SIGRAD 2014
M. Obaid, D. Sjölie, E. Sintorn and M. Fjeld (Editors)

TopoLayout-DG: A Topological Feature-Based Framework for Visualizing Inside Behavior of Large Directed Graphs

Ragaad AlTarawneh, Max Langbein, Shah Rukh Humayoun, Hans Hagen

Computer Graphics and HCI Group
University of Kaiserslautern, Germany
{tarawneh, m_langbe, humayoun, hagen}@cs.uni-kl.de

Abstract

Directed graphs are a useful model of many computational systems including software, hardware, fault trees, and of course the Internet. We present the TopoLayout-DG framework, an extension to the original TopoLayout algorithm, for visualizing the inside behaviour of large directed graphs. The proposed framework consists of: a feature-based multi-level algorithm, called ToF2DG*, that detects topological features in large directed graphs in a hierarchical fashion; and visualization methods for the resulting levels of details of the graph's topological structure. In this work-in-progress paper, we highlight the main steps of the proposed ToF2DG algorithm. Moreover, we show some preliminary visual representations of artificial directed graphs. These preliminary representations indicate that the framework promises to solve some scalability issues in the visualisation of large directed graphs.*

Categories and Subject Descriptors (according to ACM CCS): G.2.2 [Mathematics of Computing]: Graph Theory—Graph algorithms I.3.8 [Computing Methodologies]: Computer Graphics—Applications

1. Introduction

Producing meaningful abstract representations are crucial in the case of large data and limited display sizes. One of the common means to express it is through a multi-level representation of the original graph. This assumes providing a hierarchical structure of the original graph. This hierarchical structure can be modelled using the containment relationships between the graph components, which helps in keeping the adjacency relations between the graph nodes. In this work, we present a multi-level framework, called **TopoLayout-DG** (**TopoLayout** for **D**irected **G**raphs), for visualising large directed graphs and to try overcoming the scalability and visualisation quality issues in these graphs. The proposed TopoLayout-DG framework aims at helping in analysing the topological structures of directed graphs based on topological features of their sub-graphs.

Our solution is based on extending the TopoLayout algorithm, initially introduced by Archambault et al. in [AMA07], for visualising large undirected graphs with less edge-crossings. In this extension, we design the detection phase to take the direction of the arc into account. The main difference between the original TopoLayout algorithm and our extension is the graph type. Such that, the original TopoLayout framework was dedicated to visualise general undirected graphs, while in our case we extend it to take care of the edge direction between the nodes. In order to detect different topological features inside the graph we then create another level of the abstraction of the original graph. We concern ourselves in finding interesting topological features like strongly connected components (e.g., cycles or complete graphs), trees, and DAGs (Directed Acyclic Graphs). Moreover, we use the graph-drawing algorithms already proposed in the literature (e.g., the tree layout algorithms [Ead92], the force directed layout algorithm [Ead84]) to visualise the detected features. Our approach is based on subdividing the underlying large directed graph into a set of sub-graphs with regard to the topological features of the local sub-graphs. Then we visualise each sub-graph using one of the algorithms that is tuned to its topological feature. Once the main graph has been decomposed into the basic features that form its original structure, we compose them into a set of MetaNodes. This composition is done in a fashion that each set of nodes which are connected with each others through a specific feature, e.g. a tree-structure, are collapsed into one node called the "MetaNode". We do this for all the interesting and required topological features in the underlying main graph. The resulting representation provides an abstract hierarchi-

cal multi-level view of the whole graph in a 2D representation, which helps in overcoming the size and visual quality limitation. It also offers a new strategy to cluster large directed graphs according to their inside topological features.

The remainder of the paper is structured as follows: First we highlight briefly the related work (Sec. 2). Then we provide the framework pipeline (Sec. 3), explain our ToF2DG algorithm (Sec. 4), and describe the visualisation of the final representation (Sec. 5). Finally, we conclude (Sec. 6).

2. Related Work

A directed graph G can be defined as a set of nodes V, where these vertices are connected with each other via a set of edges $A \subset V \times V$. The presented work focuses on providing a 2D visual presentation of the directed graph G. Until now, only a few algorithms have been proposed to visualise directed graphs. The Sugiyama algorithm is one of the first algorithms that draws general directed graphs [STT81]. It works in two phases: First, it layers the graph nodes, which means assigning a layer for each node and then placing all nodes to the corresponding layers; Secondly, it reduces edges crossings and nodes overlapping. Many of the suggested Sugiyama algorithm's steps are proved as NP-hard [GJ83] while some of these steps were proved as NP-Complete [EW94].

In [KT13], authors presented DAGView framework that aimed at visualising directed acyclic graphs more clearly. They take into account the users' preferences during the layout phase such as users can control the size of the underlying grid and the crossings that appear in the final layout. H3Viewer is one of the tools that visualises large directed graphs (semi trees) in 3D spaces [Mun97]. Technically, this tool can be used for visualizing large graphs. However, visual cluttering and extra occlusion makes it difficult to read or extract information from the generated visualisation. To tackle this problem, many clustering algorithms have been proposed in order to provide a multi-level visualisation of large graphs [RS97]. Additionally, edge-clustering and edge-bundling techniques have also been proposed to reduce the cluttering issue in the final representation [Hol06]. Our work is the first try towards providing a feature-based multi-level visualisation technique for large directed graphs.

3. The Approach

The TopoLayout-DG pipeline consists of four main phases (see Fig. 1), based on the original TopoLayout algorithm [AMA07]:

1. *The Detection Phase:* This is similar to the coarsening operation in multi-level techniques. In this phase, the feature hierarchy is recursively created through our algorithm by identifying the feature type of each sub-graph (see Sec. 4).
2. *The Visualisation Phase:* The detected topological feature for each sub-graph is drawn using a layout algorithm tuned to its type.
3. *The Crossing Reduction Phase:* It aims at reducing the crossings between edges.
4. *The Overlapping Elimination Phase:* The phase reduces the overlapping between nodes in the final graph.

The TopoLayout-DG differs in working from the original TopoLayout in the first two phases. In the forthcoming sections, we focus only on these first two phases, as phases three and four are out of the scope of this work. TopoLayout-DG uses the Libgraph package [Hei11] and the OpenGL library for the realisation of detection and visualisation phases respectively. To handle the third and fourth phases of the pipeline, we use the approximate solutions provided by [AMA07].

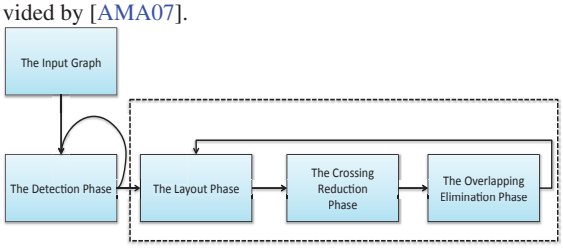

Figure 1: *The four main phases in TopoLayout-DG pipeline*

4. The ToF2DG Algorithm

We provide an algorithm, called **ToF2DG** (**To**pological **F**eatures **D**etector for **D**irected **G**raphs), for performing the topological feature-based detection during the *detection* phase of TopoLayout-DG pipeline. ToF2DG detects the set of topological features in the input graph, where each topological feature is then collapsed into one node called the MetaNode. This process is applied recursively over the graph to build the required hierarchy of the graph. Fig. 2 provides ToF2DG's pseudo code. In the following, the term *node* means the actual node in the graph while the term *MetaNode* means a group of nodes (either actual graph nodes or another set of MetaNodes), which together belong to one of the considered topological features. For keeping the original graph connectivity, TopoLayout-DG creates a *MetaEdge* to represent the initial set of edges between those nodes that belong to different MetaNodes. Below we provide the details of the algorithm.

First of all, ToF2DG finds the number of connected components in the graph G using the *depth-first-search* algorithm. This information is stored in a list C. The step of detecting the connected components is performed only once at the beginning. Each connected component is represented as a MetaNode in the final representation. For each connected component in C, ToF2DG performs the following steps: First, it measures the sub-graph size. If it is 1 then this sub-graph is detected as an isolated node and the ISOLATED

```
Procedure ToF2DG
Input: Direct Graph G
Output: Hierarchy container to describe the highest level of details
C = List of unconnected sub-graphs in G
for each sub-graph S in C do
    if sizeOf(S) == 1 then
        create y=MetaNode(S); mark y as ISOLATED;
    else
        for x in all SCC's in S do
            create y=MetaNode(x);
            if (x.edges () == x.vertices())
            mark y as CYCLE;
            else if (x.edges() > 2*x.vertices())
            mark y as COMPLETE;
            else mark y as SCC;
            create MetaEdges e⃗ to replace the connecttions of y to the other
            nodes;
            if y is SCC and |y| > limit then
                Bisect y into Metanodes u and v;create a Metaedge
                between them containing the cut edges;
                TOF2DG(u);Hierarchy.push(u);
                TOF2DG(v);Hierarchy.push(v);
            end
            Hierarchy.push(y);
            S.edges.push(e⃗);
            S.edges.remove(edges contained in e⃗);
        end
        while x = Extract.Trees(S) do
            create y = MetaNode(x);Mark y as TREE;
            Hierarchy.Push(y);
            S.Remove(x.nodes except x.root);
            Assign y to x.root;
        end
        while x = Extract.DAGs(S) do
            create y = MetaNode(x);Mark y as DAG;
            Hierarchy.Push(y);
            S.remove(x.nodes except x.root);
            Assign y to x.root;
        end
        ToF2DG(S)
    end
end
```

Figure 2: *The ToF2DG pseudo code.*

feature is assigned to it. ToF2DG creates a MetaNode to represent it in the final view (see Section 5). If the size is greater than 1, ToF2DG finds the Strongly Connected Components (SCCs) using the Tarjan algorithm [Tar72]. For every SCC, ToF2DG checks if it is a normal cycle or a complete directed graph based on the number of edges in this SCC, using an approximate solution similar to the proposed one by [AMA07]. In both cases, it creates a MetaNode to represent the nodes in the underlying SCC and assigns the CYCLE or COMPLETE label accordingly. This step is performed iteratively until all SCC features are detected and collapsed into MetaNodes.

For SCCs which are neither CYCLE nor COMPLETE, the following bisection is proposed, which is a compromise between a minimum number of cut edges and an even bisection: a node is chosen at random and the other nodes are classified according to the distance (i.e., the number of edges) from that node. We denote these node classes as D_e with e = distance. We define $A_j := D_0 \cup \ldots \cup D_j$, $B_j := D_{j+1} \cup \ldots \cup D_n$, $\alpha(e) :=$ the number of edges between nodes in A to nodes in B, $n :=$ the maximum distance. Then the Metanode is bisected into A_j and B_j with $j = \max_i\{\Big(\min\{|A_j|, |B_j|\}/\alpha(i),\ i\Big)\}$ (see Fig. 3).

Then the MetaNodes are created for them and a MetaEdge to connect them is also created. After it, ToF2DG is applied on these MetaNodes again. All MetaNodes are then pushed in a special data structure, called *Hierarchy*, to represent the next level of detail of the input graph. This *Hierarchy* container stores all the required information about the original nodes and their corresponding MetaNodes. After storing the required information in this container, ToF2DG removes all the original nodes, except the roots, from the original graph, as now a MetaNode is there to represent them in the graph next view. This helps in reducing the time complexity during the detection phase.

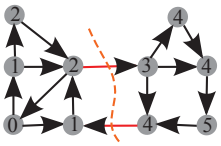

Figure 3: *Bisection of an SCC into an SCC (left) and a DAG (right). The numbers are the distance from node 0.*

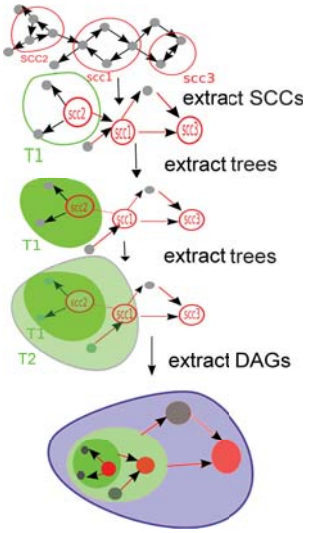

Figure 4: *An example of topology extraction. Colours indicate the feature type while the nested nodes show the hierarchy of the graph*

After the above step, ToF2DG starts detecting the acyclic features in each particular connected component. ToF2DG finds the tree structures by searching all leaves in the subgraph (with in-degree = 0 or out-degree = 0). For this, it starts from one leaf and then expands upward to the leaf parents until the root of the tree is found. Then it creates a MetaNode to represent this set of nodes and assigns a TREE feature to this MetaNode. ToF2DG detects all Tree-structures in two directions: from top to bottom (called *DownTree*) in which all edges are directed away from the root to leaves, and in bottom-up style (called *UpTree*) in which directions of the edges go from leaves to the root. ToF2DG performs this iteratively until there are no more trees in the sub-graph.

For detecting a DAG structure, ToF2DG starts from those leave nodes that have two parents. A DAG can have more than one parent compared to only one in a tree. ToF2DG performs the DAG detection steps in the same manner as the

tree detection steps. It distinguishes in this structure between the downward direction and the upward direction. Here the root nodes are not removed from the graph because they keep the graph connectivity.

The result of ToF2DG is that each node in the original graph is associated with a topological feature attached to its MetaNode. The set of MetaNodes represents the graph hierarchy. It is used to produce the higher level of detail for the same graph resulting in a *MetaTree*. As in the original TopoLayout algorithm, the graph hierarchy represents the different levels of detail such that level i is a parent of level $i+1$, where each level is a different abstraction of the graph. In order to create a multi-level representation of the graph, ToF2DG performs the same mentioned steps over the list of MetaNodes to find how these nodes are topologically connected with each other. In this step, a MetaGraph is created to represent the next higher level of details. This step is repeated until the targeted graph is abstracted into one single node, which represents the highest and the most abstracted level of details of this sub-graph. The process of feature extraction is illustrated in Figure 4.

5. The Layout Phase

Due to the application of ToF2DG, each vertex in the original graph now refers to a topological feature and belongs to a MetaNode which is passed to the visualisation phase. First the topological feature of each MetaNode is checked and then it is passed to the corresponding layout algorithm tuned to the MetaNode's topological feature. Then it is visualised according to one of the following cases:

- *The ISOLATED Structure:* In this case, a point is displayed on the screen to represent the MetaNode.
- *The TREE Structure:* If the topological feature is a tree then the radial layout algorithm is called. In this, internal nodes of the underlying MetaNode are displayed as a red colour radial tree. Because we deal with relatively large graphs, we selected the radial layout [Ead92] to utilize screen space efficiently.
- *The DAG Structure:* If internal nodes of the MetaNode are connected as a DAG then the force directed layout [Ead84] is used to visualise it. However, we are planning to use the Sugiyama algorithm [STT81] in future as it is more convenient for DAGs or drawing layered graphs.
- *The COMPLETE/CYCLIC Structure:* In the case of complete or cyclic topological feature, the circular layout algorithm is used to visualise the nodes.

TopoLayout-DG framework creates multilevel views of the original graph in a manner that different views of the graph are related somehow to each other in a multi-level representation. The edge direction between any two nodes is shown using the colour interpolation from red (source) to green (target). Through this, each cell represents a MetaNode while the included graph provides the lower level of detail. In future, we aim at relating the multilevel of the structure without losing the global context.

6. Conclusion

In this work-in-progress paper, we presented the TopoLayout-DG framework to handle directed graphs. The framework offers ToF2DG algorithm for detecting the topological features in large directed graphs. In the future, we intend to provide practical results using large directed graphs. The proposed framework can improve the visualisation of compound graphs naturally, as it provides a better understanding of the structural relations between the graph nodes using the multi-level layout technique. This solves some scalability issues in the final layout of large graphs. For example, visualising large software systems according to the topological features between the different system elements in a multilevel fashion can help software architects in understanding the behavioural pattern of the system. However, one important aspect to be investigated is the time complexity analysis of the framework. We also intend to analyse the time complexity using different theoretical and practical methods while testing TopoLayout-DG over large directed graphs in real situations.

References

[AMA07] ARCHAMBAULT D., MUNZNER T., AUBER D.: Topolayout: Multilevel graph layout by topological features. *IEEE TVCG 13/2* (2007), 305–317. 1, 2, 3

[Ead84] EADES P.: A heuristic for graph drawing. *Congressus Numerantium* (1984), 149–160. 1, 4

[Ead92] EADES P.: Drawing free trees. *Bulletin of the Institute for Combinatorial and its Applications 5(2)* (1992), 10–36. 1, 4

[EW94] EADES P., WHITESIDES S.: Drawing graphs in two layers. *Theoretical Computer Science 131* (1994), 361–374. 2

[GJ83] GAREY M. R., JOHNSON D. S.: Crossing number is np-complete. *SIAM Journal on Algebraic and Discrete Methods* (1983). 2

[Hei11] HEINE C.: Libgraph, November 2011. URL: http://www.informatik.uni-leipzig.de/~hg/libgraph/. 2

[Hol06] HOLTEN D.: Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. *IEEE TVCG.* (2006). 2

[KT13] KORNAROPOULOS E. M., TOLLIS I. G.: DAGView: An approach for visualizing large graphs. In *Proceedings of the 20th International Conference on Graph Drawing* (Berlin, Heidelberg, 2013), GD'12, Springer-Verlag, pp. 499–510. 2

[Mun97] MUNZNER T.: H3: laying out large directed graphs in 3d hyperbolic space. *Proceedings of the IEEE Symposium on Information Visualization* (1997). 2

[RS97] ROXBOROUGH T., SEN A.: Graph clustering using multiway ratio cut. *Proceedings of Graph Drawing, Lecture Notes on Computer Science 1353* (1997). 2

[STT81] SUGIYAMA K., TAGAWA S., TODA M.: Methods for visual understanding of hierarchical system structures. *IEEE Transactions On Systems Man And Cybernetics* (1981). 2, 4

[Tar72] TARJAN R. E.: Depth-first search and linear graph algorithms. *SIAM J. Comput. 1*, 2 (1972), 146–160. 3