Parallel Edge Splatting for Scalable Dynamic Graph Visualization

Michael Burch, Corinna Vehlow, Fabian Beck, Stephan Diehl, Member, IEEE, and Daniel Weiskopf, Member, IEEE Computer Society

Abstract—We present a novel dynamic graph visualization technique based on node-link diagrams. The graphs are drawn side-by-side from left to right as a sequence of narrow stripes that are placed perpendicular to the horizontal time line. The hierarchically organized vertices of the graphs are arranged on vertical, parallel lines that bound the stripes; directed edges connect these vertices from left to right. To address massive overplotting of edges in huge graphs, we employ a splatting approach that transforms the edges to a pixel-based scalar field. This field represents the edge densities in a scalable way and is depicted by non-linear color mapping. The visualization method is complemented by interaction techniques that support data exploration by aggregation, filtering, brushing, and selective data zooming. Furthermore, we formalize graph patterns so that they can be interactively highlighted on demand. A case study on software releases explores the evolution of call graphs extracted from the JUnit open source software project. In a second application, we demonstrate the scalability of our approach by applying it to a bibliography dataset containing more than 1.5 million paper titles from 60 years of research history producing a vast amount of relations between title words.

Index Terms—Dynamic graph visualization, graph splatting, software visualization, software evolution.

1 INTRODUCTION

Relationships among a set of objects are typically modeled as a graph consisting of a set of vertices and a set of edges that connect related vertices. Many of these relational datasets can be found in typical applications, not only limited to small datasets. For example, software systems may consist of hundreds or thousands of hierarchically organized classes connected by method calls and other dependencies, or digital libraries may include millions of papers linked by citations and keywords.

A major challenge for the graph drawing community is the efficient and effective computation of graph layouts. There are some generally accepted aesthetic graph drawing criteria such as the minimization of edge crossings, the even distribution of nodes in the frame, or the uniformity of graph link lengths [9, 17]. Force-directed, orthogonal, or hierarchical layouts aim at optimizing a variety of these criteria.

The temporal evolution of a graph introduces another interesting aspect to graph drawing but also comes along with further visualization challenges concerning the additional time dimension. The prevailing approach taken by graph drawing researchers is to use animation to represent the time dimension [10, 12]. This time-to-time mapping is intuitive but has some substantial drawbacks. The exploration of animated graph diagrams leads to high cognitive efforts due to our limited short term memory [1, 33]. For instance, it is difficult to observe trends over longer periods or to compare two non-subsequent time steps. Abrupt changes of the graph might furthermore destroy the viewer’s mental image of the graph (mental map).

Previous research investigated different dynamic graph visualization techniques that show the complete evolution at once in a single image instead of using animation [6, 7, 13, 14]. While these approaches largely prevent the problems of animated diagrams mentioned above, they come along with other problems [3]: due to the limited space that is left for a single graph when drawing all graphs of the sequence in a single diagram, scalability is hard to achieve especially with respect to the number of vertices.

In this paper, we introduce a single-image visualization technique for exploring dynamic and hierarchically organized graphs, now focusing on scalability. We map each time step of the graph to a narrow rectangular area on screen. The vertices are arranged along the vertical borders of these areas, according to the order implied by the hierarchy. Directed graph edges connect the vertices by straight links starting on the left hand side of the area and ending at the right hand side. In large graphs, this strategy leads to a plethora of overlapping edges. We tackle this problem by applying parallel edge splatting—a technique that transforms the edges to a density scalar field by splatting those edges to the screen. Visualizing this field by color mapping allows us to recognize the trajectories of edges even in quite cluttered areas. The term “parallel” indicates the layout of graph vertices on parallel lines. Parallel edge splatting adopts the idea of density representation from graph splatting [34], yet is complementary because we splat edges instead of vertices.

Our visualization tool provides several interactive features to manipulate and navigate dynamic graph data. With hierarchically structured graph vertices, expanding and collapsing of this hierarchy may be used to aggregate a certain number of nodes and their corresponding relations. The graph data can also be aggregated in the time dimension and filter functions can be applied to only display relations within a given weight interval. Pixel-based elements can be enlarged by a lens function. The tool design follows the information seeking mantra: overview first, zoom and filter, then details-on-demand [30]. Furthermore, we formalize graph patterns and provide functionality to interactively highlight these patterns.

In two application scenarios, we demonstrate how the above combination of techniques increases the scalability of the single-image dynamic graph visualizations. First, we show the usefulness of our approach by a comprehensive case study on the evolution of the JUnit open source software system. Second, we analyze the evolution of title words occurring in more than 1.5 million paper titles acquired in the DBLP [25].

2 RELATED WORK

Visualizing dynamic relational data is a challenging task due to a number of visual dimensions to be represented in the same view: vertices, edges, and time. Node-link diagrams are effective for perceiving related objects and solving path-related tasks. However, graph datasets may reach sizes with which traditional graph layout algorithms cannot deal efficiently. A mapping of all edges of a dense graph to a 2D plane leads to visual clutter [29] caused by many edge crossings, even after applying sophisticated layout algorithms that take various aesthetic criteria for graph drawing into account [4, 37]. The time dimension
is typically displayed by a natural time-to-time mapping in graph animation [10, 12]. Complex algorithms have to be applied to reduce the user’s cognitive load, especially to preserve the user’s mental map of the graph diagram over time. A recent state-of-the-art report by van Landesberger et al. [36] surveys existing research and challenges for visualizing large static and dynamic graphs.

Instead of representing the time dimension in the graph data by animation, we map time to space and show the subsequent graphs in a static diagram. Our approach is inspired by the TimeArcTrees visualization [14], where graph nodes are arranged on vertical lines and directed edges are drawn as arcs. In TimeArcTrees, the size of each arc depends on the number of nodes that are located between source and target node, whereas the position of each arc (left or right hand side of the vertical axis) depends on whether the corresponding edge is an upward or downward edge. Visual clutter is reduced by optimizing the arrangement of the nodes. In contrast, the technique presented in this paper uses straight lines instead of arcs to represent the graph edges and splits vertices so that all edges point from left to right.

Related to TimeArcTrees, the TimeRadarTrees [7] and the TimeSpiderTrees [14] approaches use a radial layout and draw each graph of the sequence in an annulus (ring). These single-image dynamic graph visualization techniques are limited to graphs consisting of a few dozens of vertices because each vertex consumes some space on a vertical line or a circumference of a circle. Another approach is to stack a sequence of node-link diagrams on top of each other such that the nodes representing the same vertex are aligned [6, 13]. This, however, increases the problem of clutter produced by overlapping visual artifacts.

The visual representation produced by our technique is visually similar to parallel-coordinate plots [21]. While polylines in parallel coordinates typically span all coordinate axes, the lines in our approach represent graph edges and are contained within the drawing area of a single graph of the sequence. Therefore, the visual signatures and interpretation differ.

Similar to the density model of parallel edge splatting, there are density-based variants of parallel-coordinate plots, e.g., [2, 5, 16, 22, 26, 38]. In particular, there are some splatting techniques for parallel coordinates [11, 15, 40]. Our approach is technically similar to splatting of parallel coordinates, but is based on discrete graph data; vertices are laid out on discrete positions on vertical lines and edges are discrete objects that cover a set of discrete points on the 2D plane. Furthermore, association rules for text mining can be visualized in a way that looks similar to our dynamic graph visualization technique: the elements are arranged on vertical lines; sequence rules connect these elements by straight lines from left to right on a timeline [39].

Van Liere and de Leeuw [34] introduced a graph splatting technique to transform graph data into a 2D scalar field, which is used as a basis to render a heatmap, a height field, or a set of contours. The main difference to our work is the fact that they only use the vertex information of the graph to generate a splat field. The relationship information—on which our approach is built—is not explicitly visualized. Holten and van Wijk [20] used a force-directed edge bundling approach in which a pixel-based representation is applied to emphasize regions with many edges. A similar idea may also be applied to vertically arranged vertices [19]. Telea and Ersoy use a combination of distance-based splatting and skeletonization to generate image-based edge bundles [32]. In contrast to our work, they focus on graphs that do not change over time, whereas we focus on time-varying directed graphs. We will discuss work related to our application scenarios in Section 5.

3 Visualization Technique

Our visualization technique targets the problem of displaying dynamic weighted directed graphs in a single static view. We use parallel vertical axes to place the aligned graph vertices. The initial vertex position follows a predefined hierarchical order if there is any available. Directed graph edges are visually encoded as direct links between two subsequent axes and are oriented from left to right. To overcome the problem of visual clutter in dense graphs, we apply splatting and anti-aliasing to the graph links, obtaining a scalable visualization. Time step after time step of the dynamic graph is arranged so that the user can perceive evolving graph structures.

3.1 Data Model

We model a directed multi-graph in the graph-theoretic sense as

\[ G = (V, E) \]

where \( V \) denotes the set of vertices and \( E \subseteq V \times V \times \mathbb{W} \) denotes the set of directed adjacency edges that consists of ordered triples of elements. The first component denotes the start vertex, the second component the target vertex, and the third component \( W \in \mathbb{W} \) a finite multi-set of positive real-valued numbers representing the corresponding multi-edge, i.e., the weights of the relations. Additionally, each graph vertex \( v \in V \) is hierarchically organized in a tree structure

\[ H = (V, E) \]

where \( V \) is the vertex set and \( E \subseteq V \times V \) are the directed inclusion edges implied by the hierarchy such that \((v,w) \in E \) if \( v \) is the parent of \( w \).

A dynamic weighted directed multi-graph in an information hierarchy is modeled as a finite sequence of graphs

\[ \Gamma := \{G_1, G_2, \ldots, G_n\} \]

where \( G_i = (V_i, E_i) \), \( V_i \subseteq V \), \( E_i \subseteq E \), \( 1 \leq i \leq n \). We assume that the hierarchical organization of the graph vertices remains constant for all graphs in the sequence.

3.2 Graph Layout

To illustrate the visual encoding of such dynamic graph data by parallel edge splatting, we start with a single static directed graph as shown in Figures 1 (a) and (b). The right part of Figure 1 (a) presents a possible layout of the adjacency information of the directed weighted graph embedded into the 2D plane. The small graph example consists of five nodes labeled A, B, C, D, and E. There are eight directed weighted edges, where the weights are visualized by color coding. One of the edges is a self-edge (at node E). The left part of Figure 1 (a) shows the hierarchical organization of the graph vertices as a traditional node-link diagram.

Figure 1 (b) illustrates the transformation of the graph data in Figure 1 (a) into our new layout. First, the graph nodes are mapped to a pair of parallel and vertically aligned lines; the mapping is identical for both lines. These lines build the axes for visually encoding the directed edges starting at the vertical position of the corresponding start node on the left axis and pointing to the vertical position of the corresponding target node on the right axis. To retain the information about the hierarchical organization of the graph nodes, we add an orthogonal node-link tree diagram in the left part of Figure 1 (b), where the tree nodes are vertically aligned with the corresponding graph nodes on the vertical axes.

Initially, there is no special ordering of the graph nodes on the vertical axes. In our approach, we use the hierarchical organization to generate a total order of the nodes where the order of the subhierarchies in each level is not yet fixed. We use a depth-first traversal of the graph and follow subhierarchies according to their order in memory. The nodes are placed equidistantly from top to bottom. If there is no hierarchy defined (e.g., for a graph without directly associated hierarchy), a hierarchy needs to be constructed in a preprocessing step, e.g., by employing hierarchical clustering of the graph nodes. Furthermore, nodes are not necessarily arranged on equidistant positions; if a metric is available to define node distances on the axis, non-uniform distances can be used.

A dynamic graph is a graph that changes over time and can be modeled as a sequence of single graphs \( \Gamma = \{G_1, G_2, \ldots, G_n\} \). Our technique is readily extended to apply it to evolving graph structures. For displaying \( n \) graphs of a sequence, we just need \((n+1)\) copies of the vertical axis and place these axes side-by-side. The space between two
Visualizing graphs in node-link diagrams may lead to many edge crossings and, hence, to visual clutter in the display. This phenomenon even occurs for graphs with a small number of vertices. Traditional graph layout algorithms become very time-consuming due to their high computational complexity, already for a dense graph with just a few hundred vertices. Furthermore, even sophisticated layout techniques may generate unsatisfying results in many situations. The visualization of an additional time dimension and an additional hierarchical organization is a challenging task for these layout approaches.

As discussed before, we represent time-varying graphs by a 1D layout of the nodes and a time-to-space mapping of the dynamics. A drawback of restricting the layout of graph nodes to 1D is a further increase of visual clutter caused by many edge crossings.

Therefore, a graph drawing method for dense coverage by edges is crucial. We follow the general strategy of transforming plots with opaque plotting elements (i.e., lines in our case) to corresponding density plots. In our case, graph edges are transformed to density distributions. We adopt a splatting approach, projecting the density representation of the edges to image space.

The basic splatting algorithm has only little graphics requirements. First, it needs a framebuffer or similar storage for the density image. Just a single value needs to be stored per pixel in the framebuffer. The number representation of such a pixel value should be chosen to capture the dynamic range of densities. Typically, somewhere between 16-bit fixed-point to 32-bit floating-point number format is appropriate. Furthermore, the framebuffer has to have support for additive blending, i.e., the plus operator according to Porter and Duff [28]. As second requirement, the splatting algorithm needs a line rastertexting method that is capable of rendering lines of a specified intensity to the framebuffer. Any of today’s typical graphics systems meets these requirements easily, regardless of whether it is graphics hardware run by low-level graphics API (OpenGL or DirectX) or whether it is a high-level software API, such as the Java 1.6.2 SE Runtime Environment used in our case.

With these graphics components, the basic splatting algorithm works as illustrated in Figure 2. Here, we only consider one pair of vertical axes; further time steps are included by applying the splatting algorithm time-after-time at subsequent axes pairs. The rendering algorithm visits all graph edges and rasterizes their line representations. The weights of the edges determine the line intensity. Additive blending is enabled throughout the rendering process. Since additive blending is commutative, the edges can be rendered in arbitrary order. At the end of the basic splatting process, the framebuffer contains the 2D image of the edge densities. Figure 3 demonstrates the effect of this edge splatting approach.

Since the density image may contain densities with high dynamic range, a postprocessing stage is recommended. Here, some data mapping is computed to reduce the dynamic range. We recommend a logarithmic mapping. Furthermore, color coding can be applied to visualize the transformed densities.

As presented, the algorithm may lead to aliasing for the line densities rendered, which may show up as “staircase” artifacts. Therefore, line filtering should be incorporated in the basic splatting algorithm. One option is to draw “thick” lines with a smooth fall-off behavior. This can be accomplished by rendering quadrilaterals whose extents rendered, which may show up as “staircase” artifacts. Therefore, line filtering should be incorporated in the basic splatting algorithm. One option is to draw “thick” lines with a smooth fall-off behavior. This can be accomplished by rendering quadrilaterals whose extents are given by the lengths and widths of the lines. The quadrilaterals are textured with a prefiltered image of the “thick” line. Typical filters include Gaussian or triangle filters. The alternative option is to apply filtering after the line rendering of the basic splatting algorithm. Here, the filter operators are applied just as 2D image filters. Since both blending of lines and filtering with Gauss or triangle functions are linear operators, the order of execution does not change the final result. Therefore, both anti-aliasing approaches are equivalent. In our tool, we use the second option with a triangle filter of size $3 \times 3$. 

Fig. 2. Edge splatting is applied to help a viewer trace links even in dense graph regions. The leftmost subfigure shows a square to be discretized and enlarged in the center subfigure. The sum of weight coverage implied by adding each link weight is color-coded in the rightmost subfigure. Color coding is changed due to a new maximum value.

Fig. 3. Visualization of a real-world dynamic graph with homogeneous edge weights (only a detail of the visualization shown)—without (a) and with edge splatting (b).
3.4 Interactive Features

Although we generate static images from dynamic data, our visualization tool includes support for a list of important interactive features. Following the visualization seeking mantra [30], we first provide an overview of the time-varying graph data. We allow zooming, filtering, and details-on-demand as potential next steps. Furthermore, the dynamic graph data can be aggregated to the vertex as well as in the time dimensions. The hierarchy can be used to expand or collapse vertices that belong to the same subhierarchy and the corresponding edges are also adjusted to the new view. The most important interaction elements are the following:

Pixel lens. Regions of the pixel-based diagram can be shown in an enlarged display. Single pixels can be easily selected and explored in the lens view.

Vertical and horizontal aggregation. A number of neighboring vertices on a vertical line can be selected and aggregated. Similarly, neighboring graphs can be selected and aggregated; see Figure 4, where a longer sequence of graphs is summarized to five graphs. The edges involved are aggregated and their new weights are computed as the arithmetic means of the original weights.

Expansion and collapse of subhierarchies. Clicking on a node in the hierarchy view expands the underlying tree structure. If the tree is already expanded, it will be collapsed again. Figure 4 provides an example where two subhierarchies are collapsed.

Edge brushing. By selecting a number of start nodes in the node selection bar, all links are highlighted that lie on paths to a given depth by putting them to the foreground. All other links are drawn transparently in the background; see Figure 4, which highlights the selected nodes applying depth 1.

Data filtering. Database queries to the full dataset are supported to select a subset of the data to be visualized. Queries may be applied to accompanying data fields that are not even shown in the graph visualization. In addition, we support filtering with respect to the weights of the graph edges and the graph density field. Furthermore, multi-edges can be represented by sum of weights, maximum of weights, or average of weights.

4 Categorization of Visual Patterns

Plotting the dynamic graphs using the new visualization technique shows types of recurring visual patterns. Understanding these patterns and their relationship to the visualized data structure provides an instrument to quickly extract information from the visualization. In the following, we discern static patterns, which are patterns in a single graph, from dynamic patterns, which cover temporal phenomena.

4.1 Static Patterns

For analyzing static patterns, we look at a single graph represented in one vertical stripe. The visual patterns that may occur in such a stripe indicate a particular graph property. Figure 5 schematically sketches some patterns that will be described in detail in this section. The following definitions prepare the ground for formally defining the idealized graph properties underlying the visual patterns. As a starting point, we describe a simplified version of the graph and hierarchy structure. The idea is that the patterns are properties of that simplified structure. To this end, we modify the hierarchy by collapsing inner nodes and propagating the edge information to the collapsed vertices. First, we have to map each vertex in the aggregated structure to a set of vertices in the original structure, which we implement by defining a mapping function $\rho_d : V \rightarrow 2^V$ (mapping a node to a set of nodes), where $d \in \mathbb{N}$ denotes the degree of simplification, or in other words, the number of aggregation steps:

$$\rho_d(v) := \begin{cases} \{v\} & \text{if height}(v) > d \\ \{v\} \cup \text{descendants}(v) & \text{if height}(v) = d \\ \emptyset & \text{otherwise} \end{cases}$$

Here, $\text{height}(v)$ is the height of a vertex $v$ in the hierarchy $H$, which is the maximal length of a downward path to a leaf from vertex $v$, and $\text{descendants}(v)$ is the set of children, grandchildren, etc. of vertex $v$. Nodes $v$ are not considered in the aggregated structure when $\rho_d(v) = \emptyset$. If $d = 0$, the mapping does not alter the structure ($\rho_0(v) = \{v\}$), whereas $d = 1$ is equivalent to collapsing all inner nodes that only have leaf nodes as children (vertices $v$ where $\text{height}(v) = 1$).

Based on the mapping function $\rho_d$, we are now able to define a new set of multi-edges $E_A(d)$ for the simplified graph structure:

$$E_A(d) := \{(v, w, W) \mid \emptyset \neq W = \bigcup_{v' \in \rho_d(v)} W_{v', w'} \}$$

This set contains all edges between vertices that are not collapsed and furthermore aggregates the edges from or to descendants of collapsed vertices.
Beam. The beam pattern describes a horizontal block of edges. If this block consists of a uniformly distributed edge density, it indicates a set of parallel horizontal edges, i.e., self-edges. If the density information is uneven and forms a cross of high density within the block, this could indicate a cluster (a dense sub-graph) or even a clique (a complete sub-graph). Formally, a function that identifies all nodes with beam property can be defined as follows:

\[ beam(d,n) := \{ v \mid (v,v,W) \in E_A(d), |W| \geq n \} \]

This function returns those nodes that have at least \( n \) self-edges in the aggregated set of edges \( E_A(d) \). Hence, \( n \) represents the minimal strength of the beam in number of edges. The degree of simplification \( d \) is used to control the accuracy of the beam: \( d = 0 \) allows only beams within a vertex, \( d = 1 \) within a hierarchy node collapsed in the first aggregation step, etc.

Cross Beam. Similar to a beam, a cross beam pattern is a solid block of edges that connects a set of vertices on the left side to a set of vertices on the right side. The difference is that these two sets consist of different vertices so that the beam crosses the diagram like a diagonal. Hence, the cross beam pattern reveals that there is a strong directed relationship from the first set of vertices to the second one. The related formal definition is also similar to the one of the beam property, though pairs of nodes are needed to identify a cross beam. Again, \( n \) controls the strength of the cross beam and \( d \) the accuracy with respect to the hierarchical structure:

\[ crossbeam(d,n) := \{ (v,w) \mid (v,w,W) \in E_A(d), |W| \geq n, v \neq w \} \]

Hour Glass. A pattern that looks like an hour glass consists of two beams and two cross beams. It may indicate a cluster or clique in the graph that is, in contrast to a single beam, distributed over two different parts of the vertical axis. The hour glass property can be identified as a pair of vertices accumulating beam and cross beam properties:

\[ hourglass(d,n) := \{ (v,w) \mid v,w \in beam(d,n), (v,w),(w,v) \in crossbeam(d,n) \} \]

Fan. The fan pattern denotes a high edge density spreading out from a single point in many directions. This pattern reveals vertices with a high out-degree (fan-out) or a high in-degree (fan-in). The density structure of the fan—gaps and denser parts—provides an overview of the set of vertices connected to the vertices in the center of the fan. Formally, vertices belonging to a fan have (outgoing/incoming) edges to at least \( n \) other vertices. The aggregated set of edges \( E_A(d) \) is used to consider also fans on higher levels of abstraction:

\[ fanout(d,n) := \{ v \mid \{(v,w,W) \in E_A(d)\} \geq n \} \]
\[ fanin(d,n) := \{ v \mid \{(w,v,W) \in E_A(d)\} \geq n \} \]

Gap. A gap at the vertical boundary of the static graph diagrams indicates vertices that do not have outgoing edges or incoming ones. The according definitions consider the degree of simplification \( d \) so that also collapsed inner vertices can be identified as having such an edge degree of zero.

\[ gapout(d,n) := \{ v \mid \{(v,w,W) \in E_A(d)\} = 0 \} \]
\[ gapin(d,n) := \{ v \mid \{(w,v,W) \in E_A(d)\} = 0 \} \]

A visual pattern is an indicator for a certain graph property as described. However, even if we cannot find a particular visual pattern, it does not necessarily mean that there is no such graph property. For instance, the ordering of the vertices may prohibit that the pattern becomes apparent. Moreover, the pattern might be occluded by another pattern that consists of stronger edges.

This problem could be partially addressed by applying different ordering strategies. We may minimize the edge lengths, reduce the number of edge crossings, randomize the order to prevent a bias, or find application-specific semantic sorting criteria. However, finding an appropriate linear arrangement is difficult because an arrangement that is optimal for highlighting one pattern may obscure other patterns. Hence, we choose a different approach to addressing the problem of reliably detecting the static graph properties: the formal definitions of the properties enable identifying those vertices or pairs of vertices that belong to a particular property.

A property is made visible by highlighting the respective nodes and edges in the visualization. This is implemented by adding visual marks to the respective vertices and blending out non-highlighted edges. This approach can be used to unambiguously and exhaustively obtain a particular property on demand. Nevertheless, the visual analysis of patterns is still needed for more explorative searches and may precede the highlighting of graph properties.

4.2 Dynamic Patterns

Complementary to static patterns, dynamic patterns describe the relationship of subsequent graphs. They can be detected by comparing the stripes containing the static graphs from left to right. These patterns are generic in the sense that different instances of these patterns can involve different static patterns and thus may look quite different.

Stability. The simplest dynamic pattern is the stability pattern—nothing visually changes in the subsequent static graphs, at least in the areas covered by the pattern. This means that the graph structure does not significantly change in this part. In our visualization technique, small changes affect the layout of the graph only moderately, which is important and does not go without saying in the area of dynamic graph visualization. For instance, animated node-link diagrams need to apply complex algorithms to reach the same stability in layout for small changes.

Trend. A monotonic change of edge density over a sequence of time steps is a trend. Such a trend may embody a change of structure or weights. Changes in the intensity of a line indicate changing edge weights of the respective set of edges. A trend in a particular point or small area indicates a systematic change in the edges that cross this area. This is of particular interest at the borders of the stripes, where such an area is directly related to the outgoing or incoming edges of the vertices. A trend may apply to a small set of vertices and edges or can be a global trend.

Periodicity. A sequence of recurring visual structures leads to periodicity in the visualization, for instance, the alternating presence and absence of a certain structure or the systematic variation of color values. Visual periodicity implies the analogous periodicity in the graph structure and edge weights. By presenting the complete sequence of graphs in one image, observing such periodicity is much simpler than in animated diagrams.

Shift. A shift pattern is described by a similar visual structure at a different vertical and horizontal position. This indicates a similar graph structure in a different part of the graph shifted in time. A problem is that similar structures may look different in other parts of the graph. For instance, an edge pointing to a vertex somewhere in the middle of the vertical axis is a descending line when the start node is in the upper part and an ascending line when it is in the lower part. Hence, we also have to look for distorted static patterns when searching for shifts.

Anomaly. In a sequence of similar visual patterns, a single graph that looks different with respect to some aspect is a visual anomaly. This individual property, for instance, could be the existence or absence of a certain static visual pattern. Anomalies reveal outliers with respect to the graph structure in the sequence of graphs.

Like static patterns, dynamic patterns could be hidden by more dominating patterns. The absence of a certain visual pattern again does not necessarily imply the absence of a certain property of the graph structure. However, highlighting static properties also helps us detect dynamic graph properties.
5 CASE STUDIES

The two case studies we present in the following apply our visualization technique to two different application scenarios. The applications use mid-size to large datasets to show the scalability of our approach. We make use of the visual patterns we introduced in the previous section. Moreover, the interactive features help us check assumptions and explore further phenomena. Hence, not all insights that we gained can be reproduced from just the few accompanying static figures.

5.1 Software Visualization

As a first application domain, we look at the visualization of software evolution where dynamic graph data was extracted from open source software systems and their development history. Existing visualization systems like CVSScan [35] or Evolution Matrix [24] do not typically show relational data. Exceptions are the GEVOL system [8] and more recently codeSwarm [27], which use animated node-link diagrams to visualize the evolution (codeSwarm actually does not draw the links).

When visualizing the artifacts contained in a software project, we often have to deal with large datasets. For example, mid-size software systems may already consist of hundreds of classes and thousands of methods. These artifacts change over time with each check-in of the software repository or with each release. In this first case study, we look at the methods of a system and how the methods interact by method calls in different releases of the software. Software developers might use this dataset to understand the system and its history, to learn when, and maybe also why, particular dependencies were introduced, or to identify design flaws like unwanted dependencies.

The JUnit regression testing framework (www.junit.org) serves as an example. We chose this project because it is a well-known open source software, its release history is readily available, it is small enough to be readable in the printed, non-interactive figures of this paper, and it is large enough to get an impression of the properties and scalability of our visualization. We collected the 21 available releases of JUnit as compiled Java bytecode and extracted the method call dependencies using DependencyFinder (depfind.sourceforge.net). The package and class structure provides a natural hierarchy on the methods of the system. The resulting dynamic graph consists of 2,817 vertices connected by 15,339 edges, where each edge is weighted by 1.

Figure 6 shows the dataset using our visualization technique. The upper part depicts all method call dependencies of the project, the part on the lower left represents the added dependencies (in each case with respect to the previous release), and the one on the lower right the removed dependencies. In the following, we describe a set of interesting phenomena and related visual patterns that we annotated in the figure as a matter of readability.

- The visualization provides an overview of the release history in a single image. Significant visual differences hint at major changes of the system (anomaly pattern). For JUnit, we easily retrieve the main steps of development, which only partly match to what we would have guessed analyzing the version numbers. For example, we observe major changes from version 3.0 to 3.2, but only few changes from version 2.1 to 3.0. Recurring visual patterns indicate the stable parts of the system (stability pattern). For instance, the junit.framework package changes only in small increments and decrements (A), as also confirmed by the diagrams for added and removed dependencies (A' and A'').
Analyzing method calls, the direction of the call relation carries very important information. By always drawing the edges from left to right, this information can be retrieved at first glance. We observe central methods that are called by many other methods (fan-in pattern) like in the junit.framework package (B1)—the interactive features, namely, collapsing hierarchy nodes, allow us to focus on this area and identify the corresponding methods, which are elements of class TestResults. Analogously, we find methods that call many other classes (fan-out pattern), for instance, in the junit.awtui package (B2).

Clusters in the call graph reveal highly cohesive sets of classes. If the classes of a cluster belong to the same package, this results in dense horizontal structures in the visualization (beam pattern) and may be interpreted as an indication of a good design according to the classical design principle of low coupling and high cohesion [31]. We can find these patterns, for instance, in the Assert class in the org.junit package (C1). In contrast, clusters that span several packages also create dense diagonal patterns (hour glass pattern) like some classes of the packages org.junit.internal.runners and org.junit.runner (C2). It might be worthwhile to consider them for restructuring.

Non-symmetric cross beam patterns, which connect a set of vertices with another set of vertices only in one direction, reveal a layered design where the classes and methods of one package access another package unidirectionally (D1). Considering the layers of the design, the packages in the lower layer do not have any outgoing dependencies leaving the package borders. Such a package is the org.hamcrest package, which seems to be an external library (D2).

Features are added to, and also sometimes removed from, the system. For instance, the Swing user interface of JUnit seems to be implemented in the junit.swingui package. As we learn from the gap patterns and the diagrams for added and removed edges, this package is introduced in version 3.2 (E1 and E1′) and again removed in version 4.0 (E2 and E2′). Possibly related to this, the junit.ui package is significantly extended in version 3.2 before being completely removed in the next version (anomaly pattern, E3). Perhaps, the user interface features were transferred to the junit.swingui package with a transitional step in version 3.2.

Trend patterns reveal incremental changes of the system. Small trends are harder to observe in the view that visualizes all dependencies because they only lead to minor color adjustments from step to step. Nevertheless, we see a steady growth of incoming call dependencies for some classes in the org.junit.runner package (F). The view on the added dependencies allows us to confirm this assumption (F′): more incoming call dependencies are added than deleted in the respective frame.

In practical application of this software visualization, knowing details like the name of a particular class or the set of dependencies belonging to a particular method is important. These details cannot be retrieved by analyzing Figure 6, but can be extracted by using the zooming and filtering features of the visualization tool.

5.2 DBLP Visualization

As a second application domain for our dynamic graph visualization, we visually explore the collection of data in the DBLP online bibliography [25] and show interesting insights for the evolution of related paper title words. As a data basis, we extracted 1,581,172 paper titles between the years 1950 and 2010, inclusively. Each paper title consists of a finite number of words. We consider each title as a transaction of words. Any two words in one transaction are related if they occur in the same paper title. Instead of generating complete cliques, we only allow an edge from words occurring earlier in the title to words occurring later and additionally, we allow self-edges. By doing this, we reduce visual clutter in the diagrams that stems from meaningless symmetries in the graphs carrying no extra information.

Formally, from an ordered sequence of words \( w_s := \{w_1, \ldots, w_m\} \), \( w_j \in \Sigma^* \) with \( \Sigma \) denoting a finite set of characters (the alphabet), we obtain the word graph \( G_{w_s} \):

\[
G_{w_s} := (V_{w_s}, E_{w_s}),
\]

where

\[
V_{w_s} := \{v \in \Sigma^* \mid v \in w_s\} \quad \text{and} \quad E_{w_s} := \{(w_i, w_j, \{1\}) \mid w_i, w_j \in w_s, i \leq j\}.
\]

Here, we introduce an edge \((w_i, w_j, \{1\})\) in the word graph if \( i \leq j \). Before generating the title word graphs, we free the list of words from filler words such as and, or, it, of,... All generated word graphs belonging to the same year are unified to one single word graph.

The order of the nodes is given by the occurrence of a word in the prefix tree. Each word \( w \) consists of a finite sequence of characters

\[
w := (\sigma_1, \ldots, \sigma_{|w|}, \sigma_{|w|+1}), \quad \sigma_i \in \Sigma
\]

where \(|w|\) denotes the length of the word \( w \) and \( l_{\text{max}} \) denotes the length of the longest word in the list of words.

We map each \( \sigma_i \) to a unique number maintaining the lexicographic order of the characters by the bijective function \( \text{val} : \Sigma \to N_{[0,|\Sigma|-1]} \). Each word \( w \) is mapped to a unique identifier

\[
id_w := \sum_{i=l_{\text{max}}-|w|+1}^{l_{\text{max}}} \text{val}(\sigma_i) \cdot |\Sigma|^i
\]
The computation of these identifiers guarantees that words that are close to each other in the prefix tree are also mapped close to each other on the vertical line.

In our first scenario, we are interested in the evolution of the relation between the word fragments “vis” and “web” and all word relations to either one or both of these words. For this reason, we extracted all paper titles containing at least one of these two words. Since we take all words of a paper title into account in which one of the selected words occurs, many other words are involved that cause additional edges. To reduce the data flood and visual clutter in the display, we only take edges into account with a minimum weight of two. After this step, we still have to display 265,733 edges in total.

Figure 7 (a) shows this dynamic relational data with parallel edge splatting. The first observation is that the plot is becoming denser and denser to the right hand side: the number of publications is growing to higher and higher rates. One can easily see that the first occurrence of relations with respect to the word fragments “vis” and “web” with a minimum weight of two occurs in the year 1968. The relation is a self-edge and belongs to the word “visual”. Around the year 1990, relations are becoming denser, starting at the vertical position of the word “visual”. The occurrence of the word “web” starts some years later in 1996 and the first co-relations to the word “visual” are in 1999 and become quite frequent in the following years. The Z-pattern right below the word “web” is due to the fact that there are many paper title containing the word sequence “world wide web”. There are only a few relations to words starting with characters “x”, “y”, and “z”, which is a natural language characteristic phenomenon.

To further reduce visual clutter and make the finer substructures more visible, we concentrate on the 21 graphs of the years 1990 until 2010. Furthermore, we filter out all edges with a weight smaller than six. Figure 8 shows the result after these operations. Only 1,422 edges are filtered out, whereas a total of 264,311 edges remain in the diagram. The 21 graphs can now be visualized in a larger display space, leading to more substructures to be visible.

For example, one visual pattern is the horizontal line in the center that is becoming stronger over the years. This word occurs very often in the context of the word “vis” but not the word “web”, which can be uncovered by inspecting the cross beam pattern there. A detail-on-demand request shows that “image” is the word that has a strong relationship to “visual”. The word “web” is strongly coupled to the word “technology”, which can be seen by inspecting the cross beam pattern starting at the vertical position for the word “web”. More insights can be won when playing with the interactive features of the visualization tool, too many to illustrate them all here.

In our second scenario, we are interested in the word “graph”, which has a longer history than “vis” and “web”, see Figure 7 (b). This figure shows the same overall density behavior as Figure 7 (a), especially the growth in number of related publications. The major difference to the diagram in Figure 7 (a) is the fact that relations to other words start much earlier—in the years 1965 until 1969. Consequently, the graph sequence contains 438,538 edges in total. Again, we reduced the amount of relational data by applying an edge weight filter. Only edges with a weight of at least two are taken into account.

Also here, we apply further filtering (minimum weight 6 and time span from 1990 until 2010) and obtain a sparser dynamic graph dataset. From the 438,538 edges in the beginning, 412,416 remain, which is a loss of 26,122 edges in total by filtering. Figure 9 represents the resulting image of the filtered dataset. Compared to Figure 8, this dynamic graph is much denser and the occurring relations start relatively earlier. There are two cross beam patterns visible, one starts from “computer” and points to the word “graphics” and the second one starts at the word “hyper” and targets at “graph”. Both cross beam relation patterns show a growing trend.

6 Conclusion

In this paper, we have introduced parallel edge splatting, a scalable visualization technique for dynamic graphs. The approach depicts the dynamic graph in a single image that looks similar to a parallel-coordinates plot. Two case studies have shown how the visualization technique can be applied to explore large datasets in practical applications, based on the detection of visual patterns. The next step would
be to evaluate the effectiveness and usability of the visualization technique in a user study.

Our approach provides a better overview of the time dimension than animated diagrams. Due to the static diagram, the user does not have to preserve a mental map of the graph from step to step, which leads to a reduction of cognitive efforts. Furthermore, laying out the graph does not require complex algorithms and graphs can be added to the sequence on-the-fly without changing the global layout.

Moreover, the technique is more scalable than other single-image dynamic graph visualization techniques. First, the graph layout is defined by arranging the vertices of each graph on vertical lines and connecting the vertices from left to right. Second, the visualization transforms the edges to a scalar field of edge densities.

The need to place the vertices on a 1D vertical line is owed to the limited space available when visualizing a dynamic graph in a single image. Simply connecting these vertices by arcs is not very scalable because the arcs frequently cross in small angles so that it becomes difficult to follow single edges [14]. Splitting each vertex and drawing the edges from left to right significantly mitigates the problem of edge crossings: although the number of crossings is not reduced, it is easier to follow edges because they cross at larger angles and do not consist of curved lines. Even if parts of the edge are not discernible, a fragment of the edge identifies the two connected vertices based on the gradient and position of the fragment. Still, edge crossings may confuse the user. Using a hierarchical structure is a first step towards ordering the vertices so that related vertices are placed next to each other on the vertical axis, where sorting based on a natural hierarchy is a good basis to reveal visual patterns. If no natural hierarchy is available, a hierarchical clustering algorithm could be employed to retrieve a meaningful structure. Beyond that, more sophisticated sorting strategies may also optimize the order of the hierarchy vertices [14]. Without a hierarchy, the reduction of edge crossings could be used as the exclusive ordering criterion.

Edge splatting addresses the problem of visual clutter that occurs when drawing many lines onto a small area of screen space. Another approach would have been to bundle the edges according to the hierarchical structure of the vertices [18] or according to a force-directed placement [20], which can also be applied to vertically arranged vertices [19]. Telea and Ersoy use a combination of distance-based splatting and skeletonization to generate image-based edge bundles [32]. We, however, did not employ bundling because it somewhat obfuscates the information of an edge—seeing only a part of an edge would not allow a user to recognize the start and end point of the edge because the edge is represented as a curved line. Furthermore, density splatting is important to summarize a set of vertices in a single pixel on the vertical axis, which is necessary when the number of vertices exceeds the height of the visualization in number of pixels.

Parallel edge splatting serves the purpose of revealing changes in the graph structure, i.e., exploring the evolution of a graph. Thus, it is not important if strong relations are caused by few strong weighted connections or several more slightly weighted connections. Hence, the accumulation of weights during the splatting of edges does not really involve a loss of important information.

In general, it would be possible to apply the splatting algorithm to graphs laid out in 2D and juxtapose these diagrams to visualize the dynamic sequence. Our arguments in favor of the 1D layout of the graph vertices are that the representation of a graph becomes space-efficient, ambiguities caused by edges crossing vertices are avoided, the alignment leads to a better comparability of graphs throughout the sequence, and an additional hierarchy can easily be added.

All in all, the visualization technique provides overview because it depicts the complete dynamic graph in a single view and visual patterns reveal the graph structure—details can be retrieved using interactions. Furthermore, it is scalable because of the special 1D vertex layout and the edge splatting technique.

ACKNOWLEDGMENTS

The project was in part funded by the German Research Foundation (DFG) grant DFG WE 2836/4-1.
REFERENCES


