by area optimization metrics [17]. Dwyer *et al.* use curved edges in force-directed layouts to minimize crossings, which implicitly creates bundle-like shapes [10]. Force-directed edge bundling (FDEB) creates bundles by attracting edge control points [19], and was adapted to separate opposite-direction bundles [34]. The MINGLE method uses multilevel clustering to accelerate the bundling process [16]. Flow maps produce a binary clustering of nodes in a directed flow graph to route curved edges [30]. Control meshes are used to route curved edges, *e.g.* [31, 43], a Delaunay-based extension called geometric-based edge bundling (GBEB) [8], and 'winding roads' (WR) which use Voronoi diagrams for 2D and 3D layouts [25, 24]. Skeleton-based edge bundling (SBEB) uses the skeleton of the graph drawing's thresholded distance transform as bundling cues to create strongly ramified bundles [12].

To render bundles, edge color interpolation for edge directions [18, 8] and transparency or hue for edge density or for edge lengths [25, 12] are used. Bundles can be drawn as compact shapes whose structure is emphasized by shaded cushions [38, 33]. Graph splatting visualizes node-link diagrams as smooth scalar fields using color and/or height maps [40, 23]. To explore crowded areas (overlapping bundles), semantic lenses can be used [21].

2.4 The challenge of bundling dynamic graphs

Given the above, it seems appropriate to use edge bundling to visualize the (simplified) structure of dynamic graphs. Pioneering work in this area has been recently done by Nguyen *et al.*, who cut a streaming graph into a set of graphs using a sliding time-window, and visualize each such graph using existing edge-bundling methods [19, 18]. Edge similarity, or compatibility, is enhanced to take into account temporal coherence. Given a stable edge-bundling layout, this method can produce animations of bundled graphs with spatial and temporal continuity.

This approach can be improved in several directions: scalability (number of edges handled), ensuring a high spatio-temporal continuity of the produced animations where large-scale and long-life structures are stable over time and display space, and using the correspondence information present in graph sequences. We next present two edge bundling methods for streaming and sequence graphs which incorporate the above-mentioned improvements.

3 Visualizing streaming graphs

Given a graph G, which includes (2D) node positions, we can think of (2D) bundling as an operator $B : G \to \mathbb{R}^2$ which creates a drawing $B(G)$ which maps edges that are close in G to close spatial positions (bundles) [22]. Different bundling algorithms propose different ways to model edge closeness in G: tree-distance of edge end-nodes in a hierarchy [18], closeness of edges in a straight-line drawing of G [12, 19, 22], or the more general combination of graph-theoretic and image-space distances [28].

Consider now a streaming graph (Eqn. 1), the "instantaneous" graph $G(t) = \{e \in G | t \in [t_{start}(e), t_{end}(e)]\}$ and its bundling $B(t) = B(G(t))$ by some bundling operator B. Ideally, we want that $B(t)$ (a) varies continuously, or smoothly, in time with respect to the input $G(t)$ and also (b) keeps the spatial properties of the underlying bundling operator B, *i.e.*, puts close edges in tight bundles.

Property (b) is readily satisfied by using a "good" bundling algorithm B that guarantees that for any input graph, the result will be (strongly) bundled, such as *e.g.* [16, 25, 8, 12, 22], or to a lesser extent [18, 19], as we shall see. Property (a) means that, when $G(t)$ changes only slightly, then $B(G(t))$ should also change only slightly, so graph structures which are stable in time are also stable in the final visualization. Conversely, if there is an abrupt change in the graph, then there should be a visible change in the animation. However, even in the presence of such large changes in the input, discontinuous bundle *jumps* in the animation should be avoided, since visually tracking such jumps is hard.

A partial answer to (a) can be achieved by reducing the dynamics of $G(t)$, *e.g.* by applying a low-pass filter to $G(t)$. In other words, the bundling result shown at moment t is $B(\tilde{G}(t))$ where $\tilde{G}$ is the filtered graph. This is the solution proposed by StreamEB, who pioneered bundled layouts for streaming graphs [28]. They use a sliding window technique (finite-support box filter) to compute $\tilde{G}$ as all edges alive in $[t, t+\Delta t]$.

However, this approach has two limitations. First, the smoothness of the final animation depends strongly on the variation rate of $\tilde{G}$. If graphs for two consecutive time moments $\tilde{G}(t)$ and $\tilde{G}(t+\Delta t)$ differ too much, *e.g.* there are too many edges added or deleted per time unit, or the filtering time-window is too small, then there is no guarantee that the corresponding bundlings $B(\tilde{G}(t))$ and $B(\tilde{G}(t+\Delta t))$ are spatially close. If this is not the case, users notice a disruptive visual jump from t to $t+\Delta t$. Secondly, the computational efficiency of the approach in [28] strongly depends on the scalability of the underlying static bundling operator B. Algorithms which ensure good spatial stability, *e.g.* [19, 8] are also quite expensive, roughly $O(|\tilde{E}|^2)$ for $|\tilde{E}|$ edges in $\tilde{G}(t)$. Faster bundling algorithms [16, 12, 25] cannot ensure continuity, *i.e.*, a small change in the input graph may generate a large change in the bundled image, so are less suitable for stream bundling.

3.1 Algorithm

We address the above challenges by exploiting the properties of a recent bundling method for large graphs: kernel-density estimation edge bundling (KDEEB) [22]. Given a graph drawing $G = \{e_i\}_{1<i<N}$, KDEEB estimates the spatial edge density $\rho : \mathbb{R}^2 \to \mathbb{R}^+$

$$\rho(\mathbf{x}) = \sum_{i=1}^{N} \int_{\mathbf{y} \in e_i} K\left(\frac{\mathbf{x}-\mathbf{y}}{h}\right) \tag{3}$$

where $K : \mathbb{R}^2 \to \mathbb{R}^+$ is an Epanechnikov kernel of bandwidth h. KDEEB iteratively moves each point $\mathbf{x}$ of each edge upstream along $\nabla\rho$ following

$$\frac{d\mathbf{x}}{dt} = \frac{h(t)\nabla\rho(t)}{\max(\|\nabla\rho(t)\|, \varepsilon)} \tag{4}$$

where ε is a small normalization constant. After a few Euler iterations for solving Eqn. 4, during which one decreases h and recomputes ρ, edges converge into bundles. A final 1D Laplacian smoothing pass is done on edges to remove small wiggles. Full details are given in [22]. Upon a closer analysis (not reported by the KDEEB authors), one can see that this process is nothing else but performing the well-known mean-shift algorithm [7] on the drawn edges. In other words, the bundled graph is a *clustering* of the graph drawing based on edge similarity. This observation is important, since smoothness, noise robustness, and stability results proven for mean shift [7] can be readily extrapolated to KDEEB.

Core to KDEEB is the fast computation of the density map ρ. This is done by splatting the kernel K, encoded as an OpenGL texture, into an accumulation map. This allows bundling graphs of tens of thousands of edges in a few seconds on a modern GPU.

Our main idea for bundling streaming graphs is now to let the KDEEB iterations vary in sync with the stream time t. The principle is simple (see also Algorithm 1): We move a sliding window $[t, t+\Delta t]$ over the entire time range of the input streaming graph, compute $\rho(t)$ from the graph $\tilde{G}(t)$, and advect edges following Eqn. 4. There are two key advantages to this approach. First, $\rho(t)$ can be very efficiently computed by the underlying KDEEB algorithm, which is $O(|\tilde{E}|)$, *i.e.* proportional to the edge count in the current graph $\tilde{G}$. Secondly, and most importantly, the original KDEEB required $I = 5..10$ iterations for a single static graph to be bundled. We remove this iterative process by letting $\tilde{G}$ bundle *while* sliding the time-window. This makes sense since (a) if $\tilde{G}$ changes very slowly, advancing the stream time t is nearly equivalent to performing iterations for a fixed t, so we obtain a strongly bundled $\tilde{G}$, which is what we want to see. If (b) $\tilde{G}$ changes rapidly, then our process has less time to bundle, and thus we see looser bundles, which conveys us precisely the dynamics of $\tilde{G}$. More details on performance and parameter settings are given in Sec. 5.2.

```
1  $t \leftarrow 0$
2  while stream not ready do
3      $\rho \leftarrow 0$
4      $E_{live} \leftarrow \{e \in E | [t_{start}(e), t_{end}(e) \cap [t, t+\Delta t] \neq \varnothing\}$
5      foreach $e \in E_{live}$ do
6          splat $e$ into $\rho$ ;                        //Splat live edges (Eqn. 3)
7      end
8      foreach $e \in E | t_{end}(e) \in [t - \delta t, t]$ do
9          relax $e$ towards its original position ;   //Vanishing edges
10     end
11     foreach $e \in E_{live}$ do
12         advect $e$ one step ;                         //See Eqn. 4
13         apply 1D Laplacian smoothing on $e$ ;
14         draw $e$ in the visualization ;
15     end
16     $t \leftarrow t + \delta t$ ;                      //Advance sliding window
17 end
```

Algorithm 1: Bundling streaming graphs with KDEEB

Intuitively, our dynamic bundling method can be thought of as a process where edges continuously track the local density maxima of a dynamically-changing graph. Since at each advection step edges move with a bounded amount h (line 12, Alg. 1), and since advection is done while advancing the stream time t, the maximal amount an edge-point can move at any time is h (Eqn. 4). Hence, the bundles move smoothly on the screen.

For disappearing edges, we can perform an additional step: We interpolate these edges from their current (bundled) position towards their original (unbundled) position they had in the input stream (line 9, Alg. 1). This makes the animation symmetric: New edges progressively bundle as times goes by, while disappearing edges relax, or unbundle, towards their original positions after exiting the sliding time-window. To further emphasize this effect, we modulate the edges' transparencies in a similar fashion. We note that this effect is optional. If left out, disappearing edges will exit silently, without relaxation. The choice of using relaxation or not depends on whether users want to see edge-vanishing events or not.

An important goal of animated visualizations is to help users detect deviations from regular patterns [39, 41]. We support this by shading bundles to convey their speed of change, using a simple and fast image-based method: We compute the density moving-average $\tilde{\rho}(t)$ over $[t, t+\Delta t]$, and color bundles by the normalized difference $|\rho(t) - \tilde{\rho}(t)|/\tilde{\rho}(t)$ using a white-to-purple colormap. Results are shown next.

3.2 Applications

Figure 1 shows several frames from a streaming visualization of US flights [36] (6 days, 41K flights). The streaming graph contains flights with start and end date-and-time and geographical locations. The resulting flight-trail bundles are smooth, clutter-free, and exhibit a continuous variation in time[1]. From the stills, we see that same time-of-day flight patterns are quite similar for several days. However, they vary strongly over a day: During the evening, the East coast has the most intense traffic. During the afternoon, the entire US is quite uniformly covered with flight routes. During the night, flights linking the two coasts dominate.

Figure 2 shows a similar visualization for flights over France (7 days, 54K flight trails). Similar to the US dataset, bundles are smooth and clutter-free in both space and time. Colors indicate the bundles' speed of change (white=stable, purple=rapid changes, see Sec. 3.1). Red dots show the first and last positions when a plane was monitored. Dots inside France are actual airports. Dots outside the French territory indicate international flights which enter/exit the French airspace. We see that, during the day, the main "backbone" flight pattern is quite stable over different days, and contains mainly north-south routes, with Paris as a key hub (Fig. 2 top row). A different pattern, also quite stable, appears at night (Fig. 2 bottom row): A salient vertical bundle shows Southern flights bound to Paris. We also see more purple, which shows that night-time flight paths are much less stable than during-the-day flight paths.

For the US dataset, a qualitative comparison of our results with StreamEB [27] shows that KDEEB produces stronger bundles and an overall smoother animation. This is first due to the fact that KDEEB can produce bundles with many inflexion points, while FDEB has a smoothing factor built in its edge compatibility metric that disfavors such shapes. Secondly, this is due to the built-in smoothness of our method which bundles edges as they arrive in the input stream.

4 Visualizing Graph Sequences

Graph sequences G^i (Sec. 2.1) exhibit different properties from streaming graphs, as follows. First, streams allow defining an infinity of "instantaneous" graphs $G(t) = (V, \{e \in E | t_{start} < t < t_{end}\}), \forall t \in \mathbb{R}$ (see Sec. 3.1). Some of these graphs may not have a direct meaning or usefulness. In contrast, graph sequences contain a finite set of graphs which have been explicitly computed in specific ways, *e.g.* for particular time moments, *e.g.* (major) revisions of a software system. Secondly, keyframe correspondences add higher-level, edge-centric, information, *e.g.* the fact that two files f_1, f_2 share a common piece of text in version 1, and next f_1 shares the same text with a file f_3 in version 2. In contrast, streaming graphs (Eqn. 1) only specify how edges appear and disappear in time, but do not necessarily encode logical connections between edges at different time moments. Thirdly, graph sequences do not necessarily come with birth and death moments for individual edges. Finally, keyframes in graph sequences must be wholly available before processing, whereas edges in a streaming graph can be, in most cases, analyzed "online" as they appear. All in all, the above make a case for treating graph sequences differently from graph streams.

4.1 Algorithm

For graph sequences, we propose the following bundling method: For each keyframe G^i, we compute its bundled layout $B^i = B(G^i)$, using a given bundling algorithm B. Next, we interpolate these layouts between a keyframe i and the previous and next keyframes $i-1$ and $i+1$ respectively using the correspondence data (see Fig. 3). Consider a time axis t along which we place keyframes *e.g.* at moments $t_i = i\Delta t$ (any other definition of t_i can be easily used, if available). For each edge $e \in G^i$, if $c(e) = e^{i+1} \in E^{i+1}$, we linearly interpolate $B^i(e)$ to $B^{i+1}(e^{i+1})$ over the interval $[t_i, t_{i+1}]$ (Fig. 3D). If $c(e)$ is the empty set, *i.e.* e has no correspondence in E^{i+1}, we interpolate $B^i(e)$ to the line segment $L(e) = (n_{start}(e), n_{end}(e))$ over the same time interval (Fig. 3B). Symmetrically, if $c^{-1}(e) = e^{i-1}$, we interpolate from $B^{i-1}(e^{i-1})$ to $B^i(e)$ over $[t_{i-1}, t_i]$ (Fig. 3A), else we interpolate from $L(e)$ to $B^i(e)$ over the same time interval (Fig. 3C).

We emphasize appearing and disappearing edges by shading: Edges that have correspondences between two keyframes i and $i+1$ are blue and thick. Edges that disappear from i to $i+1$ get a color

[1]For this and the other examples next, see the submitted videos.

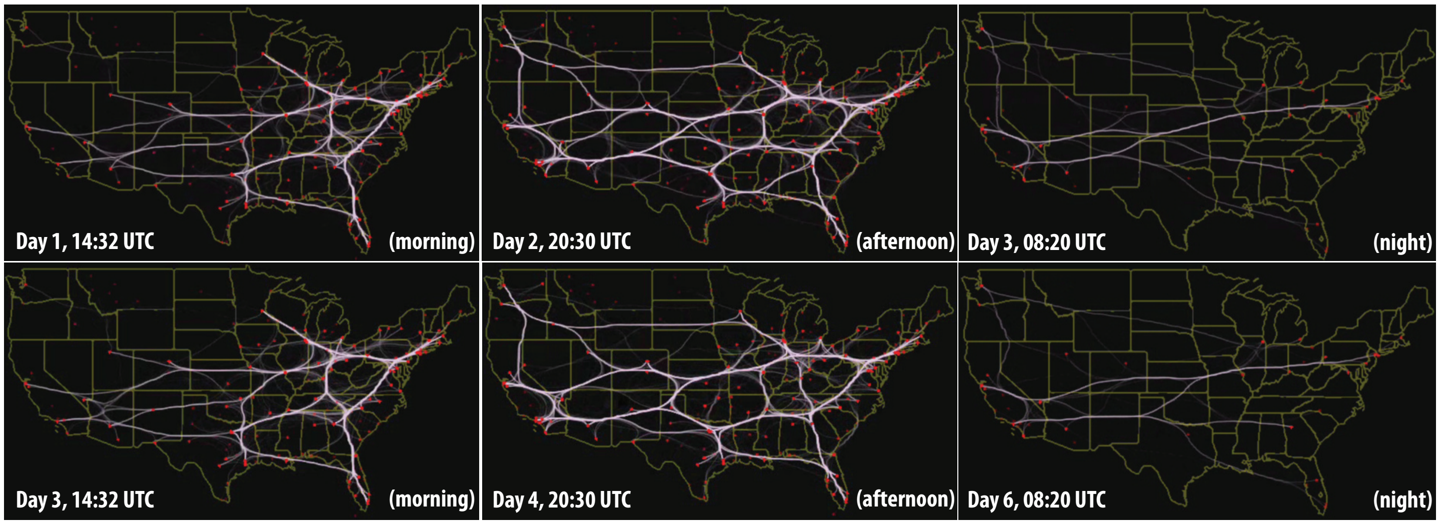


Figure 1: Streaming visualization for 6-days US airline flight dataset (Sec. 3.2)

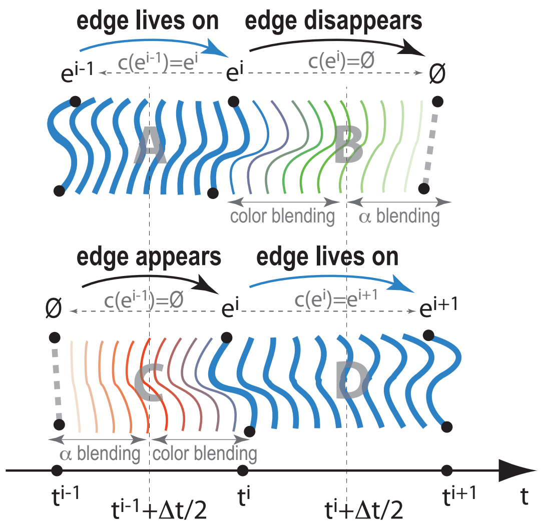


Figure 3: Interpolation for graph sequence visualization

linearly interpolated between blue (at t_i) and green (at $t_{mid} = \frac{t_i+t_{i+1}}{2}$), and next get the transparency (alpha) value decreasing from opaque (at t_{mid}) to fully transparent (at t_{i+1}). Edges that appear from $i-1$ to i get an alpha increasing from fully transparent (at t_{i-1}) to opaque (at t_{mid}), followed by a color interpolated from red (at t_{mid}) to blue (at t_i). Hence, edges appear by fading in to red (highlights their incoming), then smoothly merge in a blue bundle, and disappear by unbundling, becoming green (highlights their vanishing), and fading out. Examples next explain the color choice.

4.2 Applications

We illustrate our sequence-graph visualization with two datasets from software engineering. The first dataset contains 22 releases of Mozilla Firefox [26]. For each revision, we extracted the code hierarchy (folders and files), and also the so-called *clones*, or code duplicates, using the freely available clone detector SolidSDD [32, 35]. Hence, for each revision, we obtain a compound hierarchy-and-associations graph where two files are linked by an edge if they share a code duplicate. If a code fragment is cloned in several files, then all these files are pair-wise linked by associations.

Figure 4 shows snapshots from SolidSDD's HEB visualization for such graphs. Node colors show duplication amount (red=high, green=low). Seeing how subsystems share clones is useful in perfective maintenance, where one needs to plan code clone removal with minimal impact on system architecture. It is also important to assess how much, and where, did adaptive maintenance (*i.e.* adding new features) introduce new clones, and how much, and where, did perfective maintenance succeed to remove clones in the past [29]. For this, we need to easily compare the clone evolution patterns. This is hard to do using such small-multiple visualizations.

To support such a task, we proceed as follows. First, we create a so-called union hierarchy containing all graph nodes in the analyzed releases [2]. This contains 13856 file and folder nodes. Next, we build correspondences between clones in consecutive releases: Two clone relations e^i and e^{i+1} correspond if they link the same files, *i.e.* files having the same fully qualified names, in G^i and G^{i+1}. Other ways to find correspondences, *e.g.* using the actual text content of the clones [29], can be readily used too, if desired. The above steps deliver a graph sequence G^i in the sense described in Sec. 2.1. This sequence contains 5687 unique edges (that is, when counting corresponding edges as one) and 48591 edges in total.

We now use our graph-sequence visualization to analyze this sequence. Figure 5 shows several frames from this animation (see submitted videos). The bottom row shows results produced using KDEEB as underlying bundling method. Disappearing edges are green (removing clones is good); appearing edges are red (introducing new clones is bad). Additionally, we color hierarchy nodes as follows: Nodes which contain a changing clone count are colored by the clone count change, using red for positive values and green for negative values. Nodes where the clone count stays constant are colored blue. In all cases, we use saturation to indicate absolute values (saturated=high, desaturated=low values).

We note several events of interest. First, there is a relatively stable "core" clone-structure that lives for a long time (blue bundles). These can be hard to remove clones, or clones that maintainers were not aware of, *e.g.* if no clone detector was actively used on this system during perfective maintenance. We also spot several moments when major clone-pattern changes occur, *e.g.* from revision 2.0.0.10 to 3.0, many green edges appear, so many clones are removed (Fig. 5 d). Node coloring helps spotting high-clone-density subsystems. For instance, from revisions 3.6.10 on, we see two such dark-blue groups (dotted circles in Fig. 5 bottom row, e-h). Since these groups stay visible in several revision, they indicate "stubborn" clones which, for several reasons, could not be removed for a long time. Although this information is encoded in the bundles too, finding such patterns on

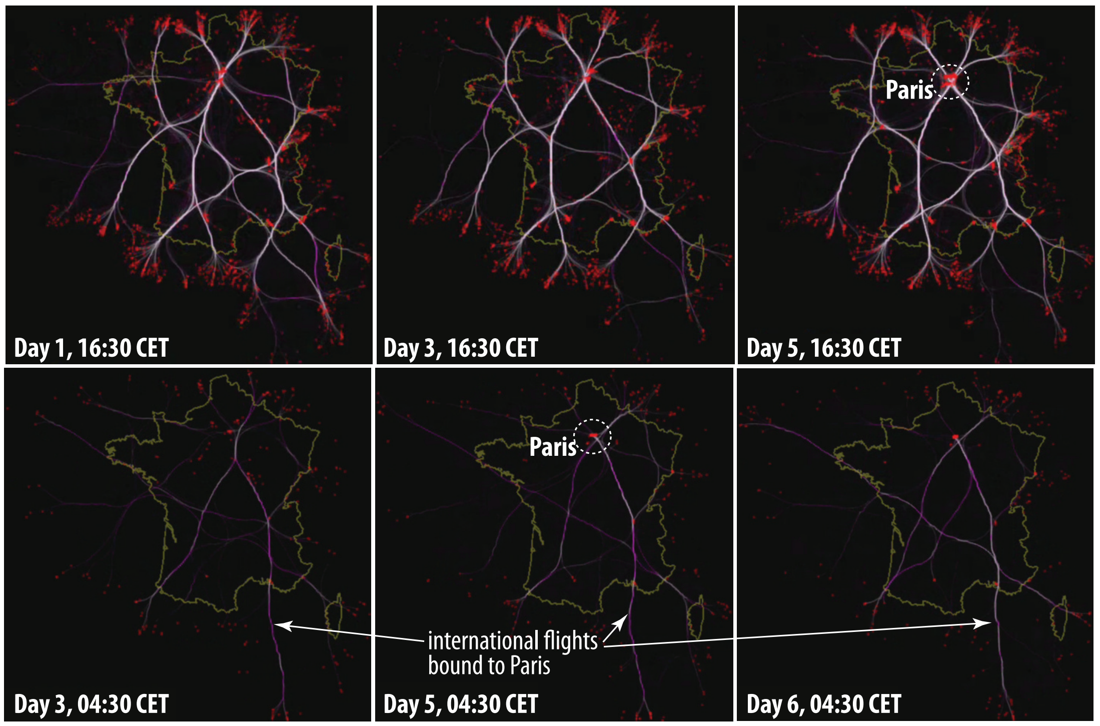


Figure 2: Streaming visualization for 7-days France airline flight dataset (Sec. 3.2)

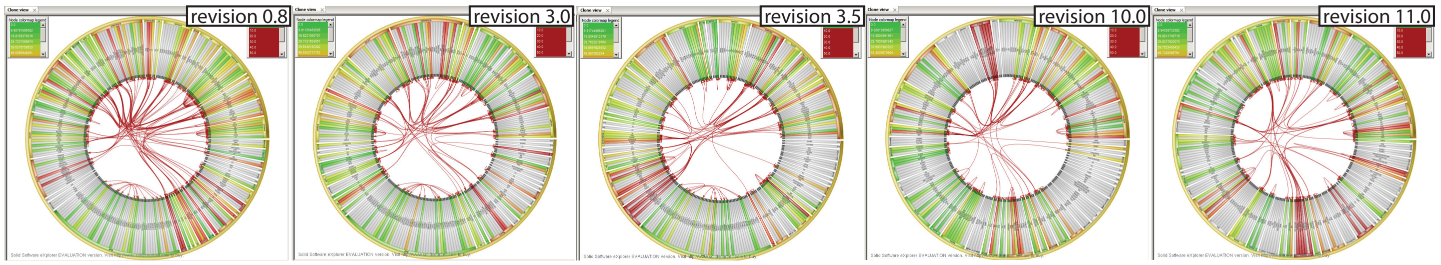
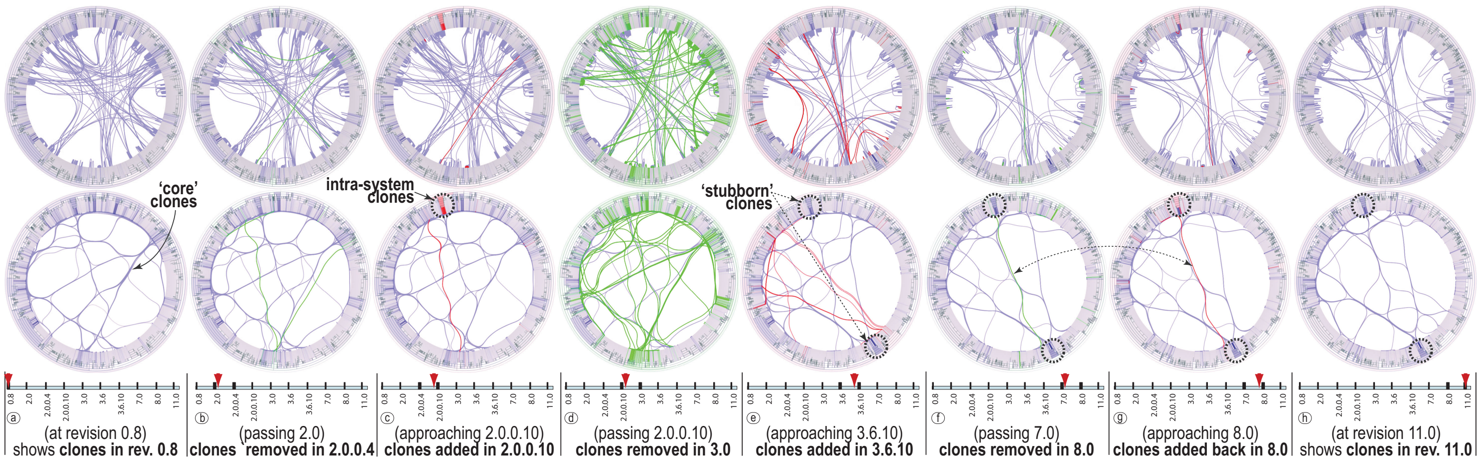


Figure 4: Small multiples visualization for clones in Mozilla Firefox for five selected revisions (Sec. 4.2)

nodes is easier than visually following bundles. In other words, node colors help finding *aggregated* patterns, *e.g.* high-clone-density systems during the evolution, while bundle changes help seeing which particular *subsystems* share such clones. We also see a red spot in revision 2.0.0.10 (Fig. 5 c): This is a subsystem where many *intra-system* clones have been added. Seeing such clones without node coloring would be hard, since their (bundled) edges are very short.

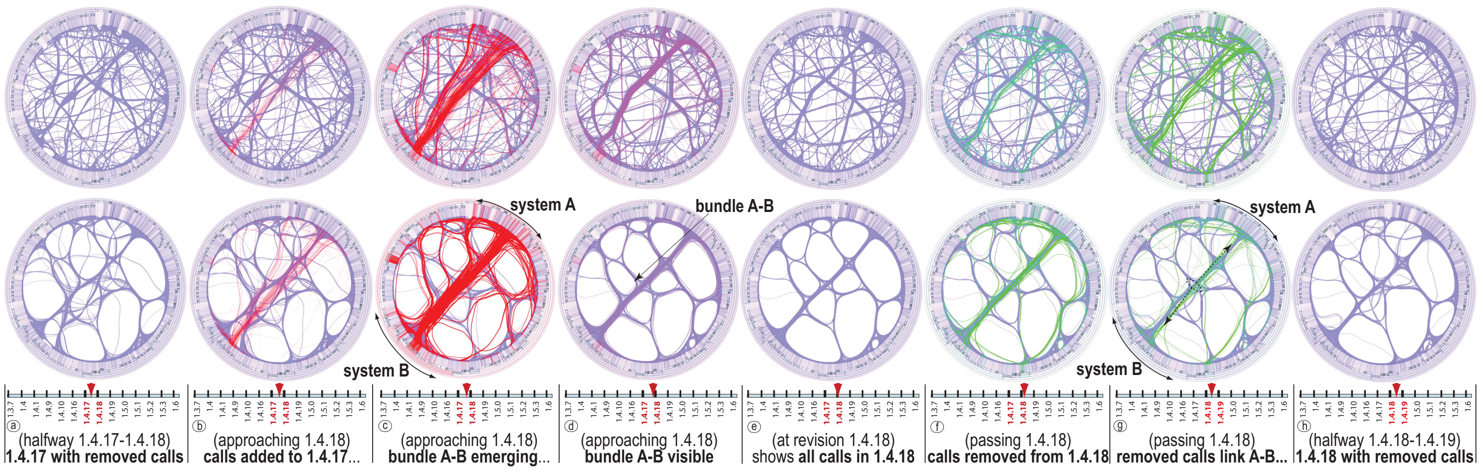
The transition between revisions 7.0 and 8.0 shows an interesting event: First, several "stubborn" clones are removed (green edges shown after passing revision 7.0) Next, clones between the *same* files are added back again (red edges shown when approaching revision 8.0). This typically happens when one modifies related code in two subsystems *e.g.* by rewriting it by independently applying twice the same given design pattern. However, developers were likely not aware of the clones, otherwise we would expect the clone to be removed during such a perfective refactoring. Finally, comparing the first and last frame shows that the core clone pattern did not change significantly. Also, the bundle pattern shows that clones connect *unrelated* subsystems, *i.e.* nodes in the radial icicle plot that are not close to each other, hence not in the same parent system. This is a negative sign for code quality, since removing such clones requires system-wide understanding and refactoring.

As a second example, we extracted a compound digraph with folders, files, and functions (forming the hierarchy) and function calls (forming the associations) from 14 revisions of the Wicket open-source software [42]. Next, we build the same union hierarchy as in the first example (8799 nodes), and compute correspondences based on the fully qualified signatures of (caller, callee) pairs. We obtain 11953 unique edges and 92810 total edges. Figure 6 shows several frames from the graph-sequence visualization. To better illustrate the animated transitions, we focus here on a short period (three revisions). This visualization helps reasoning about the system's (change of) modularity, a challenging task in program comprehension [2]. The interpretation is as follows: The stable pattern (blue bundles) shows the stable control-flow logic of the system, *i.e.* calls that do not change much across versions. We see that this pattern is quite complex, *i.e.* connects many subsystems in different hierarchy parts, so the overall modularity of this software is *and* stays relatively low. In more detail, we see that in version 1.4.18, a significant coupling is added between systems A and B (large red bundle A-B, Fig. 6 c). Interestingly, at the *same* moment (1.4.18), many calls are removed between the same systems (large green bundle A-B, Fig. 6 f). This indicates a refactoring of the A-B system interaction – note the similarity with the clone insertion-and-deletion pattern and its interpretation discussed above for the Firefox dataset.



'core' clones
intra-system clones
'stubborn' clones
(a) (at revision 0.8) shows clones in rev. 0.8
(b) (passing 2.0) clones removed in 2.0.0.4
(c) (approaching 2.0.0.10) clones added in 2.0.0.10
(d) (passing 2.0.0.10) clones removed in 3.0
(e) (approaching 3.6.10) clones added in 3.6.10
(f) (passing 7.0) clones removed in 8.0
(g) (approaching 8.0) clones added back in 8.0
(h) (at revision 11.0) shows clones in rev. 11.0

Figure 5: Sequence-based visualization for clones in Firefox (8 frames). Top row: HEB bundling. Bottom row: KDEEB bundling (Sec. 4.2)



system A
bundle A-B
system B
system A
system B
(a) (halfway 1.4.17-1.4.18) 1.4.17 with removed calls
(b) (approaching 1.4.18) calls added to 1.4.17...
(c) (approaching 1.4.18) bundle A-B emerging...
(d) (approaching 1.4.18) bundle A-B visible
(e) (at revision 1.4.18) shows all calls in 1.4.18
(f) (passing 1.4.18) calls removed from 1.4.18
(g) (passing 1.4.18) removed calls link A-B...
(h) (halfway 1.4.18-1.4.19) 1.4.18 with removed calls

Figure 6: Sequence animation – Wicket call graphs (8 frames around release 1.4.18). Top: SBEB bundling. Bottom: KDEEB bundling (Sec. 4.2)

5 Discussion

5.1 Streaming *vs* sequence graphs

In Sections 3 and 4 we have presented two techniques for visualizing streaming and sequence graphs. One question is: Can we use the streaming algorithm for a sequence graph, and/or conversely? Why do we need two techniques? Below we analyze this aspect.

5.1.1 Streams with sequence-based visualization

For the first experiment, we convert our France air-traffic streaming graph (Sec. 3.2) to a sequence graph of 7 keyframes. For this, we divide the 7-days stream into 7 one-day periods. Edges are assigned to keyframes based on start time. Next, we add correspondences between edges in consecutive keyframes (days) whose geographic start and end locations are very similar and flight IDs are identical. We obtain a 7-keyframe sequence, with 8811 unique edges (when counting corresponding edges as one), and 54K edges in total.

The attached videos show a sequence-based visualization of this dataset. As visible, bundled patterns are much less structured, and their change is harder to follow. This is not too surprising, since the stream-to-sequence conversion quantized the fine-grained time information. Hence, while the streaming-based visualization uses this information to *continuously* bundle edges as they appear, the sequence-based visualization only bundles at keyframes, and uses edge interpolation in between. Additionally, visualizing streams as graph sequences involves delicate data modifications, *e.g.* cutting the stream at possibly irrelevant moments into disjunct chunks, and adding edge-correspondences that may not be meaningful. When such a transformation is not evident, and when fine-grained time data is important for comprehension, one should not visualize graph streams as graph sequences.

5.1.2 Sequences with stream-based visualization

For the second experiment, we convert our Wicket graph sequence (Sec. 4.2) to a streaming graph, by inserting 100 uniformly-spaced time moments between each two consecutive keyframes. Figure 7 shows three frames from the resulting animation, taken between revisions 1.5.0 and 1.5.1. The sequence method (top row) clearly shows a stable core indicating unchanging call patterns (blue bundles), and also outlines the removed calls (green) and added calls (red). The streaming method (bottom row), although doing a good job in creating a smooth and stable bundling, cannot emphasize such additions and removals, since it has no correspondence data to separate the treatment of stable and (dis)appearing edges.

5.2 Scalability

The streaming graph visualization 3.1 has a complexity of $O(|\tilde{E}|)$ per animation frame, where $|\tilde{E}|$ is the average number of edges in any time-window of size Δt at any moment t in the stream. This is

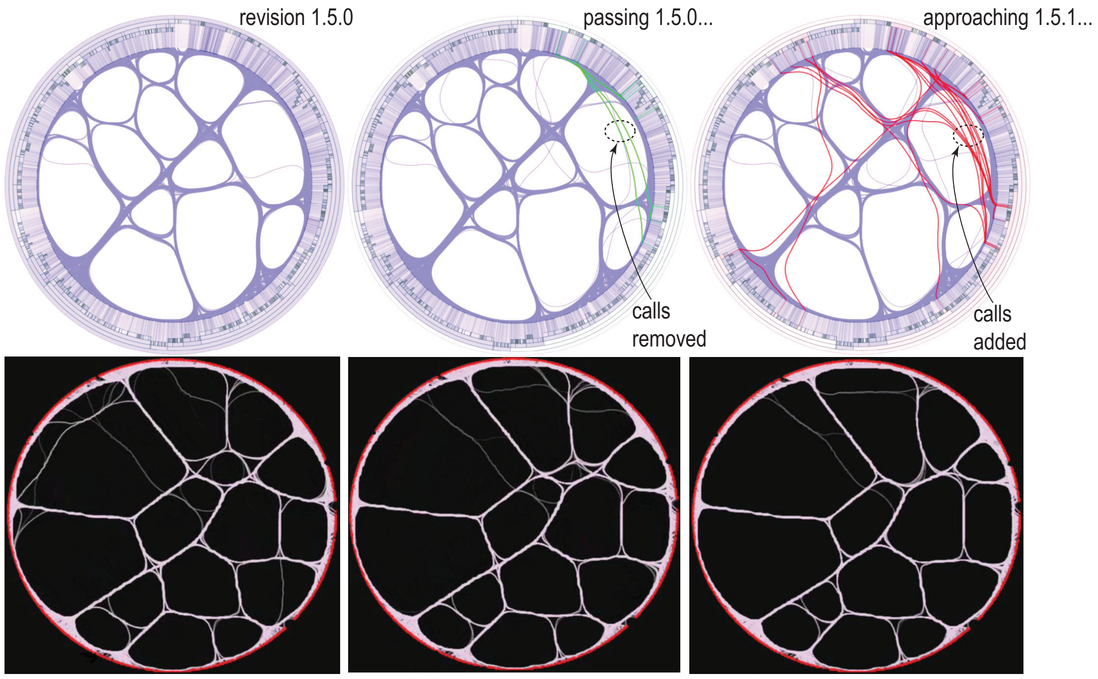


Figure 7: Streaming visualization of graph sequence (3 frames around revision 1.5.0, Wicket software dataset)

so since we run the bundling process in sync with the stream time, as explained in Sec. 3.1. In other words, there is a single density-splat and advection step for each edge present in a frame. In comparison, the method in [28] is $O(|\tilde{E}|^2)$ per frame. We implemented our graph streaming and sequence visualizations in C# using the KDEEB algorithm which is itself written in C# using OpenGL 1.1, and run them on a 2.3 GHz PC with 8 GB RAM and an NVidia GT 480 card. On this platform, producing one streaming-animation frame took 0.05 seconds for the US dataset ($|\tilde{E}| = 2$K edges on average) and 0.17 seconds/frame for the France dataset ($|\tilde{E}| = 15$K edges on average). Per frame, we are roughly 10 times faster than the original KDEEB ([22], Tab. 1), which is expected, as we do only one iteration per frame (see Sec. 3.1). In contrast, if we were to use FDEB, we would need, for the US dataset, 19 seconds/frame on comparable hardware ([19], Sec. 4.2), or 6 seconds/frame for a graph of $|\tilde{E}| = 900$ edges on a 1.7 GHz PC ([28], Fig. 12). Of course, the total time needed for a stream depends on the stream's length.

The sequence graph visualization (Sec. 4.1) has a complexity of $O(BN)$ for a sequence of N graphs, and an underlying bundling algorithm of complexity B. This is basically the same cost as in [28], modulo the fact that our algorithm B is faster, as already explained. However, note that our visualization is different, since we (a) emphasize appearing and disappearing edges and (b) smoothly interpolate consecutive bundled layouts by using edge correspondences.

5.3 Bundling algorithm choice

For streaming visualizations, KDEEB is arguably a very good solution: KDEEB works for general graphs, produces bundles with little clutter even for very complex graphs, and is robust and simple to use. However, the most important point is that KDEEB's design allows to *incrementally* update the graph *during* the bundling. In contrast, most other bundling layouts require a full recomputation of the bundling when the input graph changes. This is due to various technical factors, *e.g.* use of spatial search data structures and compatibility metrics that need reinitialization upon graph changes [12, 8, 16], or encoding the bundle polylines separately from the input graph's straight-line edges [16, 25, 34]. FDEB comes closest to KDEEB in flexibility, as it represents (partially) bundled edges as a set of unstructured polyline curves, so it can be used for incremental smooth bundling upon input graph changes. However, KDEEB's linear complexity in the input graph size makes it more suitable than FDEB which is quadratic in the same input size.

For sequence visualizations, any bundling algorithm can be technically used. However, here KDEEB also proved better than alternatives. Figure 5 shows the differences between using HEB (top row) *vs* KDEEB (bottom row). As visible, HEB produces less structured and compact bundles. A similar effect can be seen in StreamEB [28]. Figure 6 shows the differences between using SBEB (top row) *vs* KDEEB (bottom row). Here, SBEB produces actually too much structure – the bundles have too many branches. KDEEB produces less clutter than SBEB, but more structure than HEB, thereby offering a good visual balance.

5.4 Parameters

Our streaming-based visualization uses the same edge sampling, smoothing, kernel size, and density-map resolution parameters as KDEEB [22]. The parameters added by our streaming method are the size of the time-window Δt and time-step δt for sliding this window (see Alg. 1). Δt controls how much one sees in one animation frame: Larger Δt values show more (bundled) edges, but inherently smooth out the dynamics of the animation. Smaller values show more of the instantaneous graph $G(t)$, but make short-lived edges (dis)appear faster. In our examples, we used a Δt corresponding to a 5% change in the number of edges in $\tilde{G}$, so that animation goes faster over uninteresting time periods, similarly to [28]. δt controls the ratio between the animation speed and the stream's own speed *and* also the bundling tightness. Large δt values subsample the stream, *i.e.* make the animation go faster and show less tight bundles, since, as outlined in Sec. 3.1, bundling occurs in sync with the stream time. Smaller δt values supersample the stream, *i.e.* make the animation go slower and also create tighter bundles. In practice, getting tight bundles with KDEEB requires roughly $I = 5..10$ iterations [22]. Hence, we set δt to $1/I$ of the average edge lifetime in the stream. A good side-effect of this setting is that bundling reflects the edge lifetime: Short-lived edges, likely outliers, do not strongly bundle. Long-lived edges, which contribute to the coarse-scale structure of the graph, get strongly bundled. Apart from Δt and δt, our algorithm has no other parameters.

5.5 Limitations

Currently, we showed that we can bundle graph streams and sequences in a fast, smooth, and clutter-free manner, and that such animations help assessing connection stability and spot fast-changing bundles (Secs. 3.2 and 4.2). However, the animation and visual mapping metaphors, *i.e.* speed, shape, tightness, and shading of bundles, would need to be adapted to support seeing finer-grained events of interest such as bundle splitting, or merging; similar bundles in far-apart time frames; and separating bundles based on additional edge attributes. Also, a quantitative and qualitative measurement of the effectiveness of animated bundles is needed.

6 Conclusion

We have presented two algorithms for the animated visualization of graph streams and sequences. By exploiting the smoothness, stability, speed, and incremental nature of the recent KDEEB image-based bundling algorithm, we succeed in creating streaming graph animations which exhibit the same desirable properties. Next, we use the same algorithm to generate sequence-based graph visualizations where edge appearance and disappearance events are emphasized. We apply our techniques on four large datasets, and present evidence that supports our choice for KDEEB as underlying layout.

Future work can address animation, visualization, and interaction refinements to find and emphasize finer-grained events of interest, such as bundle merging and splitting, and support tasks such as detecting graph patterns that match problem-specific patterns of interest. Furthermore, user evaluations can help in validating and refining the design choices presented here.

References

[1] N. Andrienko and G. Andrienko. *Exploratory analysis of spatial and temporal data: a systematic approach*. Springer, 2006.

[2] F. Beck and S. Diehl. On the impact of software evolution on software clustering. *Empirical Software Engineering*, 2012. DOI: 10.1007/s10664-012-9225-9.

[3] C. Binuccia, U. Brandes, G. D. Battista, W. Didimo, M. Gaertler, P. Palladino, M. Patrignani, A. Symvonis, and K. Zweig. Drawing trees in a streaming model. *Inform. Process. Lett.*, 112(11):418–422, 2012.

[4] I. Boyandin, E. Bertini, and D. Lalanne. A qualitative study on the exploration of temporal changes in flow maps with animation and small-multiples. *Comp. Graph. Forum*, 31(3):1005–1014, 2012.

[5] M. Burch, F. Beck, and S. Diehl. Timeline trees: Visualizing sequences of transactions in information hierarchies. In *Proc. AVI*, pages 75–82, 2008.

[6] M. Burch and S. Diehl. TimeRadarTrees: Visualizing dynamic compound digraphs. *Comp. Graph. Forum*, 27(3):823–830, 2008.

[7] D. Comaniciu and P. Meer. Mean shift: A robust approach toward feature space analysis. *IEEE TPAMI*, 24(5):603–619, 2002.

[8] W. Cui, H. Zhou, H. Qu, P. Wong, and X. Li. Geometry-based edge clustering for graph visualization. *IEEE TVCG*, 14(6):1277–1284, 2008.

[9] M. Dickerson, D. Eppstein, M. Goodrich, and J. Meng. Confluent drawings: Visualizing non-planar diagrams in a planar way. In *Proc. Graph Drawing*, pages 1–12, 2003.

[10] T. Dwyer, K. Marriott, and M. Wybrow. Integrating edge routing into force-directed layout. In *Proc. Graph Drawing*, pages 8–19, 2007.

[11] G. Ellis and A. Dix. A taxonomy of clutter reduction for information visualisation. *IEEE TVCG*, 13(6):1216–1223, 2007.

[12] O. Ersoy, C. Hurter, F. Paulovich, G. Cantareira, and A. Telea. Skeleton-based edge bundles for graph visualization. *IEEE TVCG*, 17(2):2364 – 2373, 2011.

[13] C. Erten, S. Kobourov, V. Le, and A. Navabi. Simultaneous graph drawing: Layout algorithms and visualization schemes. In *Proc. Graph Drawing*, pages 437–449, 2004.

[14] D. Forrester, S. Kobourov, A. Navabi, K. Wample, and G. Yee. Graphael: A system for generalized force-directed layouts. In *Proc. Graph Drawing*, pages 454–464, 2004.

[15] Y. Frishman and A. Tal. Online dynamic graph drawing. In *Proc. EuroVis*, pages 75–82, 2007.

[16] E. Gansner, Y. Hu, S. North, and C. Scheidegger. Multilevel agglomerative edge bundling for visualizing large graphs. In *Proc. PacificVis*, pages 187–194, 2011.

[17] E. Gansner and Y. Koren. Improved circular layouts. In *Proc. Graph Drawing*, pages 386–398, 2006.

[18] D. Holten. Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. *IEEE TVCG*, 12(5):741–748, 2006.

[19] D. Holten and J. J. van Wijk. Force-directed edge bundling for graph visualization. *Comp. Graph. Forum*, 28(3):670–677, 2009.

[20] M. Huang, P. Eades, and J. Wang. On-line animated visualization of huge graphs using a modified spring algorithm. *JVLC*, 9(6):623–645, 1998.

[21] C. Hurter, O. Ersoy, and A. Telea. Moleview: An attribute and structure-based semantic lens for large element-based plots. *IEEE TVCG*, 17(12):2600–2609, 2011.

[22] C. Hurter, O. Ersoy, and A. Telea. Graph bundling by kernel density estimation. *Comp. Graph. Forum*, 31(3):435–443, 2012.

[23] C. Hurter, B. Tissoires, and S. Conversy. FromDaDy: Spreading data across views to support iterative exploration of aircraft trajectories. *IEEE TVCG*, 15(6):1017–1024, 2009.

[24] A. Lambert, R. Bourqui, and D. Auber. 3D edge bundling for geographical data visualization. In *Proc. Information Visualisation*, pages 329–335, 2010.

[25] A. Lambert, R. Bourqui, and D. Auber. Winding roads: Routing edges into bundles. *Comp. Graph. Forum*, 29(3):432–439, 2010.

[26] Mozilla. Firefox repository, 2012. `http://www.mozilla.org/en-US/firefox/fx`.

[27] Q. Nguyen, P. Edges, and S.-H. Hong. StreamEB results, 2012. `http://rp-www.cs.usyd.edu.au/~qnguyen/streameb`.

[28] Q. Nguyen, P. Edges, and S.-H. Hong. StreamEB: Stream edge bundling. In *Proc. Graph Drawing*, pages 324–332, 2012.

[29] J. R. Pate, R. Tairas, and N. A. Kraft1. Clone evolution: A systematic review. *J. Soft. Maint. Evol. Res. Pract.*, 2012. DOI:10.1002/smr.579.

[30] D. Phan, L. Xiao, R. Yeh, P. Hanrahan, and T. Winograd. Flow map layout. In *Proc. InfoVis*, pages 219–224, 2005.

[31] H. Qu, H. Zhou, and Y. Wu. Controllable and progressive edge clustering for large networks. In *Proc. Graph Drawing*, pages 399–404, 2006.

[32] D. Reniers, L. Voinea, O. Ersoy, and A. Telea. The Solid* toolset for software visual analytics of program structure and metrics comprehension: From research prototype to product. *Sci. Comput. Program.*, 2012. doi:10.1016/j.scico.2012.05.002.

[33] R. Scheepens, N. Willems, H. van de Wetering, G. Andrienko, N. Andrienko, and J. J. van Wijk. Composite density maps for multivariate trajectories. *IEEE TVCG*, 17(12):2518–2527, 2011.

[34] D. Selassie, B. Heller, and J. Heer. Divided edge bundling for directional network data. *IEEE TVCG*, 19(12):754–763, 2011.

[35] SolidSource IT. SolidSDD Clone Detector, 2012. `http://www.solidsourceit.com`.

[36] Statistical Computing. US flights dataset, 2012. `http://stat-computing.org/dataexpo/2009/the-data.html`.

[37] A. Telea and D. Auber. Code flows: Visualizing structural evolution of source code. *Comput. Graph. Forum*, 27(3):831–838, 2008.

[38] A. Telea and O. Ersoy. Image-based edge bundles: Simplified visualization of large graphs. *Comp. Graph. Forum*, 29(3):543–551, 2010.

[39] B. Tversky, J. Morrison, and M. Betrancourt. Animation: Can it facilitate? *Intl. J. Human Computer Studies*, 57:247–262, 2002.

[40] R. van Liere and W. de Leeuw. GraphSplatting: Visualizing graphs as continuous fields. *IEEE TVCG*, 9(2):206–212, 2003.

[41] T. von Landesberger, A. Kuijper, T. Schreck, J. Kohlhammer, J. van Wijk, J.-D. Fekete, and D. Fellner. Visual analysis of large graphs: State-of-the-art and future research challenges. *Comp. Graph. Forum*, 30(6):1719–1749, 2011.

[42] Wicket. Apache Wicket, 2012. `http://wicket.apache.org`.

[43] H. Zhou, X. Yuan, W. Cui, H. Qu, and B. Chen. Energy-based hierarchical edge clustering of graphs. In *Proc. PacificVis*, pages 55–62, 2008.