Smooth Animation of Structure Evolution in Time-Varying Graphs with Pattern Matching

Yunzhe Wang
Department of Computing
The Hong Kong Polytechnic University
csyzwang@comp.polyu.edu.hk

George Baciu*
Department of Computing
The Hong Kong Polytechnic University
csgeorge@polyu.edu.hk

Chenhui Li
Department of Computing
The Hong Kong Polytechnic University
cscli@comp.polyu.edu.hk

ABSTRACT

Drawing a large graph into the limited display space often raises visual clutter and overlapping problems. The complex structure hinders the exploration of significant patterns of connections. For time-varying graphs, it is difficult to reveal the evolution of structures. In this paper, we group nodes and links into partitions, where objects within a partition are more closely related. Besides, partitions maintain stable across time steps. The complex structure of a partition is simplified by mapping to a pattern and the evolution is exposed by comparing patterns of two consecutive time steps. We created various visual designs to present different scenarios of changes. In order to achieve a smooth animation of time-varying graphs, we extract the graph layout at each time step from a *super-layout* which is based on the *super-graph* and *super-community*. The effectiveness of our approach is verified with two datasets, one is a synthetic dataset, and the other is the DBLP dataset.

CCS CONCEPTS

• **Human-centered computing** → **Visualization**; **Graph drawings**;

KEYWORDS

time-varying, graph visualization, simplification, structure pattern

1 INTRODUCTION

Complex time-varying graphs usually contain tremendous entities and keep changing structures over time. It is difficult to gain significant insights into the structure and its evolution. In visualization, researchers tend to aggregate graph elements spatially or temporally. On the spatial dimension, hierarchical methods [Noel and Jajodia 2004] collapse subgraphs to new single vertices. While in temporal space, the granularity of time intervals is enlarged [Von Landesberger et al. 2016]. After aggregation, the size of visual elements that need to be presented is reduced. Also, users can have a high-level understanding of graph structures. In this paper, we facilitate the exploration of the structure evolution in time-varying graphs by dividing them into stable divisions.

Community detection methods divide graphs by clustering more densely connected vertices. Relevant studies have been fully developed in static graphs [Blondel et al. 2008; Girvan and Newman 2002]. However, it remains challenging for time-varying graphs. Because it would be very time-consuming if applying the detection iteratively. In addition, we need to match corresponding communities across time steps [He et al. 2017]. To obtain divisions effectively, we propose to conduct community detection on the super-graph at a global level. Graphs at individual time steps are divided according to the detection result. By doing so, we can achieve high consistency and time efficiency and the animation outstands in preserving users' mental map [Diehl et al. 2001].

Analysing graph structure is an important research area. We try to reveal the structure evolution by investgating into partitions of graphs. Sometimes, vertices get connected by obeying a certain pattern. For example, in an egocentric network, an entity occupies the leading place and others connect to it. Li et. al [2015] designed several structure patterns and mapped a graph to the most resemblant one. Their work aims at exploring local patterns in massive static graphs. Though the matched pattern only approximates the actual structure, the matching procedure runs time-efficiently and the visualization provides support for discerning a pattern of interest. Inspired by their work, we implemented a similar idea to time-varying graphs. The objective is to represent the structure evolution with compact visual expressions. At the same time, we exhibit significant structure patterns for users to explore. Patterns are defined based on topologies that are prevalent in graph analysis. We name the patterns *chain*, *loop*, *clique*, and *egocentric*. The similarity between graphs and patterns is measured by either a generic or customised algorithm. In this way, a most suited pattern is assigned to the graph. Given the four patterns we designed, the customised algorithm outperforms the generic one.

Taking time-efficiency and stability of visualization into consideration, we obtain the super-community by carrying out detection on the super-graph. Communities are regarded as meta-nodes and the force-directed layout method yields their positions on the display. We use animation that collaborates with node-link diagrams for visualization. Instead of detecting communities and computing the layout iteratively, we extract partitions and layouts of each time step from the super-community. Meanwhile, the smoothness of

*George Baciu is the corresponding author.

animation is well achieved. Through a succinct visual design, users can identify both global and local changes of the graph. Generally, contributions of our work reflect in the following aspects:

- First of all, we define four structure patterns which are portraits of the underlying topology of graphs. Accordingly, we introduce two matching methods. With the generic method, users can flexibly append new patterns into the system.
- Successively, we provide compact visualization of the structure evolution. Users can compare and observe the changes between consecutive time steps. We use intuitional visual encodings to display diverse changes so that users can immediately distinguish them.
- We simplify the complex structure by dividing graphs into partitions. Partitions and layouts at each time steps are subsets of the super-community and the super-layout. Hence, our system involves less computation and abrupt changes can be avoided in animation.

2 RELATED WORK

2.1 Visualization of Time-varying Graphs

The primary goal in drawing time-varying graphs is to keep readers′ mental map, which means the graph layout should not change abruptly. To decide the layout of static graphs, force-directed approaches are widely used. An incremental approach by Frishman et al. [2004] obtained an initial layout by the force-directed algorithm. Then it adjusted the layout when nodes are added or removed. This approach proves to maintain a good visual stability.

There are mainly two ways to represent time in visulization, one is animation and the other is timeline. Though timelines enable users to compare arbitrary time steps, they are also criticized for the bad scalability [Brehmer et al. 2016]. Animations map the time stamps of a graph sequence into visualization time. Bach et al. [2014] designed a visualization system called GraphDiaries for identifying changes happened in animated node-link diagrams. If nodes pertinent to visual tasks are highlighted through the full time series of the graph, then a more stable drawing provides little benefit. Feng et al. [2012] achieved smooth animation by firstly generating the initial graph layout of each time step from a super-graph [Diehl et al. 2001]. Then they optimized the layout by fulfilling a few constraints. Compared to timelines, animations convey more information by embodying an unlimited number of time steps. Besides, in animation, graph of one time step takes full use of the display, while in timelines, multiple graphs share the display.

2.2 Visualize Group Structures

Assigning nodes or links into groups simplifies the general analysis of large graphs. Objects within a group share similarities regarding attributes or topology [Vehlow et al. 2017]. *Community Detection* approaches aim at dividing nodes into communities so that intra-community links are much denser than inter-community ones. Such approaches involve a high computational cost, because the number of communities is unknown and they may have different structures and densities. The Girvan–Newman algorithm [Girvan and Newman 2002] is widely used for its reasonable results. It is implemented by removing links of high-betweenness centrality [Brandes 2001] in each iteration and finally taking remaining groups of nodes as communities. However, this method fails to handle large-scaled graphs. The Louvain method [Blondel et al. 2008] is based on modularity maximization. Modularity is a scalar value which measures the density of links inside communities as compared to links between communities. This method has succeeded in managing graphs of sizes up to 100 million nodes and billions of links.

Vehlow et al. [2015] visualized the evolution of communities by presenting the dynamic graph and community structures in a single image. Consequently, users are supported to identify the relationship between community evolution and topology changes of the graph. In their design, a graph is depicted as a stack of ordered rectangles, each denoting a community. The topology of a community is drawn in the rectangle so that users can flexibly track the connection of a specific node. However, due to the limit of the display space, not too many time steps can be presented.

Sometimes, users seek for specific graph topologies. Dunne and Shneiderman [2013] defined three different patterns called fan, connector and clique respectively. They search for these patterns and represent them with glyphs. The graph representation is largely simplified for visual exploration. Different from their work, we assign an approximate pattern to a partition, rather than exhaustively search the patterns in the whole graph.

3 PROBLEM STATEMENT

First of all, we give definitions of the time-varying graph, as well as several pertinent terminologies.

Definition 3.1. ***Time-Varying Graph:*** A time-varying graph can be defined as a sequence of ordered static graphs, $G_T = \{G_1, G_2, \ldots, G_t\}$, where t is the number of all time steps. At time step i ($1 \le i \le t$), $G_i = (V_i, E_i)$, in which V_i is a set of nodes and E_i is a set of links between nodes, $E_i \subseteq V_i \times V_i$. At any time step, nodes and links might be added or removed.

Definition 3.2. ***Super-graph:*** This is a concept introduced by Diehl et al. [2001]. Suppose $G_S = (V_S, E_S)$ is the *super-graph* of a time-varying graph G_T. $V_S = V_1 \cup V_2 \cup \ldots \cup V_t$ and $E_S = E_1 \cup E_2 \cup \ldots \cup E_t$.

Definition 3.3. ***Super-community:*** We conduct community detection on the super-graph. The set of communities detected is called *super-community* which is denoted as $C_S = \{com_1, com_2, \ldots, com_k\}$, k is the number of communites.

Definition 3.4. ***Graph partition:*** We decompose each G_i into partitions so that $G_i = P_{i1} \cup P_{i2} \cup \ldots \cup P_{ik}$, and nodes in P_{ij} are defined as $\left\{v | v \in V_i, v \in com_j\right\}$, $1 \le j \le k$. Partitions P_{ia} and P_{ib} are connected if $\{\exists xy | x \in P_{ia}, y \in P_{ib}, xy \in E_i\}$. Some partitions might be empty, hence the number of partitions is no larger than k.

Definition 3.5. ***Super-layout:*** $L_S = L_1 \cup L_2 \cup \ldots \cup L_t$. L_S is the force-directed layout of $\{com_1, com_2, \ldots, com_k\}$ if each community is represented by a meta-node. L_i is the layout of non-empty partitions at time i and the position of P_{ij} and com_j are the same.

4 METHOD

In this section, we explain the details of our method. As shown in Figure 1, our work follows a general routine. Above all, we need

to decide graph partitions at each time step. This is achieved by community detection on the super-graph. It is worth noting that during the whole process, detection only needs to be executed once. Based on Definition 3.4, partitions of G_i are $P_{i1}, P_{i2}, \ldots, P_{ik}$. Moreover, the graph layout at all time steps is decided by the super-layout which is the force-directed layout of meta-nodes denoting communities in super-graph. The community detection approach we adopted is the Louvain algorithm [Blondel et al. 2008], and the force-directed layout method is the one proposed by Dwyer [Dwyer 2009]. These two methods run quite fast and possess good scalability. We implement the simplification of structures by mapping them to corresponding patterns. Hence, visual comparability can be achieved.

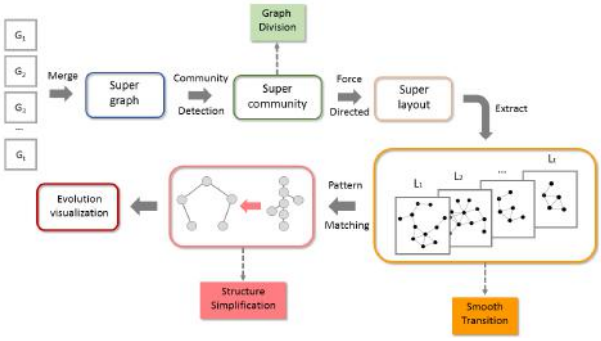


Figure 1: Routine of our work. The input is a time-varying graph which consists of a sequence of static graphs. Arrows indicate the actions we take in order. Rectangles give the results of actions. Nodes in $L_1, L_2, \ldots, L_t$ represent partitions.

4.1 Pattern Design

Patterns imply the topology types of graphs. With a pattern, we can easily tell how entities are connected (e. g., the connections are sparse or dense). The portrait of a pattern is very simple, similar to a skeleton of the graph. But it reflects fundamental topological properties. We design patterns according to a few graph strucures that researchers might concern about.

Particularly, we define four types of patterns, as shown in Figure 2: (a) *chain*: nodes connect to each other linearly, but the head and tail do not connect. (b) *loop*: nodes connect to each other one by one and constitute a closed circle. (c) *clique*: all pairs of nodes are fully connected. (d) *egocentric*: one node locates in the center and all others connect to it.

Nodes in the *chain* pattern connect in a sequence, which is likely to reflect the order information. The *loop* pattern contains a closed path and it also has sparse density. Conversely, nodes are densely connected in the *clique* pattern. Besides, the degree distribution in a *clique* is relatively flat. The *egocentric* network has been extensively explored [Heer and Boyd 2005]. One node dominates the whole network and it connects to all other nodes. This kind of pattern may lead us to some important nodes. In consideration of compactness and aesthetics, we use five nodes as the basis of patterns. Provided that three nodes are used, we cannot distinguish the *loop* and *clique* patterns. Given four nodes, they cannot illustrate dense connections

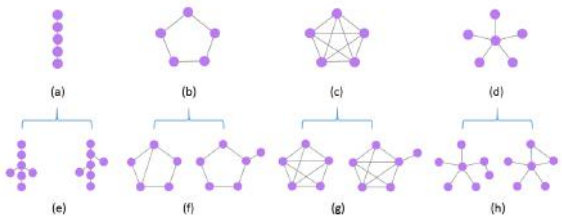


Figure 2: The first row lists the four patterns: (a) *chain*, (b) *loop*, (c) *clique* and (d) *egocentric*. (e), (f), (g) and (h) display variants of the four patterns respectively.

as well as five nodes do. Thus, the minimal requirement is five nodes. More nodes are acceptable, but too many nodes may raise visual clutter problems.

Patterns are designed for the sake of finding topologies of interest. Users are allowed to define new patterns and integrate them into the system. For instance, Figure 2 also lists some variants of the four patterns. Any one of them can be a substitute for the corresponding pattern.

4.2 Pattern Matching

In the literature of sociology, the morphology of networks vary a lot. Usually, they do not appear in a form as inerratic as that of patterns in Figure 2. Thus, we need to analyze the networks and map them to the most resemblant pattern. In this section, we introduce two matching methods, one is generic, and the other is customised. Both methods are adaptive to the network size. Note that though we use five nodes to illustrate patterns, it does not mean that the matched network has a small size.

The first method is based on the Netsimile algorithm [Berlingerio et al. 2012]. This method is generic and is used when users add new patterns. The second approach is customised only for patterns defined in subsection 4.1. It achieves higher accuracy in the context of this paper.

The generic method returns a similarity score ($score \in [0, 1]$, 0 means totally different, 1 means exactly the same) between the pattern and the input network. After comparing with all patterns, we select the one which produces the highest similarity score. However, if all similarity scores are smaller than a threshold (i. e., 0.5), then the network mismatches with all patterns. Hence, it would be assigned an *undefined* pattern.

First of all, we adjust the number of nodes in patterns. For instance, if the network has 100 nodes, then the graph induced by a *chain* pattern will be constructed by 100 nodes. Each node is represented by a vector which contains values of a few structural features. Graphs are described by feature matrices. Suppose the network has n nodes, and we extract k features from each node, then the feature matrix can be defined as: $M = \begin{bmatrix} f_{11} & f_{12} & \cdots & f_{1k} \\ f_{21} & f_{22} & \cdots & f_{2k} \\ \cdots & \cdots & \cdots & \cdots \\ f_{n1} & f_{n2} & \cdots & f_{nk} \end{bmatrix}$, where f_{ij} is the jth feature of node i. A matrix produces a signature vector V and the similarity score equals to the Canberra Distance between

two signature vectors, $Distance(V, V') = \sum_{i=1}^{k} \frac{|V_i - V'_i|}{V_i + V'_i}$. The distance measure is sensitive to small changes near zero [Berlingerio et al. 2012].

The customised approach is implemented with a few decisions. Suppose the number of nodes in the network is n. A primary step is to check if there exists a cycle in it. If cycles are detected, we count the number of nodes Cyc_G in the largest cycle. If it exceeds $threshold_{cyc}$, the network can be initially inferred to have a *loop* or *clique* pattern. Otherwise, the pattern can be either *chain* or *egocentric*. The *loop* and *clique* patterns are further distinguished by measuring the graph density $Den_G = \frac{2m}{n(n-1)}$, where m and n are the number of links and nodes respectively. If Den_G is larger than $threshold_{cli}$, then the pattern is *clique*. Otherwise, if Den_G is less than $threshold_{loo}$, then the graph has a *loop* pattern.

To decide between the *chain* and *egocentric* patterns, we calculate the graph depth Dep_G, which equals to the length of the longest path between nodes. If Dep_G is larger than $threshold_{cha}$, the pattern is *chain*. If the maximum node degree Deg_G exceeds $threshold_{ego}$, the graph is likely to have a *egocentric* pattern. Algorithm 1 gives the details. The cycle detection and the calculation of graph depth are realized by a *depth-first* search.

At individual time steps, graphs induced by $P_{i1}, P_{i2}, \ldots, P_{ik}$ in G_i are input into the matching algorithm so that we get corresponding patterns of each partition. In our work, large graphs are divided into partitions to obtain manageable sizes. We then project complex and diverse structures of partitions to few visual comparable patterns. The visual clutter and overlapping problems are largely reduced. Due to the simplicity of patterns, users can clearly see the distribution of different types of structures. Besides, it becomes easier to identify partitions of similar structures and monitor the evolution of the whole network.

4.3 Smooth Animation

During the transition between time steps, stable visualizations let users concentrate on fundamental changes of structures. We improve the extent of smoothness from two aspects, one is the graph layout and the other is the pattern transformation.

In our design, a node depicts a partition of the graph. For G_i, partitions $P_{i1}, P_{i2}, \ldots, P_{ik}$ are obtained by checking individual entities′ community id in the super-community. Some partitions might be empty. We can see from Figure 1 that community detection which implements the division only needs to be executed once. If divisions have to be executed for t times, it would cost a long time. Though $P_{i1}, P_{i2}, \ldots, P_{ik}$ might differ from actual communities in G_i, nodes in P_{ij} still maintain relatively dense connections, because they used to belong to the same group in the super-community.

We only display non-empty partitions on the screen. If we layout the graph separately at each time step, it would be time-consuming and also the layout would be dramatically different. According to Definition 3.5, P_{ij} and com_j have same positions. Hence, for two time steps m and n, P_{mj} and P_{nj} stay the same place. If no dramatic changes happen, the division of the network would be very alike over time steps. The layout can keep stable because same partitions stay in the same positions. As shown in Figure 1, the layout calculation also needs for one-time execution.

Algorithm 1: Customised algorithm of matching a graph with one of the *chain*, *loop*, *clique*, and *egocentric* patterns. If none of the patterns is matched, return *undefined*.

```
G = (V, E) // initiate a graph
Function Pattern_Matching (G)
    Cyc_G=DetectCycle(G); // detect cycles in G, if detected,
     return the size of the largest cycle, otherwise return False
    if Cyc_G > threshold_cyc then
        Den_G=CalcDensity(G); // calculate the graph density
        if Den_G > threshold_cli then
            return "Clique Pattern";
        else
            if Den_G < threshold_loo then
                return "Loop Pattern";
            else
                return "Undefined";
            end
        end
    else
        Deg_G=MaxDegree(G); // calculate the maximum
         degree of nodes
        Dep_G=CalcDepth(G); // calculate the longest path
         starting from any node of the graph
        if Deg_G > threshold_ego then
            return "Egocentric Pattern";
        else
            if Dep_G > threshold_cha then
                return "Chain Pattern";
            else
                return "Undefined";
            end
        end
    end
end
```

In Figure 3 (b), the layout of consecutive time steps are extracted from the *super-layout*, while in Figure 3(a), the layout is computed separately. With the *super-layout*, we can achieve much more stable visualizations.

Except for the layout, we also consider the smoothness of pattern transitions. The pattern of a partition is embeded into its corresponding node. We want the pattern to change smoothly if it is different at two neighboring time steps. To achieve this goal, we make a little transformation to patterns. Details are given in subsection 5.1.

5 VISUAL DESIGN

Our animated visualization is designed for the purpose of assisting users to perform the following visual tasks, which mainly relate to partitions and patterns:

- *Q1:* How many partitions does the graph have at each time step? How does the number change over time? Is there any reason that could explain the change?
- *Q2:* How is the stability of partitions? Does any of them exist for continuous time steps? What contributes to the stability?

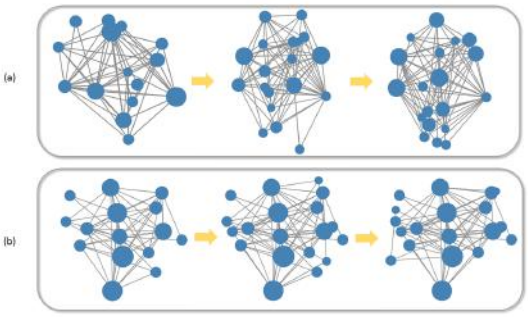


Figure 3: Comparison of layouts generated with and without the *super-layout*. Arrow indicates the time order. (a) Layouts are generated by iteratively applying the force-directed algorithm. (b) The layout of each time step is extracted from the *super-layout*.

- *Q3:* For those durable partitions, do they also keep a steady structure? How does the structure change? Which pattern appears most frequently in a partition? At which time step, does the structure change?
- *Q4:* At a specific time step, what is the distribution of patterns? Which one occupies the most? How many partitions have a different structure, while how many others preserve their previous structure?

5.1 Pattern Transformation

To let the users have a straightforward understanding of the underlying structure, the pattern representation is embeded concentrically into the node which indicates a partition. As we mentioned in subsection 4.3, visual representations of patterns should proceed to each other smoothly. Therefore, we need to do a little transformation to patterns. Above all, we bend the *chain* pattern, so that all patterns can be illustrated by a pentagon where the five vertices are fixed and only edges change. Figure 4 shows how the *loop* pattern smoothly evolves to other patterns. For example, if the target pattern is a *chain*, only the bottom edge needs to be removed. For the *egocentric* pattern, there would be edges that appear and disappear simultaneously. Evolutions between other pairs of patterns are implemented in a similar way. In any case, the solid frames of patterns would bring a fluent visual perception during the animation.

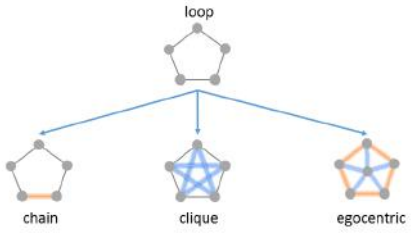


Figure 4: The *loop* pattern smoothly moves to all other patterns. Emerging edges are marked by *blue* halos and *orange* halos represent disappearing edges.

5.2 Evolution Representation

Because in animation, time steps are displayed one by one, users can barely do comparisons between them and see the difference. To alleviate this problem, we present two patterns of a partition together, one is for the previous time step, and the other is for the current time step. In this way, users can distinctly see changes through visual comparisons. Considering the compactness, we place a smaller and concentric circle in a node, then fit the previous pattern into this circle. As shown in Figure 5, previous patterns locate in smaller circles. Also, we assume that current patterns are more important, thus they should occupy more space. It is worth noting that though the *clique* and *egocentric* patterns are partly covered by the inner area, we can still clearly recognize them.

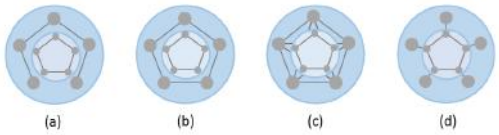


Figure 5: Pattern comparison of two consecutive time steps. The current pattern is placed in the outer circle, while the previous one locates in the inner circle. (a), (b), (c) and (d) have the current pattern as *chain, loop, clique* and *egocentric* respectively. In this example, we draw a *loop* as the previous pattern. In reality, patterns depend on matching results.

Because of the graph evolution, multiple changes might happen at each time step. For example, new partitions appear and old partitions disappear. Even for durable partitions, their sizes might expand or shrink, not to mention the structure changes. To reveal these changes, we take advantage of diverse visual encodings which are listed in Table 1. Partitions are represented by circles with a default blue filling. We also make the filling translucent in case that patterns become too vague to see.

We take seven scenarios of changes into consideration. If a partition newly appears at the current time step, then it must not exist in the last time step. Hence, we fill the inner circle with white color. Conversely, if a partition disappears, users are still allowed to view its previous pattern, but the outer circle is filled with white color. If the inner or outer circle is empty, it suggests that the pattern is *undefined*. For partitions whose sizes are increased, they have a thick and solid border line of the circle. While for those that are decreased, the border line is thin and dashed. If the size remains the same, then we use a thin and solid line. However, even if the size does not change, the members or the structure of a partition might vary. Though users can discover structure changes by comparing patterns in outer and inner circles, it is not feasible for them to check all partitions. Therefore, it is necessary to visually inform users which part of the network involves structural changes. In our design, we use the *red* or *green* color to fill inner circles, corresponding to situations that structures are changed or unchanged. In general, we only use colors and shapes to differentiate changes, and users will not be overwhelmed by the visual encodings.

Table 1: Visual encodings of changes that might happen to a partition. The appearance and disappearance of a partition is revealed by the white filling. Changes of the size are distinguished by the border line as solid or dashed, thin or thick. The inner area is filled with *red* or *green* color, if the partition's structure is changed or not.

Type of Changes	appear	disappear	expand	shrink	same size	structure changed	structure unchanged
Visual Encoding							

6 USE CASE

We experimented with a synthetic dataset and the DBLP dataset [DBL 2017] to show the effectiveness of our method. The effectiveness reflects on two aspects, one is the accuracy of the pattern matching algorithm, and the other is how effectively our visual design can help users to accomplish the visual tasks. In this paper, visual designs are implemented with D3.js [Bostock et al. 2011].

6.1 Example 1

The accuracy of our pattern matching approaches, especially the customised one, is verified by a synthetic dataset. Graphs in the dataset are drawn manually so that their structures can be controlled. We judge the patterns of these graphs by observation, and label them with *chain*, *loop*, *clique*, *egocentric*, or *undefined*. In order to keep the reliability of judgement, we avoid to adopt complex graphs whose patterns are difficult to be observed by naked eyes.

This synthetic dataset is utilized as the benchmark of the pattern matching algorithm. The *samples* row of Table 2 lists 20 out of the total 50 graphs. These samples come in the form of different sizes and structures. The *label* row gives the observed type of patterns. GPM refers to the generic method, and CPM is the customised method. If the matching result is different from the label, we think that the method fails in this case.

CPM is implemented by Algorithm 1, and we need to set values for a few thresholds. Suppose the number of nodes in a partition is represented by $n = |V|$. Initially, four patterns fall into two branches by $threshold_{cyc}$, which is the number of nodes in the largest cycle. We set its value to $\frac{n}{2}$, which means at least half nodes of the partition composes the largest cycle. For the first branch of the algorithm, if the edge density exceeds $threshold_{cli} = 0.7$, then we infer that the pattern is *clique*. Otherwise, if the density is less than $threshold_{loo} = 0.3$, the inferred pattern is *loop*. For the second branch, either *chain* or *egocentric* is selected by calculating the maximum node degree and the longest path between nodes. If $Deg_G > threshold_{ego}$, where $threshold_{ego} = 0.7n$, the algorithm ends with a *egocentric* pattern. It implies that there exists a node which connects to at least 70% other nodes in the partition. If the algorithm returns a *chain* pattern, Dep_G must be greater than $threshold_{cha} = 0.6n$. We tested different values for parameters and the current setting achieves highest accuracy on the dataset.

Table 2 shows the matching results of GPM and CPM. Compare the results with corresponding labels, we find that CPM can obtain a higher accuracy. Among the all 50 samples, CPM has an accuracy up to 82%, while the accuracy of GPM reaches to 56%.

6.2 Example 2

To present the visual effectiveness of our design, we experiment with the DBLP dataset [DBL 2017]. The objective is to help users to accomplish the visual tasks proposed in section 5. We extracted the coauthorship data of a researcher over 15 years, and each year is a time step. The data of a year constructs an egocentric network, where the researcher is the *ego* and nodes that directly connect to the ego are called *alter*. In this example, we also consider the relationships between alters. There are about 640 authors and 3800 coauthorships involved in 15 years.

Figure 6(a) shows the super-graph of 15 time steps. After community detection, we get the super-community, which is shown in Figure 6(b). It is the basis of partition and layout extractions.

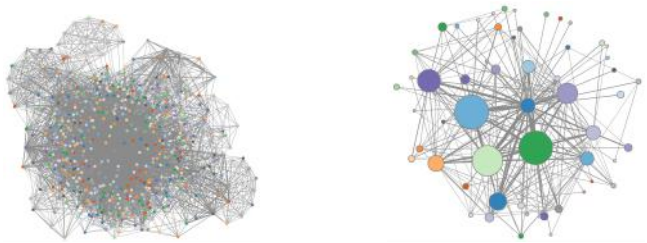
Figure 7 shows snapshots of the network at three years. Despite the size change of partitions, we can observe that the animation is quite smooth, because the locations of partitions are fixed. Besides, each time step has similar components of partitions, which means nodes are not frequently added or removed from the nework. To have a clear observation, we only show patterns of partition 0, 1, 2, 4, 8, and 9. The partition id is placed at the node center. There are about 10 partitions in each year, and their sizes are relatively stable. For example, partition 1, 4, 8 always have lager sizes than others (*Q1, Q2*). Hence, we can say that the overall membership of the network does not change too much in these three years. However, it does not mean that the relationships between members stay the same. The structural changes are especially obvious in local areas. In our design, *red* color of the inner circle denotes the change of structures. In all three years, there exist partitions that encounter structural changes. In year 2007, the topology of partition 4 turns from *chain* to *egocentric*. Also, compared to year 2006, its size becomes smaller, which can be seen from the dashed border (*Q3*). If the pattern of a partition is *undefined*, we leave the inner or outer circle of the node as empty. In year 2005, the pattern of partition 1 is *egocentric*, while in year 2006 and 2007, its pattern is *undefined*. Except for those dynamic partitions, some others have very solid structures, and partition 0 is one of them. It has an *egocentric* topology all the time. After investigation, we found that the author who acts as an *ego* belongs to partition 0. As we explained, the *ego* dominates in the network, and all other objects connect to it. Therefore, partition 0 always keeps the *egocentric* pattern. We can also discover some emerging nodes by the *white* filling of the inner circle, such as partition 2 in year 2006. In this example, four pre-defined patterns appear in different partitions, and their distributions seem random (*Q4*), excluding partition 0.

Table 2: Pattern matching results of the generic method (GMM) and the customized method(CMM). The 'samples' row gives the node-link representation of graphs. These graphs are the input to pattern matching algorithms. The 'label' row contains pattern labels that are marked by observation. The 'GPM' and 'CPM' rows show the outcome of GPM and CPM algorithms.

	01	02	03	04	05	06	07	08	09	10
samples										
label	string	ring	undefined	star	undefined	ring	star	dense	star	undefined
GPM	string	dense	undefined	star	undefined	dense	dense	undefined	star	string
CPM	string	ring	undefined	star	undefined	star	dense	undefined	undefined	undefined

	11	12	13	14	15	16	17	18	19	20
samples										
label	string	star	string	ring	undefined	dense	dense	star	ring	star
GPM	dense	star	string	dense	star	dense	dense	star	dense	star
CPM	string	star	string	ring	undefined	undefined	dense	string	ring	star

Because, the *ego* and *alters* always compose an *egocentric* network. However, relationships between *alters* are uncertain.



(a) (b)

Figure 6: DBLP network. (a) Super-graph of 15 time steps. Each node indicates an author. (b) Super-community and super-layout. Each node denotes a community. We decide the partitions and layouts of all time steps based on this figure.

6.3 Discussion

When dealing with large data sets, focusing on the sub-sets can largely relieve the difficulty of analysis. For time-varying graphs, the sub-set can be obtained from both the temporal dimension and the spatial dimension. If data is uniformly divided along the temporal dimension, we can get various time granularity. Namely, the time step can be a year, a month or a day. Aggregating multiple time steps into a larger time granularity (e. g., aggregating the hourly data into daily data) allows us to take a longer time period into consideration. In this paper, we tend to group data in the spatial dimension. For each time step, we desire to simplify the structure of the graph. Therefore, *community detection* approach is applied to cluster graph nodes into communities where nodes are highly connected. The graph is separated into sub-graphs according to link densities, which is a type of the topology property. However, we can also group nodes or links with respect to their attributes. Take the social network as an example, nodes that represent users can be divided into two groups with different genders. In the first example, we try to test the reliability of the pattern matching algorithm. However, expanding the volume of the benchmark data set and covering more graphs with diverse topologies may better reveal the applicability. Besides, with more benchmark samples, we can improve the algorithm as well as its parameters. Our approach has quite good scalability and time-efficiency, because we decide all partitions and layouts at a global scale. Operations of aggregating nodes and producing layouts only need to be conducted once, rather than doing repeated calculations at each time step. Method proposed in this paper works under the premise that the time-varying graph data has been given. Hence, it is ill-suited to be used in online visualizations.

7 CONCLUSION

In this paper, we implemented the smooth animation for exhibiting the structure evolutions of time-varying graphs. Users are supported to view both global and local structure changes. We achieved the consistency of graph layout by extracting partitions from the super-community. The complex structures of graphs become visually comparable, because we project them to simple patterns. Patterns represent topological properties of underlying graphs induced by partitions. It is convenient to visually search for partitions of similar structures and also identify changes that occur to a specific partition. For future work, we plan to improve the current visual design by experimenting with more datasets. In addtion, we want to apply the basic idea of this paper on multivariate time-varying graphs so that entities can be clustered by their attributes instead of topological properties.

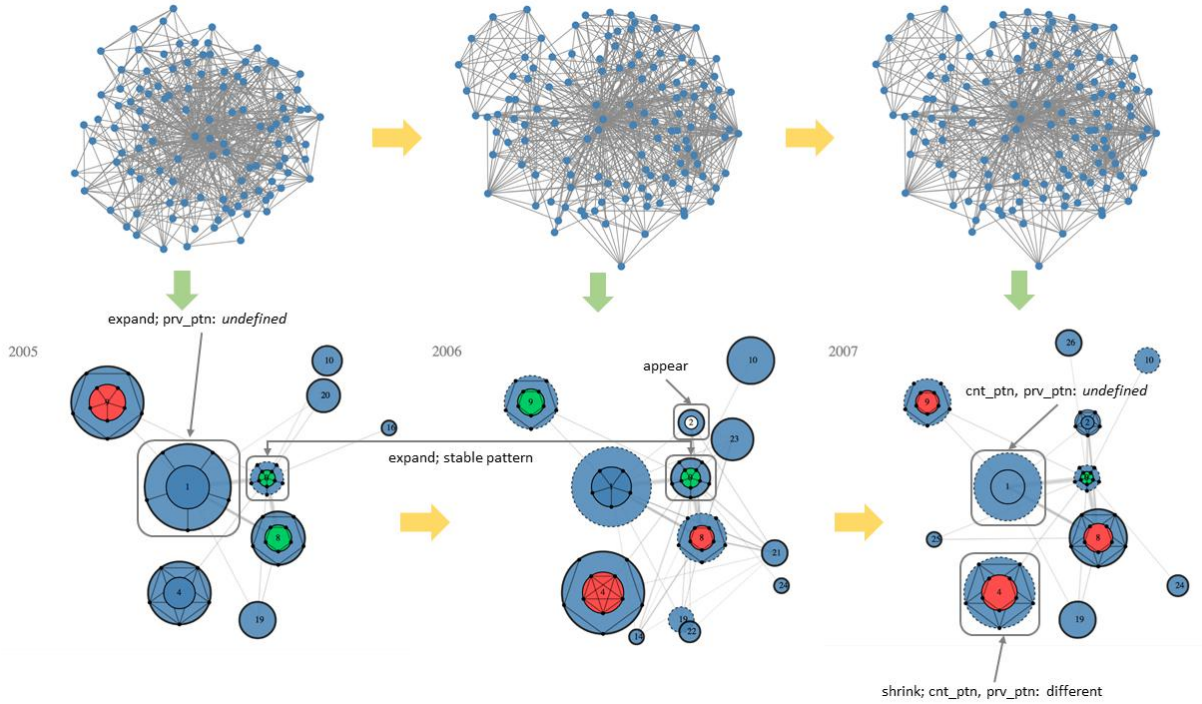


Figure 7: Snapshots of the animation at three time steps. The overall structure remains stable, while local structures keep changing. Only patterns of partitions 0, 1, 2, 4, 8, and 9 are shown. Rectangles and texts explain typical changes of partitions. prv_ptn means the previous pattern, and cnt_ptn is the current pattern.

ACKNOWLEDGMENTS

The authors would like to acknowledge the partial support of IGRF PolyU 152142/15E and Project 4-ZZFF from the Department of Computing, The Hong Kong Polytechnic University.

REFERENCES

2017. DBLP. (2017). http://dblp.uni-trier.de/xml/

Benjamin Bach, Emmanuel Pietriga, and Jean-Daniel Fekete. 2014. GraphDiaries: animated transitions andtemporal navigation for dynamic networks. *IEEE Transactions on Visualization and Computer Graphics* 20, 5 (2014), 740–754.

Michele Berlingerio, Danai Koutra, Tina Eliassi-Rad, and Christos Faloutsos. 2012. NetSimile: a scalable approach to size-independent network similarity. *arXiv preprint arXiv:1209.2684* (2012).

Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. 2008. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment* 2008, 10 (2008), P10008.

Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. 2011. D^3 data-driven documents. *IEEE transactions on visualization and computer graphics* 17, 12 (2011), 2301–2309.

Ulrik Brandes. 2001. A faster algorithm for betweenness centrality. *Journal of mathematical sociology* 25, 2 (2001), 163–177.

Matthew Brehmer, Bongshin Lee, Benjamin Bach, Nathalie Henry Riche, and Tamara Munzner. 2016. Timelines Revisited: A Design Space and Considerations for Expressive Storytelling. *IEEE Transactions on Visualization and Computer Graphics (TVCG, Proceedings of InfoVis 2015)* 22, 1 (2016), 449–458.

Stephan Diehl, Carsten Görg, and Andreas Kerren. 2001. Preserving the mental map using foresighted layout. In *Data Visualization 2001*. Springer, 175–184.

Cody Dunne and Ben Shneiderman. 2013. Motif simplification: improving network visualization readability with fan, connector, and clique glyphs. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 3247–3256.

Tim Dwyer. 2009. Scalable, versatile and simple constrained graph layout. In *Computer Graphics Forum*, Vol. 28. Wiley Online Library, 991–998.

Kun-Chuan Feng, Chaoli Wang, Han-Wei Shen, and Tong-Yee Lee. 2012. Coherent time-varying graph drawing with multifocus+ context interaction. *IEEE Transactions on Visualization and Computer Graphics* 18, 8 (2012), 1330–1342.

Yaniv Frishman and Ayellet Tal. 2004. Dynamic drawing of clustered graphs. In *Information Visualization, 2004. INFOVIS 2004. IEEE Symposium on*. IEEE, 191–198.

Michelle Girvan and Mark EJ Newman. 2002. Community structure in social and biological networks. *Proceedings of the national academy of sciences* 99, 12 (2002), 7821–7826.

Jialin He, Duanbing Chen, Chongjing Sun, Yan Fu, and Wenjun Li. 2017. Efficient stepwise detection of communities in temporal networks. *Physica A: Statistical Mechanics and its Applications* 469 (2017), 438–446.

Jeffrey Heer and Danah Boyd. 2005. Vizster: Visualizing online social networks. In *Information Visualization, 2005. INFOVIS 2005. IEEE Symposium on*. IEEE, 32–39.

Chenhui Li, George Baciu, and Yunzhe Wang. 2015. ModulGraph: Modularity-based Visualization of Massive Graphs. In *SIGGRAPH Asia 2015 Visualization in High Performance Computing (SA '15)*. 11:1–11:4.

Steven Noel and Sushil Jajodia. 2004. Managing attack graph complexity through visual hierarchical aggregation. In *Proceedings of the 2004 ACM workshop on Visualization and data mining for computer security*. ACM, 109–118.

Corinna Vehlow, Fabian Beck, Patrick Auwärter, and Daniel Weiskopf. 2015. Visualizing the evolution of communities in dynamic graphs. In *Computer Graphics Forum*, Vol. 34. Wiley Online Library, 277–288.

Corinna Vehlow, Fabian Beck, and Daniel Weiskopf. 2017. Visualizing group structures in graphs: A survey. In *Computer Graphics Forum*, Vol. 36. Wiley Online Library, 201–225.

Tatiana Von Landesberger, Felix Brodkorb, Philipp Roskosch, Natalia Andrienko, Gennady Andrienko, and Andreas Kerren. 2016. Mobilitygraphs: Visual analysis of mass mobility dynamics via spatio-temporal graphs and clustering. *IEEE transactions on visualization and computer graphics* 22, 1 (2016), 11–20.