Visually Analyzing Large Scale Graphs

Dissertation

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

Yang Zhang, M.S.

Graduate Program in Computer Science and Engineering

The Ohio State University

2015

Dissertation Committee:

Srinivasan Parthasarathy, Advisor
Yusu Wang
Arnab Nandi
Abstract

Many datasets can be modeled as graphs, and graphs naturally capture the relations among entities in the datasets. Graph visualization is an effective way to help users understand the graph data. However a fundamental challenge for graph visualization is to visualize large scale graphs: most of the graph visualization methods usually run into the limitation of display device – that is at large graph scales one is often unable to represent nodes and edges effectively within the available pixel limits of modern displays. In this thesis, we propose a framework focusing on visually analyzing large-scale networks. Our framework has the following three components, and each component focuses on visually analyzing one particular aspect of a graph.

The first component is to visually analyze dense subgraphs. We define and introduce the notion of a Triangle K-Core, a dense subgraph pattern and can be used as a proxy for identifying and extracting clique-like structure from large graphs. Based on this definition we first develop a localized algorithm for extracting Triangle K-Cores from large graphs. Subsequently we extend the simple algorithm to accommodate dynamic graphs (where edges can be dynamically added and deleted). Finally, we extend the basic definition to support various template pattern cliques with applications to network visualization and event detection on graphs and networks.

Based on the Triangle K-Core motif definition, we propose a parallel method to detect the approximate cliques in large graphs. Our method first partitions the graph, and extends
each partition by its vertices’ immediate neighbors. Then we distribute these extended partitions to different computing nodes, and detect approximate cliques (Triangle K-Cores) on each partition. The benefit of our method is each computing node works independently, and the communication overheads are minimized, and therefore we observe good scalability and parallel efficiency.

The second component is to visualize scalar graphs. We define a scalar graph as a graph with numerical attributes associated with nodes (or edges). Such numerical attributes can represent raw content information or derived information reflecting important network measures of interest such as triangle density and centrality. Our method aims to uncover the relationship between attribute values and graph topology and relies on transforming the network to generate a terrain visualization in 3D space. We demonstrate that the terrain reveals the overall distribution of the attribute values over the graph, and highlights the areas in the graph that are of particular interest to the user. We also make use of terrain visualization to visually capture the relationship between multiple numerical attributes on one graph.

The third component focuses on visually analyzing dynamic graphs. We propose to represent a dynamic graph as bitstreams – every edge is associated with a bitstream where each bit indicates its appearance/disappearance in each snapshot. Then we apply time series analysis methods to analyze the bitstreams, and identify major periods in the dynamic graph. Based on the periods detected, we could find the communities that appear periodically in the dynamic graph. Also, we propose to use terrain visualization to identify topologically and temporally correlated edges in the dynamic graph. These edges usually comprise a cluster which occurs many times in the dynamic graph. Moreover, based on bitstream representation, we propose to use “dense block” to represent a dense subgraph
occurring frequently during a certain period, and design an efficient way to detect “dense blocks” in the dynamic graph.
This work is dedicated to my parents
Acknowledgments

The first one I would like to thank is Dr. Srinivasan Parthasarathy (Srini). He leads me into the data mining research area, and pointed the interesting and important directions for my research. Under his guidance I learned how to conduct research and how to write research paper. He always encouraged me and helped me when my research got stuck, and worked with me until late night when paper submission deadline approaches. He also gave me invaluable help to find collaborators for my research project and refer me to internship opportunities. It is a great luck to have Srini as my advisor.

I also want to thank Dr. Yusu Wang for her great help on the terrain visualization project. She discussed the details of algorithm and experiments with me, and helped improve the quality of this work a lot. It is very interesting to work with her.

My candidacy committee members, Dr. Arnab Nandi and Dr. Raghu Machiraju gave me a lot of useful suggestions during my candidacy exam. I also took their courses about data visualization, and these courses help me a lot during my research.

All the data mining lab members have generously given me great help during the past 6 years. They are Dave Fuhry, S M Faisal, Yiye Ruan, Yu-keng Shih, Tyler Clemons, Ye Wang, Venu Satuluri, Xintian Yang, Jiongqian Liang, Yu Wang, Aniket Chakrabarti, Bortik Bandyopadhyay. They discussed ideas with me and worked with me on research projects. I learn a lot from them, and have a lot of fun with them.
I did three internships during the PhD study, and I want to thank my mentors during my internship, they are Dr. Raghavachari Madhavan (JPMorgan Chase), Dr. Yuli Ye(Amazon), Dr. Dmitriy Belyi(Amazon) and Dr. Gengxin Miao (Google). They opened the window to the industry world for me, so I can have a chance to learn the exciting techniques used in these great companies.

Finally and the most importantly, I want to thank my parents for their love and support, they always stand by me and help me get through every difficult time during the PhD study.
Vita

06/2009 .......................... Bachelor of Engineering, Zhejiang University, China
06/2012 - 08/2012 ........................ Intern at JPMorgan Chase
05/2013 .......................... Master of Science, the Ohio State University, United States
05/2013 - 08/2013 ........................ Intern at Amazon
05/2014 - 08/2014 ........................ Intern at Google
09/2009 - present ......................... Graduate Student, the Ohio State University, United States

Publications

Research Publications


Fields of Study

Major Field: Computer Science and Engineering
# Table of Contents

| Abstract | ii |
| Dedication | v |
| Acknowledgments | vi |
| Vita | viii |
| List of Tables | xiii |
| List of Figures | xiv |

1. Introduction ................................................. 1
   1.1 Related Work ........................................... 2
   1.2 Challenges .............................................. 5
   1.3 Thesis Statement ....................................... 7
   1.4 Our Contributions ...................................... 7
   1.5 Organization .......................................... 10

2. Extracting Analyzing and Visualizing Triangle K-Core Motifs within Networks 11
   2.1 Introduction ............................................. 11
   2.2 Preliminaries .......................................... 13
   2.3 Triangle K-Core Algorithm ............................ 15
      2.3.1 Detecting Maximum Triangle K-Core ............. 15
      2.3.2 Updating Maximum Triangle K-Core ............. 19
   2.4 Extensions .............................................. 22
   2.5 Relationship to DN-Graph ............................... 27
   2.6 Experiments ............................................. 29
      2.6.1 Comparison with CSV and DN-Graph ............. 29
2.6.2 Protein-Protein Interaction (PPI) Case Study .................................. 31
2.6.3 Experimental Results of Update Algorithm ..................................... 31
2.6.4 Dual View Plots: Wiki Case Study .................................................. 34
2.6.5 Dynamic Template Pattern Cliques: DBLP Study ............................... 35
2.6.6 Static Template Pattern Cliques: PPI Case Study ............................... 38
2.7 Details of Update Algorithm ............................................................... 40
2.7.1 Triangle K-Core Update Algorithm .................................................. 40

3. A Fast Parallel Method to Detect Approximate Cliques .............................. 45
3.1 Introduction .......................................................................................... 45
3.2 Preliminaries and Background .............................................................. 46
3.2.1 Clique-like subgraph structures ....................................................... 46
3.3 Parallel Algorithm to Detect Approximate Cliques ................................. 48
3.3.1 Basic Method .................................................................................. 48
3.3.2 Sparsification .................................................................................. 49
3.3.3 Load Balancing .............................................................................. 52
3.3.4 An Out-of-Core Variant .................................................................. 53
3.4 Application of Approximate Cliques ...................................................... 54
3.4.1 Generate CSV plot ......................................................................... 54
3.5 Experiments ......................................................................................... 54
3.5.1 Performance .................................................................................... 55
3.5.2 CSV plots ....................................................................................... 56

4. Analyzing and Visualizing Scalar Fields on Graphs .................................... 61
4.1 Introduction .......................................................................................... 61
4.2 Single Scalar Field ............................................................................... 64
4.2.1 Vertex-based Scalar Graph .............................................................. 64
4.2.2 Vertex Scalar Tree .......................................................................... 66
4.2.3 Edge-based Scalar Graph ................................................................. 72
4.2.4 Relation between maximal α-(edge) connected component and Dense Subgraph .................................................. 75
4.2.5 Visualizing Scalar Trees via Terrain Metaphor .................................. 77
4.3 Handling Multiple Scalar Fields ............................................................ 81
4.4 Evaluation ............................................................................................ 82
4.4.1 Visualizing Dense Subgraphs .......................................................... 83
4.4.2 Visualizing Communities and Roles ............................................... 86
4.4.3 Comparing Different Centralities ................................................... 91
4.4.4 Extension to Visualizing Non-graph Data ........................................ 94
4.5 User Study ............................................................................................ 96
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Time Cost Comparison (seconds)</td>
<td>29</td>
</tr>
<tr>
<td>2.2 Memory Comparison</td>
<td>29</td>
</tr>
<tr>
<td>2.3 Update Algorithm Time Cost (seconds)</td>
<td>34</td>
</tr>
<tr>
<td>3.1 Parallel Algorithm Data Sets</td>
<td>54</td>
</tr>
<tr>
<td>3.2 Parallel Algorithms Time Cost Comparison (seconds)</td>
<td>55</td>
</tr>
<tr>
<td>4.1 Dataset Properties</td>
<td>83</td>
</tr>
<tr>
<td>4.2 Terrain Visualization Time Cost(sec)</td>
<td>88</td>
</tr>
<tr>
<td>4.3 Book names of some nodes in Figure 4.8(b)</td>
<td>91</td>
</tr>
<tr>
<td>4.4 Time Cost(sec) and Accuracy of Finishing Task 1</td>
<td>98</td>
</tr>
<tr>
<td>4.5 Time Cost(sec) and Accuracy of Finishing Task 2</td>
<td>98</td>
</tr>
<tr>
<td>4.6 Time Cost(sec) and Accuracy of Finishing Task 3</td>
<td>99</td>
</tr>
<tr>
<td>5.1 Periodic Communities</td>
<td>124</td>
</tr>
</tbody>
</table>
**List of Figures**

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Overview</td>
<td>10</td>
</tr>
<tr>
<td>2.1</td>
<td>K-Core vs. Triangle K-Core</td>
<td>14</td>
</tr>
<tr>
<td>2.2</td>
<td>Examples for Illustrating Triangle K-Core Algorithms</td>
<td>17</td>
</tr>
<tr>
<td>2.3</td>
<td>Several <em>template pattern cliques</em> and their characteristic triangles</td>
<td>24</td>
</tr>
<tr>
<td>2.4</td>
<td>Qualitative Comparison between CSV and Triangle K-Core</td>
<td>32</td>
</tr>
<tr>
<td>2.5</td>
<td>Cliques in PPI dataset</td>
<td>33</td>
</tr>
<tr>
<td>2.6</td>
<td>Dual View Plots for Clique Changes</td>
<td>36</td>
</tr>
<tr>
<td>2.7</td>
<td>Plot of <em>Emerging Cliques</em> in DBLP 2004</td>
<td>37</td>
</tr>
<tr>
<td>2.8</td>
<td>Plot of <em>Bridge Cliques</em> in DBLP 2004</td>
<td>37</td>
</tr>
<tr>
<td>2.9</td>
<td>Plot of <em>Expanding Cliques</em> in DBLP 2001</td>
<td>37</td>
</tr>
<tr>
<td>2.10</td>
<td>Detect <em>Bridge Cliques</em> in PPI dataset</td>
<td>38</td>
</tr>
<tr>
<td>3.1</td>
<td>CSV plots comparison for Web</td>
<td>57</td>
</tr>
<tr>
<td>3.2</td>
<td>CSV plots comparison for Flickr</td>
<td>58</td>
</tr>
<tr>
<td>3.3</td>
<td>CSV plots comparison for Wiki2010</td>
<td>59</td>
</tr>
<tr>
<td>3.4</td>
<td>CSV plots comparison for Orkut</td>
<td>60</td>
</tr>
</tbody>
</table>
4.1 Examples of Terrain Visualization .................................. 63
4.2 Scalar Graph and Scalar Tree ......................................... 68
4.3 Postprocessing Scalar Tree ........................................... 71
4.4 Terrain Visualization of a Simple Scalar Tree ....................... 80
4.5 Visualizing K-Cores in GrQc and wikiVote network .................. 85
4.6 Visualizing K-Cores and K-Trusses .................................. 87
4.7 Visualizing two communities in DBLP network ....................... 89
4.8 one community of Amazon co-purchase network ..................... 90
4.9 Compare Degree and Betweenness Centralities ........................ 93
4.10 Terrain Visualization of Medical Dataset ............................ 94
4.11 Terrain Visualization of Query Results .............................. 96
4.12 Optional caption for list of figures ................................ 100
4.13 Optional caption for list of figures ................................ 101
4.14 Optional caption for list of figures ................................ 104
5.1 An example dynamic graph and its aggregated graph with bitstreams .... 107
5.2 Converting Dynamic Graph to Weighted Graph ....................... 115
5.3 An Example of Terrain Visualization ................................ 116
5.4 An Example of Dense Block ........................................... 118
5.5 Cumulative DFT/Autocorrelation Plots for Enron ...................... 125
5.6 Cumulative DFT/Autocorrelation Plots for DBLP ...................... 126
5.7 Cumulative DFT/Autocorrelation Plots for Facebook ................... 127
5.8 Cumulative DFT/Autocorrelation Plots for Congress ................. 128
5.9 Congress Dataset .................................................. 130
5.10 DBLP Dataset ....................................................... 132
6.1 Example of a Matrix Representation ................................. 139
6.2 Hierarchical Structure of Bit Streams ............................... 140
6.3 Local Structural Pattern ............................................ 141
Chapter 1: Introduction

Many real world problems can be modeled as complex entity-relationship graphs where nodes represent entities of interest and edges mimic the relationships among them. Fueled by technological advances and inspired by empirical analysis, the number of such problems and the diversity of domains from which they arise – physics, sociology, biology, chemistry, and world wide web – are growing steadily.

The analysis of such graphs can help us understand the structure and function of such systems, and the ability to visualize graphs at scale is fundamental to analyzing them. Graph visualization methods can not only confirm the analysis results generated by other graph mining algorithms, but also strengthen the users’ analytic abilities by giving them flexibility to explore the data themselves. Graph visualization can be used in several contexts but in this thesis we focus on the following three major use cases:

- Detecting clusters: it tries to identify groups of nodes/edges that are close in topology. For example, identifying groups of proteins that function in a coherent fashion.

- Detecting outliers: it tries to identify nodes/edges that behave differently from others, for example, identifying spammers in an email network.
• Detecting dynamic patterns: it tries to identify patterns in evolving dynamic graphs.

For example, identifying the communities that appear periodically in a dynamic social network.

In each of the above aspects, graph visualization methods could be used along with other graph mining algorithms to convey additional information. For example, when visualizing communities in a social network, graph visualization methods could reveal the relationships among the communities (which community is closely related to which other community), while most community detection algorithms do not have such ability.

Visualizing such complex networks and honing in on important and possibly evolving topological characteristics is difficult given the size and complexity of such networks, but nonetheless important. This in turn has led to numerous advances in graph drawing and network visualization techniques. The challenge is due to the limitation of display device – that is at large graph scales one is often unable to represent nodes and edges effectively within the available pixel limits of modern displays.

In the following we first give a brief survey of the recent work related to graph mining and visualization. Then we list a few challenges related to graph visualization. Finally we present our work that addresses these challenges.

1.1 Related Work

Information visualization conveys the information in a dataset through visual representation, and aims to help users have more insight into a dataset. A few techniques have been proposed to visualize different types of data. For example, some techniques use tree/treemap to represent hierarchical data [6, 64], some use matrix to represent the relationship among entities in a dataset [42, 43], and some use parallel coordinates to represent...
high-dimensional data [82, 86]. Visualizing a large dataset is especially challenging. Many large scale data visualization methods are designed based on the idea “overview first, zoom and filter, then details on demand” [63]. In the following we will introduce some related works focusing on visualizing and analyzing graph data.

Many graph visualization methods are closely related to graph clustering. In the context of graph clustering, several methods have often found favor. For example, spectral methods[53], stochastic flow methods[59], multi-level methods[45, 23] have all been used for discovering clusters of interest. Some graph visualization works have been proposed to reveal cluster structures within graphs. Abello et al.[1] propose a graph visualization system which uses clustering to construct a hierarchy of large scale graphs. Gronemann et al [37] propose to use topographic map to visualize hierarchical clustering structures within a graph – each mountain corresponds to a cluster. Batagelj et al. [11] propose a hybrid visualization method (combining node-link and matrix-based representations) to visualize graph clusters. Didimo et al. [24] propose a fast method to layout graph by combining space-filling and fast force-directed methods, vertices that are close in graph topology will be positioned together in the layout, so the clusters of graph can be easily identified.

Dense subgraph is a subgraph pattern in which vertices densely connected to each other, it can be viewed as a special cluster. The most common dense subgraph pattern is clique. In spite of the fact that CLIQUE problem is NP-Hard[32], and approximating the size of the largest clique in a graph is almost NP-complete[29], mining cliques for a graph has received much attention recently. The CLAN method [73] for example, aims to mine exact cliques in large graph datasets, CLAN uses the canonical form to represent a clique, and the clique detection task becomes mining strings representing cliques. Some other methods[2, 83] have been proposed to detect quasi-clique, which is a clique with some edges missing.
Other clique-like dense subgraph patterns, such as DN-graph [75], are also proposed. Wang et al. [74] propose CSV to visualize approximate cliques. CSV uses a notion of local density, co-clique size, and plots all vertices based on co-clique sizes. The plot is a OPTICS [5] style plot, and visualizes the distribution of all the potential cliques. Another important dense subgraph structure is K-Core, and it has been used to detect communities in directed graphs [34], and uncovering the structure of internet network [4]. Alvarez-Hamelin et al. [3] propose a visualization tool to visualize K-Cores in large graphs, the layout of the graph is composed by a series of concentric circular shells, and each shell contains vertices of the same core value, these shells reveal the hierarchial structure of K-Cores.

Network scientist is often interested in measures/attributes on both nodes and edges of the graph [87, 80]. There has been some recent research on this front. For example, Athenstadt et al. [8] visualize the hierarchical clusters of a graph as 3D landscape, and use height to indicate the numerical attribute of each vertex. Bourqui et al. [16] visualize graphs with weights by using Voronoi diagram to layout the hierarchial clusters on a 2D plane. Bezerianos et al. [15] propose a visual system to explore social networks with multiple attributes. Some other works [44, 30] use matrix or histogram to analyze attribute information in graphs.

Analyzing dynamically changing graphs also draws much attention in recent works. Leskovec et al. [48] study the topological properties of some evolving real-world graphs, and propose “forest fire” spreading process including these properties. Backstrom et al. [9] study the relation between the evolution of communities and the structure of the underlying social networks. Asur et al. [7] define several events based on graph clusters evolution, and analyze group behavior through these events. Sun et al. [65] present a non-user-defined parameters approach to cluster evolving graphs based on Minimum Description Length.
principle. Lin et al. [49] propose FacetNet framework to detect community structure both by the network data and the historic community evolution patterns.

Some visualization methods have been proposed to analyze dynamic graphs. Diel et al. [25] propose a foresighted layout for dynamic graphs so the mental map is preserved. They first compute a global layout of the dynamic graph, then the layout of each snapshot of the dynamic graph is induced from the global layout. Brandes et al. [17] represent a dynamic graph in a 3D space, and use a layer to indicate the changes between consecutive snapshots, this enables users simultaneously examine the state of a network and the changes. Rufiange et al. [58] propose a hybrid representation of dynamic graphs, it displays the dynamic graph in a sequence of tiles, where each tile can be a snapshot, or the difference between two snapshots, or the animation clip that interpolates between two snapshots.

1.2 Challenges

The first challenge of graph visualization is scalability. The size of many real world networks is growing fast, several social networks (e.g. Facebook, Gmail) have reached hundreds of millions of nodes. How then to efficiently visualize the large graphs and let users quickly capture the important information is a great challenge. Most graph visualization methods first analyze the graph to extract the useful information, and then visualize the extracted information through visual methods. The scalability issue could arise during the stage of extracting information from graph – the graph analysis algorithm can be slow. Many graph clustering or subgraph mining algorithms produce accurate results at a great time cost, it is hard to incorporate these methods in visualization tool because the long waiting time can hurt user experience. The scalability issue could also arise during the stage of displaying information on a screen. The screen size is limited, and there can be a
large amount of information to display, how to organize the information on the screen so
users can quickly understand the essence of it is an challenging question.

The second challenge is to visualize graphs with attributes. Such attributes can be
computed from graph topology, and often encode information about connectivity - both
locally (e.g. clustering coefficient [76], K-Core [12], K-Truss [72]) as well as globally (e.g.
various centrality measures [52] such as closeness, betweenness and harmonic, and also
measures of importance such as PageRank [18] and Influence [68]). Beyond just attributes
from the topology, such attributes may also incorporate heterogeneous information related
to content (e.g. sequence information of a protein etc.). Many visualization tools use node
color/size to represent node attributes, this does not work well for visualizing large graphs,
because while visualizing large graphs, it is hard to draw every node/edge on screen. In
such cases summarization tools are usually employed, however, the attribute information
can be lost or inaccurate.

The third challenge relates to the dynamic nature of the data, or changes to the network.
One question is how to effectively visualize the dynamic graph so users can quickly iden-
tify the portions of the network that are changing, and characterize the type of change. As
graph size increases, many change events could occur to the graph at the same time, the
visualization method should be able to highlight the important change events so users can
easily capture them. Another interesting question is, many graphs analysis/visualization al-
gorithms often require re-computation from scratch when graph changes, this can be very
expensive and hurts user experience. So we need to design graph analyses/visualization al-
gorithms that can efficiently update the previous generated results while graph changes.
Moreover, besides analyzing changing aspect of the dynamic graph, users might also be in-
terested in analyzing the stable part of the dynamic graph, For example, detecting subgraph
pattern that occurs periodically, or detecting clusters that stay stable during the evolution of graph. Most previous works focus on the changing part of the dynamic graph while ignoring the stable part.

1.3 Thesis Statement

Large scale graph visualization is becoming increasingly complicated with the widening gap between the amount of data that has to be visualized, available pixel space and cognitive abilities of humans in-the-loop. In this thesis we provide mechanisms to alleviate this challenge by leveraging the idea of density plots and extending them to a terrain metaphor while accommodating the dynamics inherent in such problems.

1.4 Our Contributions

- We design a new clique-like structure, Triangle K-Cores, to approximate cliques, and develop a simple algorithm for computing Triangle K-Cores from graphs. In our experiments we demonstrate its efficiency by several factors over competing strategies such as DN-Graph[75] and CSV[74]. We also show that in most cases Triangle K-Cores are very close to cliques, and can substitute for cliques in some applications.

  Based on Triangle K-Cores, we propose a parallel method to compute approximate cliques from large graphs. We further speed up the parallel method by a sparsification mechanism. The experimental results show that the parallel approximate clique detection method is very scalable on large graphs, and as the number of computing nodes increases, the time cost is reduced significantly.

To detect Triangle K-Cores in dynamic graphs, we present an incremental variant of Triangle K-Core algorithm that can update the previous results while local change
occurs in graph, it is more efficient than recomputing from scratch. Also we present a few visualization methods that can be used to capture the major changes of Triangle K-Cores in dynamic graphs.

- We propose a 3D terrain visualization method to explore graphs with numerical attributes associated with nodes (or edges) – referred to as scalar graphs, and the attribute values are referred to as scalar values. Each scalar value could either be a natural attribute or a derived attribute summarizing information from multiple natural attributes (e.g. triangle density, centralities, cliquishness). We model the problem as a continuous function $f : X \rightarrow \mathbb{R}$: the domain $X$ is a simplicial complex whose vertex set is the set of input graph nodes, and its topology is determined by the input graph topology. We call such a representation of a graph a graph-induced scalar field. We then borrow the powerful “terrain metaphor” idea for visualizing scalar fields and adapt it to the graph context. In the final terrain visualization, vertices of the same scalar value will be in the same height, and if they are connected in the graph, they will be in the same “mountain” of the terrain. Intuitively, a high mountain represents a cluster of vertices of high scalar values. Based on our visual platform, we also propose methods to analyze the relation between multiple numerical measures on one graph. We evaluated our terrain visualization by a user study, in which we ask 10 users to perform 3 tasks. The results show that compared with two other visualization methods, users can better capture the overall topological and scalar value information through our terrain visualization.

- We propose analyze the dynamic graph through bitstreams representations – every edge is associated with a bitstream to indicate its appearance in every snapshot.
Based on the bitstream representation, we analyze the dynamic graph in the following two aspects:

1. Detecting Periodicities in the Dynamic Graph: The bitstream of each edge can be considered as a time series signal, in this work we apply two popular time series analysis methods (Discrete Fourier Transform and Autocorrelation) to bitstreams to identify major periods in a dynamic graph. Then based on the detected major periods, we propose an algorithm to detect communities that appear periodically in the dynamic graph. The periodicities and periodic communities can be viewed as stable part of the dynamic graph, and very few previous works have paid attention to this part.

2. Detecting Topologically and Temporally Correlated Edges: In the context of dynamic graph, only considering topological features may be insufficient. For example, we might be interested in clusters of vertices/edges that are not only topologically close but also temporally correlated (i.e. co-occur in many snapshots). In our work, we identify the topologically and temporally correlated edges by using the terrain visualization. We convert the dynamic graph to a weighted graph and apply the terrain visualization to visualize the weighted graph, the terrain will highlight the clusters of edges in the dynamic graph that are both topologically and temporally correlated. Based on the terrain visualization, we define a “dense block” which indicates a dense subgraph that frequently occurs in a certain period, and then we develop an efficient method to detect “dense blocks” in the bitstreams.

We summarize the contributions of the our visualization framework in Figure 1.1.
1.5 Organization

The rest of the dissertation is organized as follows. In Section 2 we propose our method to detect and visualize Triangle K-Cores in graphs, and extend Triangle K-Cores to analyze dynamic graphs. In Section 3 we provide a fast parallel method to detect approximate cliques in graphs. In Section 4 we introduce the terrain visualization method to visualize graphs with numerical attributes. In Section 5 we introduce the bitstream representation of dynamic graphs, and propose some visual methods to extract periodicities and clusters from bitstreams. In Section 6 we conclude with some suggested directions for future research.
Chapter 2: Extracting Analyzing and Visualizing Triangle K-Core Motifs within Networks

2.1 Introduction

One important task in graph mining is to probe, uncover, and understand the evolution of dense structures (communities or cliques) within such networks. Here we propose a scalable visual-analytic framework, for probing and uncovering dense substructures within networks. Central to our approach is the novel notion of a Triangle K-Core motif. We develop a simple algorithm for computing Triangle K-Cores from graphs.

We then discuss a mechanism to plot such Triangle K-Cores – essentially realizing a density plot in a manner analogous to a CSV plot[74]. This plot follows an Optics[5]-style enumeration of vertices in the network. The keynote [63] pointed out that density plot is an effective way to visualize aggregated and abstracted data, and could facilitate interactive visual exploration of very large dataset, because it shows concentrations of important information.

The proposed algorithm is provably efficient on several real world scale-free (sparse) social and biological networks. In fact as our experimental results show, we produce plots that are very similar to CSV at a fraction of the cost. Moreover, our empirical results suggest that Triangle K-Cores motifs, can be used as preprocessing step for detecting exact cliques as demonstrated elsewhere[74].
Subsequently we extend the above static algorithm to handle dynamic graphs. A key challenge addressed here is that of cognitive correspondence – the same community in two different density plots must be clearly identified as long as the local relationship structure has not changed significantly. We develop a suitable incremental algorithm, with cognitive correspondence (by relying on an adaptation of dual-view plots), which we show to be significantly faster than the naive approach which recomputes Triangle K-Cores from scratch.

An additional feature of our algorithms is the ability for the domain expert to dynamically specify, explore and probe the network for various user-defined template patterns of interest defined upon the Triangle K-Core. Such template patterns can be extremely informative. We design and adapt our density plot framework (here density is defined by the density of the template pattern of interest) based on this notion and discuss several applications of this work on real world datasets. To sum up, the main contribution of our work is to introduce a new motif for estimating clique like structure in graphs (Triangle K-Core). Specifically in this article we demonstrate its:

1. **Utility:** We demonstrate its use for visualization (in a manner similar to a CSV plot), probing, exploring and highlighting interesting patterns in both static as well as dynamic graphs. We compare its utility with respect to recent state of the art alternative (e.g. CSV[74] and DN-Graph[75] motifs.

2. **Efficiency:** We present a localized algorithm for extracting such motifs and demonstrate its efficiency by several factors over competing strategies such as DN-Graph[75] and CSV[74]. Additionally, we present an incremental variant that can be extended to handle dynamic graphs with much lower cost than the iterative method[75] and global method[74] used by extant approaches.
3. **Flexibility**: An important feature of the Triangle K-Core motif is its inherent simplicity which lends itself to flexible probing of user-defined pattern cliques of interest within both static and dynamic graphs.

### 2.2 Preliminaries

Given a graph $G = \{V, E\}$, $V$ is the set of distinct vertices $\{v_1, \ldots, v_{|V|}\}$, and $E$ is the set of edges $\{e_1, \ldots, e_{|E|}\}$. A graph $G' = \{V', E'\}$ is a subgraph of $G$ if $V' \subseteq V$, $E' \subseteq E$.

The Triangle K-Core subgraph proposed in this paper is derived from K-Core subgraph, and we explain and compare them as follows.

**Definition 1.** A **K-Core** is a subgraph $G'$ of $G$ that each vertex of $G'$ participates in at least $k$ edges within the subgraph $G'$. The **K-Core number** of such a subgraph is $k$.

**Definition 2.** The **maximum K-Core** associated with a vertex $v$ is defined by the subgraph $G_v$ containing $v$ whose K-Core number is the maximum from among all subgraphs containing $v$. The K-Core number of $G_v$ is the **maximum K-Core number** of $v$.

Batagelj et al [12] propose an efficient method to compute every vertex’s maximum K-Core number with $O(|E|)$ time complexity.

Based on definition of K-Core, we are now in a position to define the notion of a Triangle K-Core:

**Definition 3.** A **Triangle K-Core** is a subgraph $G'$ of $G$ that each edge of $G'$ is contained within at least $k$ triangles in the subgraph. Analogously, the **Triangle K-Core number** of this Triangle K-Core is refered to as $k$.

**Definition 4.** The **maximum Triangle K-Core** associated with an edge $e$ is the subgraph $G_e$ containing $e$ that has the maximum Triangle K-Core number. Analogously, the Triangle
K-Core number of $G_e$ is the **maximum Triangle K-Core number** of edge $e$. We use $\kappa(e)$ to denote the maximum Triangle K-Core number of edge $e$.

The main advantage of a Triangle K-Core over a K-Core is that it offers a natural approximation of clique, we illustrate this in the Figure 2.1.

![K-Core Number = 2](image1)

![Triangle K-Core Number = 2](image2)

**Figure 2.1: K-Core vs. Triangle K-Core**

Figure 2.1(a) is a 5-vertex K-Core with K-Core number 2 constructed by minimal number of edges, Figure 2.1(b) is a 5-vertex Triangle K-Core with Triangle K-Core number 2 constructed by minimal number of edges, and we can easily see that the Triangle K-Core is much closer to a 5-vertex clique than the K-Core. In fact, Triangle K-Core is a relaxation of clique, a $n$-vertex clique is equivalent to a $n$-vertex Triangle K-Core with Triangle K-Core Number $n-2$.

The Triangle K-Core motif is based on triangles of each edge, the intuition is, for example, an edge participating in 4 triangles implies a subgraph of 6 nodes and 9 edges (in the worst case). A node participating in 4 triangles does not say much about its local density (worst case, it could involve 9 nodes and 12 edges – hub-pattern). The former is closer to a clique structure (9/15, density: 60%) than the latter (12/36, density:33%). Note that a
Triangle K-Core makes an even stronger assertion on density (every edge in the subgraph is contained within at least k triangles).

For edge $e_t$, in $e_t$’s maximum Triangle K-Core there is a triangle $T$ containing $e_t$, such that we have the following property for $T$:

**Theorem 1.** If triangle $T$ is in $e_t$’s maximum Triangle K-Core, and contains three edges, $e_t, e_1$ and $e_2$, then $\kappa(e_i) \geq \kappa(e_t) \ (i = 1, 2)$.

**Proof.** Since edge $e_i$ is in triangle $T$, and $T$ is in $e_t$’s maximum Triangle K-Core, denoted as $G_{e_t}$, we have subgraph $G_{e_i}$ contains $e_i$. According to Definition 4, $e_i$’s maximum Triangle K-Core should have Triangle K-Core number no less than $G_{e_t}$’s Triangle K-Core number, that is $\kappa(e_i) \geq \kappa(e_t)$.

2.3 Triangle K-Core Algorithm

2.3.1 Detecting Maximum Triangle K-Core

In Algorithm 1, input is Graph $G$, output is the maximum Triangle K-Core number and optionally the maximum Triangle K-Core associated with each edge. In each iteration, this algorithm processes a particular edge $e_i$ and determines its maximum Triangle K-Core number.

Before describing the Algorithm 1 we define the notions of processing an edge and a triangle. If an edge’s maximum Triangle K-Core number has been determined, it is considered to be processed. A triangle $T$ is processed if any one of its edges is processed.

In step 2, each edge is set to unprocessed. In step 3, each triangle on edge $e$ is constructed by $e$’s two vertices and one common neighbor of them. One triangle could be constructed three times by its three edges, but we only store one instance of each triangle,
Algorithm 1: Detect each edge’s maximum Triangle K-Core

1: for each edge \( e \) in the graph do
2:   set \( e \) to be unprocessed;
3:   find all the triangles on \( e \), set them to be unprocessed;
4: for each triangle \( t \) on edge \( e \) do
5:   AddToCore\((t, e)\);
6: \( \tilde{\kappa}(e) += \) \\
7: Place all the edges in list \( Edges \), sort them in increasing order of \( \tilde{\kappa} \) value;
8: for \( i = 0 \) to \( |E| - 1 \) do
9:   \( e_i = Edges[i] \);
10: \( \kappa(e_i) = \tilde{\kappa}(e_i) \);
11: for each unprocessed triangle \( T \) on \( e_i \) do
12:   for each edge \( e_t \) other than \( e_i \) in \( T \) do
13:     if \( \tilde{\kappa}(e_t) > \tilde{\kappa}(e_i) \) then
14:       DelFromCore\((T, e_t)\);
15: \( \tilde{\kappa}(e_t) -= \) \\
16:       update \( e_t \)'s position in the sorted list \( Edges \);
17: set triangle \( T \) to be processed;
18: set \( e_i \) to be processed;

by giving a unique id to each edge and only creating a triangle instance on its edge with smallest id. Note that all triangles on edge \( e \) could be in \( e \)'s maximum Triangle K-Core, so in step 5 the algorithm (AddToCore), updates its bookkeeping to reflect the fact that each triangle \( t \) is possibly in \( e \)'s maximum Triangle K-Core. Finally, \( \tilde{\kappa}(e) \) contains the upper bound of \( e \)'s maximum Triangle K-Core number \( \kappa(e) \).

In step 7 we place all the edges in a list sorted by increasing order of \( \tilde{\kappa} \) value. Bucket sort can be used as an optimization step here with time complexity \( O(|E|) \). In steps 8-18, we process each edge \( e_i \) and determine its exact maximum Triangle K-Core number \( \kappa(e_i) \) since thus far we only had an upper bound. In step 10, we determine that \( \kappa(e_i) \) is exactly \( \tilde{\kappa}(e_i) \), the correctness is proved later. Then we update \( e_i \)'s neighbor edges’ \( \tilde{\kappa} \) value in steps 11-17. If an unprocessed triangle \( T \) on \( e_i \) contains edge \( e_t \) that \( \tilde{\kappa}(e_t) \) is greater than \( \tilde{\kappa}(e_i) \) (step 13), we delete \( T \) from the upperbound of \( e_i \)'s maximum Triangle K-Core.
DelFromCore updates its bookkeeping to indicate that $T$ is not in the upperbound of $e_i$’s maximum Triangle K-Core. In step 16, based on bucket sort the update could be optimized with complexity $O(1)$.

In fact, steps 5 and 14 are not necessary here, but it will be useful for dynamic update algorithms. The time complexity for steps 1-7 is $O(\sum(d_i^2))$, $d_i$ is the degree for node $i$, $i=1,2...|V|$. The time complexity for Steps 8-18 is $O(|Tri| + |E|)$, where $|Tri|$ is the total number of triangles in the graph.

![Figure 2.2: Examples for Illustrating Triangle K-Core Algorithms](image)

**Example:** Figure 2.2(a) is an example to illustrate Algorithm 1. We find the triangles on each edge, and sort edges in increasing order of $\tilde{\kappa}$ value, $\{AB(1), AC(1), BD(2), BE(2), CD(2), CE(2), DE(2), BC(3)\}$, where the number in parenthesis indicates the $\tilde{\kappa}$ value of the edge. We process AB first, and get $\kappa(AB)=1$. For unprocessed $\triangle ABC$ on AB, $\tilde{\kappa}(BC)=3$ is greater than $\tilde{\kappa}(AB)=1$, so $\tilde{\kappa}(BC)$ decrease 1 to be 2 (step 15), and $\triangle ABC$ becomes processed. Then we process edge AC, and have $\kappa(AC)=1$, there is no unprocessed triangle on AC, so no update is needed. Next we process edge BD, and get $\kappa(BD)=2$, $\triangle BDC$ and $\triangle BDE$ on BD are unprocessed, but no edge of the two triangles has greater $\tilde{\kappa}$ value than $\tilde{\kappa}(BD)$, so no update. In the same way we find all left edges having $\kappa$ value equals 2.
Proof of Correctness of Algorithm 1: We show the following invariances of Algorithm 1: at the end of each iteration $i$, (1) for the edge $e_t$ whose $\tilde{\kappa}(e_t)$ value updated, $\tilde{\kappa}(e_t)$ is still the upperbound of $\kappa(e_t)$; (2) for the edge $e_i$ processed in current iteration, $\tilde{\kappa}(e_i)$ is equal to $\kappa(e_i)$.

We firstly prove the invariance (1) of Algorithm 1. In steps 11-12, for an unprocessed triangle $T$ on edge $e_i$, all $T$’s edges are unprocessed, so $T$ is still in the upperbound of maximum Triangle K-Cores of all its edges(including edge $e_i$ and $e_t$). If $\tilde{\kappa}(e_t) > \tilde{\kappa}(e_i)$ (step 13), we have:

Claim 1. $\tilde{\kappa}(e_t) > \kappa(e_t)$

Proof. We prove by contradiction. Assume $\tilde{\kappa}(e_t) = \kappa(e_t)$, then all the triangles in the current upper bound of $e_t$’s maximum Triangle K-Core are exactly in $e_t$’s maximum Triangle K-Core, so $T$ is in $e_t$’s maximum Triangle K-Core. However, in triangle $T$, $\kappa(e_t) = \tilde{\kappa}(e_t) > \kappa(e_i) = \kappa(e_t)$, which violates Theorem 1, so the assumption is incorrect. We have $\tilde{\kappa}(e_t) > \kappa(e_t)$.

According to the proof of Claim 1, after decreasing $\tilde{\kappa}(e_t)$ by 1 (step 15), $\tilde{\kappa}(e_t)$ still remains as the upper bound of $\kappa(e_t)$. So invariance (1) is held.

Now we prove invariance (2). In iteration i, assume $\tilde{\kappa}(e_i) = k$, we use the edges whose current $\tilde{\kappa} \geq k$ to construct a subgraph $G_k$ (including $e_i$), and have the following claim:

Claim 2. The subgraph $G_k$ is a Triangle K-Core with Triangle K-Core number $k$.

Proof. For any edge $e$ in $G_k$, $\tilde{\kappa}(e) \geq k$, so the upper bound of $e$’s maximum Triangle K-Core now contains at least $k$ triangles. Assume triangle $T$ is one of them, considering $T$’s two other edges $e1$ and $e2$, if $e1$ is not in subgraph $G_k$, then $\tilde{\kappa}(e1) < k$. We could see that Algorithm 1 processes edges in increasing order of $\tilde{\kappa}$, so $e1$ should already be processed.
When processing $e_1$, $\tilde{\kappa}(e_1) < \tilde{\kappa}(e)$ (step 13) is true, so triangle $T$ should be deleted from the upper bound of $e$'s maximum Triangle K-Core (step 14), which is a contradiction to the assumption that triangle $T$ is in upper bound of $e$'s maximum Triangle K-Core. So $e_1$ is in subgraph $G_k$, and so is $e_2$. Because edges $e$, $e_1$ and $e_2$ are all in subgraph $G_k$, triangle $T$ is in $G_k$. So all the triangles now in upper bound of $e$'s maximum Triangle K-Core are in subgraph $G_k$, which means any $e$ in $G_k$ is contained in at least $k$ triangles in $G_k$, so $G_k$ is a Triangle K-Core with Triangle K-Core number $k$.

In Claim 2, we have a subgraph $G_k$ containing $e_i$ with Triangle K-Core number equals $\tilde{\kappa}(e_i)$, so $\tilde{\kappa}(e_i)$ is exactly $\kappa(e_i)$, invariance (2) is held, and $G_k$ is obviously the maximum Triangle K-Core of $e_i$.

In step 3 we could store all triangles in main memory, then reuse them in step 11. However for a large graph, storing all triangles in main memory might be impossible. In such a case, we do not store triangles in step 3, and compute each edge’s triangles again in step 11, then we test whether a triangle is unprocessed by testing whether its three edges are all unprocessed.

### 2.3.2 Updating Maximum Triangle K-Core

So far we have worked on static graphs. In scenarios when edges are added and removed from a graph over time however, rather than recomputing the Triangle K-Cores from scratch after each change, we can use Algorithm 2 to efficiently update edges’ maximum Triangle K-Cores. The detailed pseudo code of Algorithm 2 is in Section 2.7.1.

Adding/deleting one edge might add/delete multiple triangles simultaneously, in Algorithm 2 we process added/deleted triangles one by one (step 1). Initially all added/deleted triangles are not updated, and when processing one triangle $T$ we set it to be updated (step
2). In step 3, we identify $T$’s edges whose maximum Triangle K-Cores might change, and store them in $\text{PotentialList}$. We use Rule 0 to help find the edges whose maximum Triangle K-Cores might change. Rule 0 is derived from Theorem 1, the proof is omitted for brevity.

- Rule 0: when triangle $t$ is added/deleted to graph $G$, assume $\mu$ is smallest $\kappa$ value of $t$’s three edges, then only the edges in $G$ whose $\kappa$ value equals $\mu$ might have their maximum Triangle K-Cores changed.

Then we process each edge $e$ in $\text{PotentialList}$ to update its $\kappa(e)$. All the triangles associated with edge $e$ should obey Theorem 1, so we process them based on Theorem 1 (steps 6-7). If $\kappa(e)$ finally changes, we put $e$ in $\text{ChangingList}$, which stores edges whose $\kappa(e)$ has been changed, and put $e$’s neighbor edges whose maximum Triangle K-Cores might change to $\text{PotentialList}$(step 8). We use Rule 0 to help select the edges to be put in $\text{PotentialList}$.

After processing all edges in $\text{PotentialList}$, we could determine edges’ maximum Triangle K-Core numbers in $\text{ChangingList}$(step 9).

Please note that if an added triangle is not updated, or a deleted triangle is updated, we do not involve them in the Algorithm 2. A brief illustration of Algorithm 2 is as follows.

**Algorithm 2 Update maximum Triangle K-Cores**

1: for each added/deleted triangle $T$ do
2:   Set $T$ to be updated;
3:   Put $T$’s edges whose maximum Triangle K-Cores might change to $\text{PotentialList}$;
4:   Add/delete $T$ from the maximum Triangle K-Cores of edges in $\text{PotentialList}$, update those edges’ $\kappa$ value;
5:   for each edge $e$ in $\text{PotentialList}$ do
6:     Find $e$’s “illegal” triangles that violate Theorem 1;
7:     Process $e$’s “illegal” triangles to obey Theorem 1, meanwhile update $\kappa(e)$;
8:     If $\kappa(e)$ changes, put $e$ in $\text{ChangingList}$, put $e$’s neighbor edges whose maximum Triangle K-Cores might change to $\text{PotentialList}$;
9:   update $\kappa(e)$ of each edge $e$ in $\text{ChangingList}$;
**Example:** In Figure 2.2(b), the original graph is comprised with solid edges, and edge AC is added. The original $\kappa$ value for each edge is \{AB(0), BC(0), AE(1), AF(1), EF(1), CD(1), CE(1), DE(1)\}. The initial value for $\kappa(AC)$ is 0. After adding edge AC, two triangles are added, $\triangle ABC$ and $\triangle AEC$.

Firstly, we process newly added $\triangle ABC$, now all its three edges are \{AB(0), BC(0), AC(0)\}, so we put all three edges in PotentialList (Rule 0), and add $\triangle ABC$ to their maximum Triangle K-Cores (step 4), their $\kappa$ value increases to be 1. Then we process each edge in PotentialList, assume AC is the first edge. In step 6 we find $\triangle ABC$ on edge AC is “legal”, and $\triangle AEC$ is not taken into consideration because it is not updated. In step 8, because $\kappa(AC)$ changes to be 1, we put edge AC’s neighbor edges AB, BC in PotentialList (they are already in). In the following iterations we process left edges in PotentialList (AB and BC) similarly, and update $\kappa(AB)$ and $\kappa(BC)$ to be 1.

Then, we process newly added $\triangle AEC$, now its three edges are \{AE(1), EC(1), AC(1)\}, so we put all of them in PotentialList, and add $\triangle AEC$ to their maximum Triangle K-Cores, their $\kappa$ value increases to be 2. Let’s process edge AC first, we find $\triangle ABC$ on edge AC is “illegal”, because $\triangle ABC$ is in AC’s maximum Triangle K-Core while $\kappa(AC) = 2$ is greater than $\kappa(BC) = 1$ and $\kappa(AB) = 1$, which violates Theorem 1. So in step 7 we delete $\triangle ABC$ from AC’s maximum Triangle K-Core and decrease $\kappa(AC)$ to be 1. Similarly edges AE and EC in PotentialList both are processed to decrease $\kappa(AE)$ and $\kappa(EC)$ to be 1.

The proof of correctness of Algorithm 2 and Rule 0 is in our technical report[84]. If we do not store triangles in Algorithm 1, then in Algorithm 2 we need to recompute triangles from edges, we explain this in Section 2.7.1.
2.4 Extensions

**Visualizing Clique-like Structures:** We now describe how Triangle K-Cores can be used for detecting and visualizing interesting clique-like structures within networks. Before describing our technique we briefly review the CSV method [74] to visualize all potential cliques in graph.

*CSV plot:* CSV first estimates *co clique size* for each edge, which is the size of the maximum clique that each edge participates in. Then subsequently CSV plots vertices along X-axis in a certain order, and the Y-axis value for each vertex is one of its neighbor edges’ *co clique size* value. The final plot is the clique distribution of the graph, and the flat peaks in the plot indicate potential cliques.

However, estimating *co clique size* for each edge takes up most of the time cost in CSV. Instead we propose to use each edge’s maximum Triangle K-Core as a proxy to approximate the maximum clique it participates in. Since the maximum clique among a subgraph with Triangle K-Core Number $\kappa$ is a $(\kappa + 2)$-vertex clique, we estimate $e.co clique size$ as $\kappa(e) + 2$ for each edge $e$, and then plot the clique distribution using the same method as that of CSV. As we demonstrate in experiments our method produces plots that are inherently similar or identical to that of CSV at a fraction of the cost.

**Dual View Plots:** In a graph $G$ that evolves over time, when edges are added to it, some clique structures in $G$ might change. We propose Dual View Plots to analyze how clique structures in $G$ change over time.

The idea is: for one snapshot $G_a$ of graph $G$, we plot all its cliques in plot(a). After $G_a$ evolves to be snapshot $G_b$ by adding new edges, in plot(b) we plot the cliques of $G_b$ that contain new edges, these cliques should not exist in $G_a$, and they are usually formed by merging/expanding cliques in $G_a$. By comparing plot(a) and plot(b), we can visually
analyze how cliques in plot(b) are formed from cliques in plot(a). We use the same plot method as CSV to plot clique distribution. The detailed steps are presented in Algorithm 3. We illustrate the benefits of Dual View Plots in the Section VII Experiments.

Algorithm 3 Dual View Plots

1: Execute Algorithm 1 to compute $\kappa(e)$ for each edge $e$ in $G_a$;
2: For each edge $e$ in $G_a$, $e.co_clique_size = \kappa(e) + 2$;
3: Plot clique distribution of $G_a$ (plot(a));
4: After $G_a$ evolves to be $G_b$ by adding new edges, execute Algorithm 2 to update $\kappa(e)$ for each edge $e$ in $G_b$;
5: For each edge $e$ in $G_b$, if $e$ is newly added edge, $e.co_clique_size = \kappa(e) + 2$, otherwise $e.co_clique_size = 0$;
6: Plot clique distribution of $G_b$ (plot(b)) based on $co_clique_size$ calculated in step 5;
7: In plot(b) select one Clique $C$ of interest, locate the corresponding vertices of $C$ in plot(a), and analyze how $C$ is formed;

Detecting Template Pattern Cliques: In this section we describe a method which allows users to detect cliques of patterns of their interest, which we call template pattern cliques. For example, in one snapshot of a graph that evolves over time, template pattern cliques might be cliques formed by merging two cliques in a previous snapshot, or by augmenting a clique in previous snapshot. Such cliques can allow a user to probe an evolving network to discover interesting or anomalous behavior[54]. The end-goal of our method is to allow the user the flexibility to specify what patterns are of interest to her/him in the context of the domain.

Several examples of template pattern cliques in evolving graphs are illustrated in Figure 2.3. The previous snapshot of the graph is denoted as $G_{old}$, the current snapshot is denoted as $G_{new}$. In Figure 2.3 black vertices/edges are old vertices/edges, i.e., vertices/edges in $G_{old}$, red vertices/dashed-lines are newly added vertices/edges in $G_{new}$. The template pattern cliques defined below are all in $G_{new}$.  


1. An *Emerging Clique* is formed by connecting old vertices with newly added edges. In Figure 2.3(a) ABCDE is an *Emerging Clique*.

2. A *Bridge Clique* is formed by connecting two disconnected cliques in $G_{old}$ with newly added edges. In Figure 2.3(b) ABCDE is a *Bridge Clique*.

3. An *Expanding Clique* is formed by augmenting a clique in $G_{old}$ with newly added vertices and edges. In Figure 2.3(c) ABCDEF is an *Expanding Clique*.

![Image of cliques and characteristic triangles](image.png)

**Figure 2.3:** Several *template pattern cliques* and their characteristic triangles

We propose Algorithm 4 to detect and extract the *template pattern cliques* of interest. We first define the notion of a *characteristic triangle* within an evolving network. The vertices and edges of a characteristic triangle are labeled as *new* (red) or *old* (black), as defined above. Two labeled characteristic triangles are of the same type if they are isomorphic. A *template pattern clique* is identified uniquely with a single characteristic triangle type (see Figure 2.3 for examples), and every vertex (this does not hold for every edge as we shall...
clarify shortly) within a template pattern clique of interest will participate in at least one characteristic triangle of the given type (again see Figure 3). Thus the vertices of all template pattern cliques are a subset of the vertices of all characteristic triangles of the given type.

We note that besides characteristic triangles, other types of triangles can also occur within template pattern cliques – we call these possible triangles, and they account for the edges that do not occur within characteristic triangles (e.g., edge AB in Figure 2.3(c)). Obviously the vertices of these possible triangles are among the vertices of characteristic triangles.

Thus identifying all characteristic triangles and possible triangles of the given type within the evolving network will cover all the vertices and edges in the template pattern cliques, and plotting their density plot (using Triangle K-Core) will ensure the complete detection and extraction of relevant template pattern cliques. Note that such a density plot will now highlight the regions of the network where the densest template clique patterns of interest are found as opposed to simply the densest clique structures. In the following we specify the characteristic triangles and possible triangles of the three template pattern cliques introduced before.

Detect Emerging Cliques: the characteristic triangle of an Emerging Clique has 3 new edges and 3 old vertices, as illustrated in Figure 2.3(d), and no possible triangles are in Emerging Cliques.

Detect Bridge Cliques: the characteristic triangle of Bridge Clique has 3 old vertices, 2 new edges, and 1 old edge, as illustrated in Figure 2.3(e). We find that in Bridge Clique there is one type of possible triangle, which is comprised of 3 old edges and 3 old vertices, such as $\triangle BCD$ in the Figure 2.3(b).
Detect Expanding Cliques: the characteristic triangle of Expanding Clique contains 1 new vertex, 2 old vertices, 2 new edges, 1 old edge, as Figure 2.3(f) shows. There are two types of possible triangles in Expanding Clique. One type is made of all new edges, such as $\triangle ABC$ in Figure 2.3(c), and another type is made of all old edges, such as $\triangle DEF$ in Figure 2.3(c).

Algorithm 4 Detecting template pattern cliques in Graph $G$

1: Define and detect the characteristic triangles of the template pattern cliques;
2: for each characteristic triangle $T_c$ do
3:    Mark $T_c$’s edges and vertices as selected;
4: Define and detect the possible triangles formed by selected vertices;
5: for each possible triangle $T_p$ do
6:    Mark $T_p$’s edges as selected;
7: Extract the subgraph $G_{sel}$ built by selected vertices and selected edges;
8: Execute Algorithm 1 on $G_{sel}$ to calculate each selected edge’s $\kappa$ value;
9: for each edge $e$ in $G$ do
10:    if $e$ is a selected edge then
11:        $e.co_clique_size = \kappa(e)+2$;
12:    else
13:        $e.co_clique_size = 0$;
14: Use the same plot method as CSV to plot clique distribution of graph $G$;

In steps 2-3 of Algorithm 4 we mark all edges and vertices of characteristic triangles to be selected. In step 4 we define and detect all these possible triangles. In steps 5-6 we mark all possible triangles’ edges as selected. In step 7, we build a subgraph $G_{sel}$ made of selected edges and selected vertices. In step 8 we execute Algorithm 1 on $G_{sel}$. In steps 9-13 we compute $co_clique_size$ for selected edges, and set $co_clique_size$ of non-selected edges to be 0, because they do not participate in any template pattern cliques. Finally we plot the distribution of the template pattern cliques.
The overall complexity of Algorithm 4 depends on the triangles on new edges and is hard to estimate, the worst case is $O(|\text{Tri}|)$, where $|\text{Tri}|$ is the total number of triangles in the graph snapshot $G_{\text{new}}$.

Please note that Algorithm 4 not only works for evolving graphs, but also for static graphs in which edges and vertices have different labels. In Section VII Experiments we will illustrate detecting template pattern cliques on both static and dynamic graphs.

### 2.5 Relationship to DN-Graph

Before we discuss the empirical evaluation we would like to highlight an interesting connection between our approach and the recent approach proposed by Wang et al.[75]. It is interesting to note that this connection was initially observed during our empirical evaluation, where we found both DN-Graph and our method converge to identical values of $\text{co\_clique\_size}$ (density). We are now in a position to also provide a theoretical justification for this connection.

DN-Graph $G'(V', E', \lambda)$ is a subgraph pattern proposed by Wang et al.[75], it satisfies two requirements

1. every connected pair of vertices in $G'$ has at least $\lambda$ common neighbors; 2. for a vertex $v$ not in $G'$, adding $v$ to $G'$ will decrease the $\lambda$ value of $G'$, for vertex $v'$ in $G'$, removing $v'$ from $G'$ will not increase the $\lambda$ value of $G'$.

A subgraph with Triangle K-Core number $\lambda$ only satisfies requirement (1), so it is a relaxation of DN-Graph. Requirement (2) makes DN-Graph a locally densest subgraph.

Since detecting all DN-Graphs in a graph is NP-Complete[75], Wang et al.[75] propose to detect $\lambda(e)$, which is the maximum $\lambda$ value of the DN-Graph that edge $e$ participates in. However, detecting $\lambda(e)$ is still difficult, so they propose to iteratively compute a valid
upperbound of $\lambda(e)$, denoted as \textit{valid} $\tilde{\lambda}(e)$. Interestingly, we find that $\kappa(e)$ is actually \textit{valid} $\tilde{\lambda}(e)$ (the proof is below).

\textbf{Definition 5.} valid $\tilde{\lambda}(e)$

Inside $\triangle(u, v, w)$, if $\tilde{\lambda}(u, v) \leq \min(\tilde{\lambda}(u, w), \tilde{\lambda}(v, w))$, we say $w$ supports $\tilde{\lambda}(u, v)$. $\tilde{\lambda}(u, v)$ is valid if and only if $|\{w| w \text{ supports } \tilde{\lambda}(u, v)\}| \geq \tilde{\lambda}(u, v)$.

\textbf{Claim 3.} For any edge $e$, $\kappa(e)$ is valid $\tilde{\lambda}(e)$.

\textbf{Proof.} Since the maximum Triangle K-Core of $e$ is a relaxation of the maximum DN-Graph containing $e$, $\kappa(e)$ is upperbound of $\lambda(e)$, denoted as $\tilde{\lambda}(e)$. In graph $G$ we assign $\tilde{\lambda}(e)$ as $\kappa(e)$ for every edge $e$.

Next we prove $\kappa(e)$ is valid $\tilde{\lambda}(e)$. For edge $e(u, v)$, assume its maximum Triangle K-Core is subgraph $G_e$. For any $\triangle(u, v, w)$ containing $e$ in $G_e$, according to Theorem 1, we have $\kappa(v, w) \geq \kappa(e)$, $\kappa(u, w) \geq \kappa(e)$, so $\tilde{\lambda}(v, w) \geq \tilde{\lambda}(e)$, $\tilde{\lambda}(u, w) \geq \tilde{\lambda}(e)$. According to Definition 5, vertex $w$ supports $\tilde{\lambda}(e)$. There are at least $\kappa(e)$ triangles containing edge $e$ in $G_e$, so there are at least $\kappa(e)$ vertices supporting $\tilde{\lambda}(e)$. $\tilde{\lambda}(e)=\kappa(e)$, therefore $\tilde{\lambda}(e)$ is valid, and $\kappa(e)$ which equals $\tilde{\lambda}(e)$ is valid $\tilde{\lambda}(e)$. \hfill \Box

The advantage of our algorithm is, we avoid the complex iterative approach suggested in DN-Graph, and yield the speedups. Also, DN-Graph does not discuss the use of \textit{template pattern cliques}, and its incremental method is costly since it is iterative.
2.6 Experiments

In this section we present our experimental results. All experiments, unless otherwise noted, are evaluated on a 3.2GHz CPU, 16G RAM Linux-based system at the Ohio Supercomputer Center (OSC). The main datasets we evaluated our results on can be found in Table 2.1.

2.6.1 Comparison with CSV and DN-Graph

Table 2.1: Time Cost Comparison (seconds)

<table>
<thead>
<tr>
<th>Graph</th>
<th>Vertices</th>
<th>Edges</th>
<th>CSV</th>
<th>TriDN</th>
<th>BiTriDN</th>
<th>T-K-Core</th>
<th>DN-Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synthetic</td>
<td>60</td>
<td>308</td>
<td>0.043</td>
<td>0.0012</td>
<td>0.0011</td>
<td>0.0010</td>
<td>0.0007</td>
</tr>
<tr>
<td>Stocks</td>
<td>242</td>
<td>522</td>
<td>0.51</td>
<td>0.017</td>
<td>0.010</td>
<td>0.006</td>
<td>0.095</td>
</tr>
<tr>
<td>PPI</td>
<td>4741</td>
<td>1514/</td>
<td>2.51</td>
<td>0.211</td>
<td>0.121</td>
<td>0.097</td>
<td>X</td>
</tr>
<tr>
<td>DBLP</td>
<td>6445</td>
<td>11848</td>
<td>1.47</td>
<td>0.062</td>
<td>0.046</td>
<td>0.034</td>
<td>X</td>
</tr>
<tr>
<td>Astro-Author</td>
<td>1,7903</td>
<td>1969/2</td>
<td>17,393</td>
<td>73.8</td>
<td>73.9</td>
<td>1.038</td>
<td>X</td>
</tr>
<tr>
<td>Epinions</td>
<td>75879</td>
<td>4057/4</td>
<td>-</td>
<td>262.13</td>
<td>15.71</td>
<td>4.09</td>
<td>188</td>
</tr>
<tr>
<td>Amazon</td>
<td>262111</td>
<td>8997/2</td>
<td>-</td>
<td>34.9</td>
<td>10.59</td>
<td>3.81</td>
<td>X</td>
</tr>
<tr>
<td>Wiki</td>
<td>1,76265</td>
<td>10102/4</td>
<td>-</td>
<td>435.8</td>
<td>17.15</td>
<td>1.89</td>
<td>X</td>
</tr>
<tr>
<td>Flickr</td>
<td>1,715,255</td>
<td>15,555,041</td>
<td>-</td>
<td>-</td>
<td>*60 hours</td>
<td>747</td>
<td>&gt;24 hours</td>
</tr>
<tr>
<td>LiveJournal</td>
<td>4,847,571</td>
<td>42,851,237</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>443</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.2: Memory Comparison

<table>
<thead>
<tr>
<th>Graph</th>
<th>CSV</th>
<th>TriDN</th>
<th>BiTriDN</th>
<th>T-K-Core</th>
<th>DN-code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synthetic</td>
<td>1920 KB</td>
<td>1428 KB</td>
<td>1436 KB</td>
<td>1440 KB</td>
<td>1640KB</td>
</tr>
<tr>
<td>Stocks</td>
<td>3496 KB</td>
<td>2328 KB</td>
<td>2352 KB</td>
<td>2364 KB</td>
<td>2924 KB</td>
</tr>
<tr>
<td>PPI</td>
<td>21 M</td>
<td>7988 KB</td>
<td>8224 KB</td>
<td>8244 KB</td>
<td>X</td>
</tr>
<tr>
<td>DBLP</td>
<td>8800 KB</td>
<td>8044 KB</td>
<td>8232 KB</td>
<td>8272 KB</td>
<td>X</td>
</tr>
<tr>
<td>Astro-Author</td>
<td>-</td>
<td>180 MB</td>
<td>183 MB</td>
<td>182 MB</td>
<td>X</td>
</tr>
<tr>
<td>Epinions</td>
<td>-</td>
<td>282 MB</td>
<td>289 MB</td>
<td>285 MB</td>
<td>386 MB</td>
</tr>
<tr>
<td>Amazon</td>
<td>-</td>
<td>370 MB</td>
<td>384 MB</td>
<td>577 MB</td>
<td>X</td>
</tr>
<tr>
<td>Wiki</td>
<td>-</td>
<td>677 MB</td>
<td>693 MB</td>
<td>684 MB</td>
<td>-</td>
</tr>
<tr>
<td>Flickr</td>
<td>-</td>
<td>-</td>
<td>2.5 GB</td>
<td>&gt;11GB</td>
<td></td>
</tr>
<tr>
<td>LiveJournal</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>6.9 GB</td>
<td>-</td>
</tr>
</tbody>
</table>
In our first set of experiments we compare the performance of Triangle K-Core algorithm (Algorithm 1) with CSV [74] and DN-Graph variants (TriDN and BiTriDN (an improvement over TriDN)) [75] both in terms of efficiency and efficacy. As noted in Section 2.5 we can theoretically show that the DN-Graph variants (TriDN and BiTriDN) converge to the same value as Algorithm 1. Table 2.1 and 2.2 document the execution time/peak memory usage of these algorithms on various datasets, while Figure 2.4 conveys a qualitative comparison by realizing the density plots produced by each algorithm (note that since DN-Graph and Triangle K-Core converge to the same values the density plots are identical).

First, for all the datasets it is clear that Triangle K-Core is the fastest to finish. For some large datasets we could not run BiTriDN or TriDN due to memory thrashing issues and CSV was taking too long to terminate. For Flickr and LiveJournal datasets, we execute Triangle K-Core Algorithm 1 without storing edges’ triangles in memory. The Flickr result for BiTriDN is taken from [75], to give the reader a ballpark figure – the machine they used had a comparable processor but with larger memory. The reason for this high processing time for BiTriDN on Flickr dataset is that each iteration is expensive (55 min per iteration) and a number of iterations (66 are needed for convergence [75]). Compared with DN-Graph, Triangle K-Core allows for a simpler abstraction and this in turns allows us to avoid the iterative approach discussed in DN-Graph. This is the rationale for the significant speedup over DN-Graph variants enabling our algorithm to scale to very large datasets. Also, the peak memory usage of Triangle K-Core algorithm and DN-Graph variants are almost the same, and are less than that of CSV.

Second, when comparing our results with CSV plots on the qualitative visual assessment (Figure 2.4), we observe that while the order in which vertices are processed may
on occasion be slightly different – due to the differences in the estimation procedure of co-clique size and resulting in a shift of the main trends – the main trends themselves are quite similar and easy to discern. In CSV[74], they illustrate the benefit of using the approximate cliques detected by CSV as preprocessing results for detecting exact cliques, we can easily see that Triangle K-Cores can be used for the same purpose.

2.6.2 Protein-Protein Interaction (PPI) Case Study

We also do a case study on PPI network, the plot is in Figure 2.5(a). The 3 red circles in the plot indicate 3 approximate cliques, we draw the 3 cliques (from left to right) in Figure 2.5(b)(c)(d). We find that clique 1 is exactly the same as what Wang et al. detected in [75]. The names in the parenthesis are the names used in [75]. Clique 2 is shown to be 10-vertex clique in the plot, in fact it is an exact 10-vertex clique. Clique 3 has 10 vertices, but it is shown to be 9-vertex clique, because the edge between APC4 and CDC16 is missed.

2.6.3 Experimental Results of Update Algorithm

To evaluate the effectiveness of our update algorithm we randomly add/delete about 1% of edges from five large datasets in Table 2.1, and in Table 2.3 we compare the time costs of re-computing and updating the maximum Triangle K-Cores incrementally. Results reported are averaged over 5 runs. Here Re-compute time is actually the execution time of steps 8-18 in Algorithm 1, and Update time is the execution time of the Algorithm 2. The results clearly demonstrate that the incremental algorithm is effective.
Figure 2.4: Qualitative Comparison between CSV and Triangle K-Core Note that in the figure we note regions in the plot where the two plots are near identical or similar (S) and regions where there is a distinct phase shift (PS).
Figure 2.5: Cliques in PPI dataset
2.6.4 Dual View Plots: Wiki Case Study

In Figure 2.6, we present an example to illustrate how Dual View Plots can highlight the change of clique-like structures within a dynamic graph setting.

We use two consecutive snapshots of Wiki datasets for this purpose. A snapshot of Wiki dataset is comprised of vertices, which are Wiki articles, and references among them. Figure 2.6(a) represents the clique distribution plot of 1st snapshot $G_a$, and it corresponds to plot(a) in Algorithm 3. Figure 2.6(b) visualizes the cliques containing new edges in the 2nd snapshot, and it corresponds to plot(b) in Algorithm 3.

Then in Figure 2.6(b) we select the 3 cliques with highest density for more analysis – denoted using a green triangle, a red rectangle, and an orange ellipse. The Dual View Plot tool can then locate their corresponding vertices in Figure 2.6(a) using the same markers, allowing the user to gain insights into how these clique-like structures evolved. For example, one can observe that the vertices (green triangle) are located in two places in Figure 2.6(a); some vertices are in a 10-vertex clique, and one single vertex is in a 5-vertex clique. Drilling down as shown in Figure 2.6(c), “Astrology” is the single vertex, the red dashed-lines are newly added edges. Essentially between two consecutive snapshots, a new Wiki page and the corresponding Wiki links were established thereby forming a larger clique. The details about the other 2 clique-like structures are presented Figure 2.6(d) and
Figure 2.6(e) and are also self explanatory – the two cliques are formed by merging vertices from different original cliques, they both indicate an expanding trend on specific topics.

### 2.6.5 Dynamic Template Pattern Cliques: DBLP Study

The DBLP graph data set is consisted of authors(vertices) and their collaborations(edges) in each year. In the following we will detect the template pattern cliques introduced in Figure 2.3 in DBLP data set, and show that such cliques reveal interesting hidden information about paper topics.

To illustrate the Emerging Clique, we use the DBLP 2003 and 2004 data as two snapshots. Emerging Clique Plot for DBLP in 2004 is shown in Figure 2.7. The red circle highlights the densest (6-vertex) Emerging Clique. The authors are Rudi Studer, Karl Aberer, Arantza Illarramendi, Vipul Kashyap, Steffen Staab, Luca De Santis. They are from 5 different countries, and they collaborated for the first time in 2004.

In a similar manner we use DBLP 2003 and 2004 to plot the Bridge Clique distribution of DBLP 2004 in Figure 2.8. The first major clique on the plot (red circle) is an interesting 6-vertex Bridge Clique. In 2003, the 6 authors were in two independent groups: Group 1: Divesh Srivastava, Graham Cormode, S. Muthukrishnan, Flip Korn; and Group 2: Theodore Johnson, Oliver Spatscheck. In Group 1, the authors primarily worked on data streams, and in Group 2 the researchers mainly worked on networking in 2003. In 2004, the 6 authors worked together on “Holistic UDAFs at Streaming Speeds”, which is a topic “merged” by data stream and network.

Using datasets DBLP 2000 and DBLP 2001, we plot the Expanding Cliques in DBLP 2001 in Figure 2.9. The densest Expanding Clique (denoted by a red circle) shows a 9-vertex clique. In 2000, the 3 authors Quan Wang, David Maier, Leonard D. Shapiro
Figure 2.6: Dual View Plots for Clique Changes
worked on a paper about Query Processing. In 2001, the 3 authors were joined by 6 other authors who did not appear in DBLP 2000 dataset, *Paul Benninghoff, Keith Billings, Yubo Fan, Kavita Hatwal, Yu Zhang, Hsiao-min Wu*, and they worked on one paper “Exploiting
Upper and Lower Bounds in Top-Down Query Optimization”, which is an extension of the previous work in 2000.

2.6.6 Static Template Pattern Cliques: PPI Case Study

We next discuss how domain-driven template pattern cliques based on Triangle K-Cores can be exploited in the case of static data such as Protein Protein Interaction (PPI) data. In PPI dataset, each vertex represents a protein, and each protein belongs to a complex, which includes proteins of similar functions. Now we define a variant of Bridge Clique to be a clique that connects vertices from two different complexes. Here we define an edge’s label to be “new” when it connects two vertices from different complexes, otherwise its label is “old”. Then we apply the previously described Bridge Clique detection algorithm on PPI dataset, and get the Bridge Clique distribution plot in Figure 2.10(a).

![Plot of Bridge Cliques in PPI dataset](image1)

![Details of Bridge Clique 1](image2)

Figure 2.10: Detect Bridge Cliques in PPI dataset
We highlight two peaks using red circles, the Bridge Clique 1 in left red circle is comprised of vertices from the following two complexes:

- **20S proteasome complex**: PRE1

- **19/22S regulator complex**: RPN11, RPN12, RPN9, RPT1, RPN5, RPN5, RPT3, RPN8

In Figure 2.10(b), we draw the details of Bridge Clique 1 in the dashed-line rectangle, where the green vertices belong to the complex “19/22S regulator”, the blue vertices belong to complex “20S proteasome”, black edges are intra-complex edges, red dashed-lines are inter-complex edges. Besides drawing Bridge Clique 1, we also draw other vertices in complex “20S proteasome”, and find that the vertex “PRE1” is an important bridge node connecting the two complexes.

The proteins in right red circle comprise two Bridge Cliques, the first is Bridge Clique 2:

- **Gac1p/Glc7p complex**: GLC7

- **mRNA cleavage and polyadenylation specificity factor complex**: PAP1, CFT2, CFT1, PTA1, MPE1, YSH1, YTH1, REF2

the second is Bridge Clique 3:

- **mRNA cleavage factor complex**: RNA14

- **mRNA cleavage and polyadenylation specificity factor complex**: PAP1, CFT2, CFT1, PTA1, MPE1, YSH1, YTH1, FIP1
We find that Bridge Clique 2 and 3 have a lot of overlap vertices, which indicate that all the vertices in them are very closely related in function, this is consistent with known biological knowledge.

2.7 Details of Update Algorithm

2.7.1 Triangle K-Core Update Algorithm

Before executing the update algorithm, for each edge $e$, we firstly initialize $e\.order$, which indicates the time when $e$ is processed in Algorithm 1. If $e\.order$ is less than $e'\.order$, then $e$ is processed earlier than $e'$. After execution of Algorithm 1 $e\.order$ is initialized as the index of edge $e$ in list $Edges$.

Algorithm 5 is to update edges’ maximum Triangle K-Cores when adding edges. In step 4, according to Rule 0, we put some edges of $t_{new}$ in $PotentialList$ because their maximum Triangle K-Cores might change. All edges in $PotentialList$ are sorted in the increasing order of $e\.order$, that is because we will simulate Algorithm 1 to recompute on $PotentialList$, we need to maintain the order. $t_{new}$ is not yet in any edge’s maximum Triangle K-Core, so in steps 5-6, we add it to the maximum Triangle K-Core of the first edge of $PotentialList$.

Steps 7-24 update $\kappa(e)$ for each edge $e$ in $PotentialList$. In step 8, $ori_\kappa(e)$ stores the original maximum Triangle K-Core number of $e$ before update, according to Rule 0, this value is equal to $\mu$. In step 9 we construct the following set of triangles that violate Theorem 1 ($IsInCore(t, e)$ tests whether triangle $t$ is in edge $e$’s maximum Triangle K-Core):

- $e\.addTris = \{ \triangle t \mid \triangle t$ is on edge $e$, and $\triangle t$ contains edge $e'$ that $\kappa(e') > \kappa(e) \land IsInCore(t, e') \land \neg IsInCore(t, e) \}$

Steps 10-12 then process these “illegal” triangles in $e\.addTris$. After that, $\kappa(e)$ might increase and lead to the following set of triangles that violate Theorem 1:
Algorithm 5 Update Algorithm for Adding Edges

1: for each added triangle $t_{\text{new}}$ do \\
2:    Create empty lists $\text{ChangingList}$, $\text{PotentialList}$, $\text{TempList}$; \\
3:    Find the smallest value $\mu$ of $t_{\text{new}}$’s edges’ $\kappa$ value; \\
4:    Put $t_{\text{new}}$’s edges whose $\kappa$ value equals $\mu$ in $\text{PotentialList}$ in order; \\
5:    AddToCore($t_{\text{new}}$, $e_0$); $\text{// } e_0$ is the first edge of $\text{PotentialList}$ \\
6:    $\kappa(e_0) + +$; \\
7:    for each edge $e$ in $\text{PotentialList}$ do \\
8:        $\text{ori}_\kappa(e) = \mu$; \\
9:        Construct triangles set $e.addTris$; \\
10:       for each triangle $t_a$ in $e.addTris$ do \\
11:           $\kappa(e) + +$; \\
12:           Construct triangles set $e.delTris$; \\
13:       for each triangle $t_d$ in $e.delTris$ do \\
14:           if $\kappa(e) > \text{ori}_\kappa(e)$ then \\
15:              DelFromCore($t_d$, $e$); \\
16:              $\kappa(e) - -$; \\
17:           Remove $e$ from $\text{PotentialList}$; \\
18:       if $\kappa(e) > \text{ori}_\kappa(e)$ then \\
19:          put $e$ to $\text{ChangingList}$; \\
20:          Insert $e.post_edges$ to $\text{PotentialList}$ in order; \\
21:       else \\
22:          $\text{TempList} = \text{Simulate\_Alg01}(e)$; \\
23:          Insert edges in $\text{TempList}$ between $e$’s previous and next edge in $\text{Edges}$ list; \\
24: while $\text{ChangingList}$ is not empty do \\
25:    $\text{TempList} = \text{Simulate\_Alg01}(\text{ChangingList.min\_edge})$; \\
26:    Insert edges in $\text{TempList}$ in $\text{Edges}$ list, between the last edge with $\kappa(e) = \mu$ and first edge with $\kappa(e) = \mu + 1$;

- $e.delTris = \{\triangle t \mid \triangle t$ is on edge $e$, and $\triangle t$ contains edge $e’$ that $e’.order < e.order \land \kappa(e’) < \kappa(e) \land \text{IsInCore}(t, e’) \land \text{IsInCore}(t, e)\}$,

Steps 14-17 then process these “illegal” triangles in $e.delTris$.

In step 19, if $\kappa(e)$ increases, some of $e$’s neighbor edges might change $\kappa$ value, according to Rule 0, these edges are in the following set,
Algorithm 6 Simulate_Algo1($e_{init}$)

1: Create an empty list TempList;
2: Add $e_{init}$ to TempList;
3: for each edge $e$ in TempList do
4:   Construct triangles set $e.addTris$;
5:   for each edge $e'$ that shares a triangle $T$ in $e.addTris$ with $e$ and $e'$ is in ChangingList do
6:      if $\kappa(e') > \kappa(e)$ then
7:         DelFromCore($T$, $e'$);
8:         $\kappa(e') = -$;
9:      if $\kappa(e') = \kappa(e)$ then
10:         Move $e'$ from ChangingList to TempList;
11:   Return TempList;

- $e.post\_edges = \{\text{Edge } e' \mid e' \text{ shares a triangle with } e, \text{ and } \kappa(e') = \mu \wedge e'.order > e.order\}$

we put these edges in PotentialList.

If $\kappa(e)$ does not change, then edge $e$ is processed now, in step 23 we use method Simulate_Algo1 to simulate Algorithm 1 to update $e$ and its neighbors’ maximum Triangle K-Cores. Simulate_Algo1 will return a list of edges whose $\kappa$ value is determined. When all edges in PotentialList have been processed, we update maximum Triangle K-Cores of edges in ChangingList (step 26), ChangingList.min_edge is the edge in ChangingList with the minimum $\kappa$ value. In step 27 we put all edges in ChangingList in the corresponding positions in sorted list Edges.

Algorithm 7 is to update edges’ maximum Triangle K-Cores when deleting edges. In step 4, according to Rule 0, we put some edges of $t_{del}$ in PotentialList. In steps 5-8, we remove deleted triangles from its edges’ maximum Triangle K-Cores. In step 11, we construct two sets of triangles on e:
\( e.addTris = \{ \triangle t \mid \triangle t \text{ is on edge } e, \text{ and contains edge } e' \text{ that, } \kappa(e') = ori.\kappa(e) \land e'.order < e.order \land IsInCore(t,e') \land !IsInCore(t,e) \} \)

\( e.delTris = \{ \triangle t \mid \triangle t \text{ is on edge } e, \text{ and contains edge } e' \text{ that, } \kappa(e') < ori.\kappa(e) \land IsInCore(t,e') \land IsInCore(t,e) \} \)

When step 13 is satisfied, all the triangles in \( e.addTris \) violate Theorem 1, so we add the first triangle of \( e.addTris \) to \( e \)'s maximum Triangle K-Core to obey Theorem 1. Then \( \kappa(e) \) changes and if now step 20 is satisfied, all the triangles in \( e.delTris \) violate Theorem 1, so we remove the first triangle of \( e.delTris \) from maximum Triangle K-Core of \( e \) to obey Theorem 1. In steps 28-30, if \( \kappa(e) \) changes, according to Rule 0 we find the following set of edges whose maximum Triangle K-Core might change, and insert them in \( PotentialList \).

\( e.share\_edges = \{ \text{Edge } e' \mid e' \text{ shares a triangle with } e, \kappa(e') = \mu \} \)

Finally we put the edges in \( ChangingList \) in correct positions in list \( Edges \).

In Algorithm 5 and 7, after each iteration, each edge’s \( order \) value needs to be re-computed, which will be costly. In our implementation, we only update edges whose \( order \) value have been changed, that is, when a set of edges \( \{e1, e2, ... en\} \) are inserted between two edges \( Ea, Eb \), then \( ei.order = Ea.order + (Eb.order - Ea.order) * i/(n + 1) \).

If we do not store triangles in Algorithm 1, when updating edge \( e \) in \( PotentialList \) we need to re-construct \( e \)'s triangles, and the triangle information we need to know is whether a triangle of \( e \) is in \( e \)'s maximum Triangle K-Core. We recover this information as following: we firstly get triangle \( t \)'s “process time”, which is the smallest \( order \) value of its edges, then we apply the following Rule to find all \( e \)'s triangles in \( e \)'s maximum Triangle K-Core.

\( Rule 1: \text{if } \kappa(e) = k, \text{ then we sort } e \text{'s triangles in the increasing order of their “process time”, the last } k \text{ triangles will be in } e \text{'s maximum Triangle K-Core.} \)
Algorithm 7 Update Algorithm for Deleting Edges

1: for each deleted triangle \( t_{del} \) do
2: Create empty lists \( ChangingList, PotentialList \);
3: Find the smallest value \( \mu \) of \( t_{del} \)'s edges' \( \kappa \) value;
4: Put \( t_{del} \)'s edges whose \( \kappa \) value equals \( \mu \) in \( PotentialList \) in order;
5: for each edge \( e \) in \( PotentialList \) do
6: if IsInCore(\( t_{del}, e \)) then
7: DelFromCore(\( t_{del}, e \));
8: \( \kappa(e) \leftarrow -\); 
9: for each edge \( e \) in \( PotentialList \) do
10: \( ori\_\kappa(e) = \mu; \)
11: Construct triangles sets \( e.addTris \) and \( e.delTris \);
12: while true do
13: if \( \kappa(e) < ori\_\kappa(e) \) then
14: if \( e.addTris \) is not empty then
15: AddToCore(\( e.addTris.first, e \));
16: \( \kappa(e) \leftarrow +; \)
17: remove \( e.addTris.first \) from \( e.addTris \);
else
18: break;
19: if \( \kappa(e) = ori\_\kappa(e) \) then
20: if \( e.delTris \) is not empty then
21: DelFromCore(\( e.delTris.first, e \));
22: \( \kappa(e) \leftarrow -; \)
23: remove \( e.delTris.first \) from \( e.delTris \);
else
24: break;
25: Remove \( e \) from \( PotentialList \);
26: if \( \kappa(e) < ori\_\kappa(e) \) then
27: Put \( e \) in \( ChangingList \);
28: Insert \( e.share\_edges \) to \( PotentialList \) in order;
29: Insert edges in \( ChangingList \) in \( Edges \) list, between the last edge with \( \kappa(e) = \mu - 1 \) and first edge with \( \kappa(e) = \mu; \)
Chapter 3: A Fast Parallel Method to Detect Approximate Cliques

3.1 Introduction

Approximate cliques are usually used as pre-processing results for detecting exact cliques[74, 21]. Recently some works [85, 74, 75] have shown that approximate cliques can also be used to uncover dense communities in networks. A community generated by a traditional community detection method may contain outliers, which are vertices loosely connected to other members. An approximate clique hardly has this problem, because every member in an approximate clique is connected to almost every other member. So the dense communities uncovered by approximate cliques have better qualities.

The most difficult problem of using approximate cliques in community detection is the time complexity to detect approximate cliques. The exact clique detection problem is NP-Complete, even the parallel methods are very costly[62, 26]. Some approximate clique structures have been proposed recently [85, 74, 75], and they are shown to be very effective in detecting dense communities in large graphs. However, as graph size increases to millions or tens of millions of nodes, these methods may take hours to detect the approximate cliques, another problem is the graph would be too large to fit in memory of a single machine. Although some I/O efficient methods[72] have been proposed to deal with the memory problem, they are still very costly.
In this chapter we propose a parallel method to detect the approximate cliques in large graphs. Our parallel method is based on Triangle K-Cores[85], since it is shown to be close to cliques and are can be more easily detected than other approximate clique structures. We first partition the graph, and extend each partition by its vertices’ immediate neighbors. Then we distribute these extended partitions to computing nodes, and detect approximate cliques (Triangle K-Cores) on each partition. The benefit of our method is each computing node works independently, and there is no communication cost, so our method guarantees that as number of nodes increases, the time cost reduces.

We notice when applying approximate cliques to uncover dense communities, we may not care too much about how accurately they can approximate cliques. So we design a sparsification mechanism to speed up the approximate clique detection process, while affecting the accuracy as little as possible.

3.2 Preliminaries and Background

Given a graph $G = \{V,E\}$, $V$ is the set of distinct vertices $\{v_1, \ldots, v_{|V|}\}$, and $E$ is the set of edges $\{e_1, \ldots, e_{|E|}\}$. A graph $G' = \{V',E'\}$ is a subgraph of $G$ if $V' \subseteq V$, $E' \subseteq E$.

**Definition 6.** A Clique is a subgraph $G'$ of $G$ that every two vertices of $G'$ are connected by an edge.

3.2.1 Clique-like subgraph structures

Since detecting and extracting cliques efficiently is known to be very hard, some clique-like structures (Triangle K-Core, DN-Graph, K-Core etc.) have been proposed to approximate cliques.
In this paper, we are going to use Triangle K-Cores[85] to approximate cliques, because it is more easily to detect than some structures (DN-Graph etc.), and provides better approximation of cliques than other structures (K-Core etc). Please note that the parallel mechanism used in this paper can be easily adapted for clique-like structures other than Triangle K-Core.

The definition of Triangle K-Core algorithm is as follows:

**Definition 7.** A **Triangle K-Core** is a subgraph $G'$ of $G$ that each edge of $G'$ is contained within at least $k$ triangles in the subgraph. The **Triangle K-Core number** of this Triangle K-Core is referred to as $k$. The **maximum Triangle K-Core** associated with an edge $e$ is the subgraph $G_e$ containing $e$ that has the maximum Triangle K-Core number. The Triangle K-Core number of $G_e$ is the **maximum Triangle K-Core number** of edge $e$. We use $\kappa(e)$ to denote the maximum Triangle K-Core number of edge $e$.

Paper [85] provides an effective method to compute $\kappa(e)$ associated with each edge $e$. After computing $\kappa(e)$ for each edge, we can construct the maximum Triangle K-Cores associated with each edge. Paper [85] also shows that $\kappa(e)$ is the upperbound of the size of the maximal clique $e$ participates in, which is a very important theoretical property, because based on this property we can use the Triangle K-Cores as pre-processing results for exact maximal clique detection.

Although paper [85] shows that the Triangle K-Core algorithm is much faster than other clique-like structure detection algorithms, for large scale graphs, Triangle K-Core algorithm still takes too much time. In the following sections we propose a parallel algorithm to detect approximate cliques based on Triangle K-Core algorithms.
3.3 Parallel Algorithm to Detect Approximate Cliques

3.3.1 Basic Method

Our parallel algorithm is built upon [22], which proposed an external memory Maximal Clique Enumeration algorithm. Before describing our algorithms, we need to introduce the definition of extended subgraph.

**Definition 8.** Given a subgraph $P(V_p, E_p)$ of graph $G(V, E)$, the extended subgraph of $P$ is denoted as $P^+(V_{p^+}, E_{p^+})$, where $V_{p^+} = V_p \cup \{ v : (v, u) \in E, u \in V_p \}$, $E_{p^+} = \{ (v, u) : v \in V_{p^+}, u \in V_{p^+}, (v, u) \in E \}$.

Assume there are $N$ computing nodes, and our parallel algorithm is as follows:

**Algorithm 8** Parallel Algorithm to Detect Approximate Clique

1: Partition the graph $G$ to $N$ partitions;
2: Compute extended subgraph $P^+$ for each partition $P$;
3: Distribute each extended subgraph $P^+$ on each computing node, and execute Triangle K-Core algorithm on $P^+$ to detect approximate cliques;

In the following Claim, we prove that

**Claim 4.** For any edge $e(u, v)$, assume $u$ is in Partition $P$, then the $\kappa(e)$ value computed in extended subgraph $P^+$ is still the upperbound of the size of the maximal clique $e$ participates in.

*Proof.* Assume the maximal clique of edge $e$ is $C$, according to the definition of clique, all the vertices of $C$ should be the neighbors of vertex $u$, so all the vertices of $C$ are in subgraph $P^+$. Since $P^+$ contains all the edges among its vertices, $P^+$ contains all the edges of $C$. Thus $P^+$ contains $C$. According to the definition of Triangle K-Core, clique $C$ is a special
type of Triangle K-Core, so its size is less than or equal to the size of e’s maximal Triangle K-Core, that is $\kappa(e)$ is still the upperbound of the size of the maximal clique of e.

The claim shows that the Triangle K-Cores detected using above algorithm can still be used as pre-processing results for exact maximal clique detection algorithm.

In our implementation, we use Metis software package to partition the original graph. Metis is very fast, and each partition generated from Metis has approximately the same number of nodes.

### 3.3.2 Sparsification

The parallel method above detects the approximate maximal clique for each edge. In some applications, however, we only need to use the approximate maximal cliques of vertices (such as mining cohesive subgraphs[74], community detection[28]). In a graph there are usually more edges than vertices, so we can reduce the cost more if we only detect the approximate maximal cliques of vertices rather than edges. In the following we explain how to adapt Algorithm 8 to do so. The idea is, for the extended subgraph $P^+$ of a subgraph P, we sparsify it by only preserving vertices and edges in $P^+$ that can possibly comprise maximal cliques of vertices in P, the sparsified graph is denoted as $P^+_s$. Here we give a “vague” definition of $P^+_s(V_{P^+_s}, E_{P^+_s})$, $V_{P^+_s} = V_P \cup \{v: v \text{ is possibly in } u \text{’s maximal clique}, u \in V_P\}$, $E_{P^+_s} = \{(v, u): (v, u) \text{ is possibly in } v \text{’s maximal clique, } v \in V_P \text{ (or } u \in V_P)\}$.

In the following we explain how to construct $V_{P^+_s}$ and $E_{P^+_s}$.

**Constructing** $V_{P^+_s}$: for each vertex v in partition P, we need to identify v’s neighbor vertices that are possible in v’s maximal clique, in the following we use v.possible_clique_vers to store these vertices.
In the first step, we use Algorithm 9 to compute the upperbound of the size of v’s maximal clique:

**Algorithm 9** Estimate Maximal Clique Size of Vertex v

1. Count triangles on each edge of vertex v. Use \( tri(e) \) to denote the number of triangles on edge e;
2. sort v’s neighbor edges in descending order of \( tri(e) \), the sorted edges are stored in \( v_{edges}[] \);
3. for int i=0; i < v_{edges}.size(); i++ do
   4. if \( tri(v_{edges}[i]) < i \) then
      5. return i+1;
   6. return v_{edges}.size()+1;

The algorithm is based on the following Claim:

**Claim 5.** for vertex v, if the size of its maximal clique is k, then there are at least k-1 edges containing v, and each of the k-1 edges has at least k-2 triangles on them.

The proof of this claim is omitted since it is straightforward. After we get the upper-bound of the size of v’s maximal clique, denoted by \( uc(v) \), we estimate the true value \( tc(v) \) to be \( tc(v) = uc(v) \times \alpha \), where \( \alpha < 1 \), and add u to v.possible.clique.vers when edge (u, v) has at least \( tc(v)-2 \) triangles on it.

Here we notice another problem, the computation of triangles on every edge in step 1 is very costly. So we use min-wise hashing method [13] to estimate the number of triangles on each edge. In our experiments, we find that the min-wise hashing method speeds up triangle counting significantly, but it will cause some inaccuracy to Algorithm 9. So we are adopting the following method to make use of K-Core to improve accuracy, especially for vertices participating in the largest cliques.
K-Core subgraph is also a relaxation of cliques, and there is a \(O(|V| + |E|)\) algorithm to detect every vertex’s maximum K-Core number, the complexity is much lower than Triangle K-Core algorithm and other algorithms. However, in terms of approximating each vertex’s maximal clique, K-Core is less accurate than Triangle K-Core algorithm. Sometimes a vertex’s maximum K-Core number is much bigger than its maximal clique size, but we can safely use it to compute v.possible_clique.vers for the vertex v, the method is in Algorithm 10.

**Algorithm 10 Identify Maximal Clique Vertices based on K-Core**

1: Compute each vertex v’s maximal K-Core number \(k(v)\);
2: for each vertex v whose \(k(v) > k_0\) do
3:   for each edge \(e(v, w)\) do
4:     if \(k(w) \geq k(v)\) then
5:       v.possible_clique.vers.push_back(w);

Step 2 shows that we only apply this strategy for vertices whose maximal K-Core number is above a threshold \(k_0\). The reason is, for a vertex v with low maximal K-Core numbers, most of its neighbor vertices have higher maximal K-core numbers than v, so most of their neighbor vertices will be added to v.possible_clique.vers. Thus the sparsification will not sparsify the graph effectively. In experiment, we set threshold \(k_0\) so that only 1% vertices have maximal K-Core numbers greater than \(k_0\). Actually these 1% vertices usually participate in the largest cliques.

**Constructing** \(E_{P^+}\): Algorithm 11 is to construct \(E_{P^+}\). In Algorithm 11, v.clique_ids is to store the ids of the maximal cliques that v possibly participates in.

In steps 7-11, we don’t need to check all the edges, for edges between a vertex in P and the vertices in its possible_clique.vers, we can skip steps 7-11 and preserve them directly.
Algorithm 11 Constructing $E_{P^+}$

1: for each vertex $v$ in $P$ do
2:   $v$.clique_ids.push_back($v$.id);
3: for each vn in v.possible_clique_vers do
4:   vn.clique_ids.push_back($v$.id);
5: for each edge $e(u, v)$ in $P^+$ do
6:   if $u$.clique_ids and $v$.clique_ids have common members then
7:     add edge $e(u, v)$ to $P^+_s$;

In step 8, in order to check whether two sets have common members, one approximate method is using min-wise hashing. Another accurate method is to sort all the elements in each set, and execute a merge sort to check the common members. In experiments the accurate method is not very costly, so we use this method in experiments.

3.3.3 Load Balancing

Load balancing is very important to our parallel algorithm. Although Metis guarantees that each partition has approximately the same number of nodes, the time cost of each partition may still vary significantly. We use the following method to make the computational costs of each partition balanced.

The idea is we first estimate the computational cost of each partition, and further divide those partitions with high cost into smaller partitions, then re-distribute the smaller partitions to other computing nodes. The benefit of our parallel algorithm is there is no communication cost among computing nodes, so we can easily and accurately estimate the computational cost of each partition by the time complexity proposed in [85], that is for each graph, the cost of Triangle K-Core algorithm is $O(\Sigma(d_i^2) + |Tri| + |E|)$, where $d_i$ is the degree for node $i$, $|Tri|$ is the total number of triangles in the graph. Since $\Sigma(d_i^2) > |Tri|$
and $\sum(d_i^2) > |E|$ , we just use $C_p = \sum(d_i^2)$ to estimate the cost of each partition P. The algorithm to divide and re-distribute costly partitions is in Algorithm12.

---

**Algorithm 12** Re-distribute Partitions

1: Estimate the cost $C_p$ of each partition P;
2: Compute the average cost of all partitions, denote it as $C_{avg}$;
3: Create two empty arrays HighCostList, LowCostList;
4: for each Partition $P_i$ do
5: \[ t = \text{floor} \left( \frac{C_{P_i}}{C_{avg}} \right) \]
6: if $t > 2$ then
7: divide $P_i$ into $t$ smaller partitions by Metis;
8: for each smaller Partition $P_{ij}$ do
9: insert pair $< P_{ij}, C_{P_{ij}} >$ to HighCostList;
10: insert pair $< \text{null}, 0 >$ to LowCostList;
11: if $t < 1$ then
12: insert pair $< P_i, C_p >$ to LowCostList;
13: sort LowCostList in ascending order of costs
14: sort HighCostList in descending order of costs
15: for $i = 0; i < \text{HighCostList.size}(); i + +$ do
16: merge the partitions in LowCostList[i] and HighCostList[i]

---

### 3.3.4 An Out-of-Core Variant

In our parallel method, there is no communications between nodes, so we can easily adapt our parallel method to be an Out-of-Core method to detect approximate cliques on a single machine. The idea is straightforward, we first partition the graph, then load each partition in a sequential way, for each loaded partition we compute its extended graph and compute the approximate cliques as above. A new problem for Out-of-Core method is how to guarantee that each partition can fit in memory. One solution is, first partition the graph into $G/M$ partitions, where $G$ is the graph size, $M$ is the memory limit, for each partition, if the size of its extended graph exceeds the memory limit, we further divide this partition
until its extended graph can fit in memory. The sparsification mechanism can also be used to speed up the Out-of-Core method.

3.4 Application of Approximate Cliques

3.4.1 Generate CSV plot

Based on the approximate cliques computed before, we can generate the CSV plot to visualize the distribution of the dense subgraphs (approximate cliques). The previous method\cite{74, 85} requires to compute the approximate maximal clique size of each edge, here through sparsification, we only compute approximate maximal clique size for a portion of edges. In order to preserve the connectivity of the graph, for those edges that are removed in sparsification, we set their approximate maximal clique size to be 2, the minimum value of a clique size. Then we input all the edges with their approximate clique sizes to the CSV plotting procedure, the experimental results show that the plot generated by our parallel+sparsification method is very close to the plot generated by previous work\cite{85}, while our parallel+sparsification method reduces cost significantly.

3.5 Experiments

We use the following 4 datasets (Table\ref{table:parallel_algorithm_datasets}) in our experiments.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
Graph Dataset & Number of Vertices & Number of Edges \\
\hline
Web & 685230 & 6649470 \\
Flickr & 1715255 & 15555041 \\
Wiki2010 & 1815914 & 34022831 \\
orkut & 3072441 & 117185083 \\
\hline
\end{tabular}
\caption{Parallel Algorithm Data Sets}
\end{table}
3.5.1 Performance

We evaluated the performance of our parallel algorithms using the 4 datasets, the results are reported in table 3.2. Here “accu” means the performance of the parallel method without sparsification, “spar” means the performance of the parallel method with sparsification. Please notice that the performance on single node without sparsification is the performance of the original Triangle K-Core algorithm[85]. Results show that our algorithms scale well on large datasets, the sparsification mechanism works effectively to improve the performance.

We also define a measure to compute the accuracy of our parallel+sparsification method. The definition is

\[
\text{accuracy} = 1 - \Sigma(\text{para}_{\text{clique}}_i - \text{ori}_{\text{clique}}_i)/\text{ori}_{\text{clique}}_i)/|V|,
\]

where \(\text{para}_{\text{clique}}_i\) is the approximate maximal clique size of vertex \(i\) computed by our parallel+sparsification method on 32 nodes, and \(\text{ori}_{\text{clique}}_i\) is the approximate maximal clique size of vertex \(i\) computed by original Triangle K-Core algorithm[85]. Results show that we maintain a good overall accuracy (> 0.9) for all vertices. For the vertices with top 1% highest approximate maximal clique size, the accuracy is greater than 0.98.

<table>
<thead>
<tr>
<th>Graph</th>
<th>1 node</th>
<th>2 nodes</th>
<th>4 nodes</th>
<th>8 nodes</th>
<th>16 nodes</th>
<th>32 nodes</th>
<th>1% Accu.</th>
<th>Accu.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web(accu)</td>
<td>622</td>
<td>430</td>
<td>376</td>
<td>351</td>
<td>320</td>
<td>303</td>
<td>0.999</td>
<td>0.902</td>
</tr>
<tr>
<td>Web(spar)</td>
<td>185</td>
<td>126</td>
<td>108</td>
<td>86</td>
<td>62</td>
<td>47</td>
<td>0.999</td>
<td>0.902</td>
</tr>
<tr>
<td>Flickr(accu)</td>
<td>781</td>
<td>730</td>
<td>686</td>
<td>631</td>
<td>590</td>
<td>557</td>
<td>0.983</td>
<td>0.982</td>
</tr>
<tr>
<td>Flickr(spar)</td>
<td>580</td>
<td>431</td>
<td>307</td>
<td>289</td>
<td>226</td>
<td>187</td>
<td>0.983</td>
<td>0.982</td>
</tr>
<tr>
<td>Wiki2010(accu)</td>
<td>2743</td>
<td>2517</td>
<td>2320</td>
<td>1789</td>
<td>1314</td>
<td>1121</td>
<td>0.982</td>
<td>0.907</td>
</tr>
<tr>
<td>Wiki2010(spar)</td>
<td>1250</td>
<td>667</td>
<td>389</td>
<td>257</td>
<td>156</td>
<td>102</td>
<td>0.982</td>
<td>0.907</td>
</tr>
<tr>
<td>orkut(accu)</td>
<td>4879</td>
<td>3736</td>
<td>2901</td>
<td>2562</td>
<td>2395</td>
<td>2010</td>
<td>0.994</td>
<td>0.960</td>
</tr>
<tr>
<td>orkut(spar)</td>
<td>2131</td>
<td>1479</td>
<td>926</td>
<td>615</td>
<td>436</td>
<td>317</td>
<td>0.994</td>
<td>0.960</td>
</tr>
</tbody>
</table>
3.5.2 CSV plots

In the following, we plot the CSV plots of the four datasets. For each dataset, the first plot is generated by original Triangle K-Core algorithm[85]. the second plot is generated by our parallel method on 32 nodes with sparsification, results show that with sparsification, we still can capture the distribution of approximate cliques in the original graph.
Figure 3.1: CSV plots comparison for Web
Figure 3.2: CSV plots comparison for Flickr
Figure 3.3: CSV plots comparison for Wiki2010
Figure 3.4: CSV plots comparison for Orkut
Chapter 4: Analyzing and Visualizing Scalar Fields on Graphs

4.1 Introduction

Fundamental to network science is the ability to visualize networks at scale. This in turn has led to numerous advances in graph drawing and network visualization. Additionally the network scientist is often interested in measures/attributes on both nodes and edges of the graph [87, 80]. Such measures often encode information about connectivity, or incorporate heterogeneous information related to content. Visualizing the measure information in such graphs (where each node or edge has one or more scalar measures associated with it) further exacerbates the visualization problem.

In this thesis we propose a 3D visual platform to explore graphs with numerical measures associated with nodes (or edges) – referred to as scalar graphs, and the measure values are referred to as scalar values. Each scalar value could either be a natural attribute or a derived attribute summarizing information from multiple natural attributes (e.g. triangle density, centralities, cliquishness) [85, 74]. We model the problem as a continuous function $f : X \rightarrow \mathbb{R}$: the domain $X$ is a simplicial complex whose vertex set is the set of input graph nodes, and its topology is determined by the input graph topology. We call such a representation of a graph a graph-induced scalar field. We then borrow the powerful “terrain metaphor” idea for visualizing scalar fields [77, 39, 40] and adapt it to the graph context. In the final terrain visualization, vertices of the same scalar value will be in the
same height, and if they are connected in the graph, they will be in the same “mountain” of the terrain. Intuitively, a high mountain represents a cluster of vertices of high scalar values.

Although our visual platform is a general visualization platform, it could be used for specific data mining tasks, such as identifying dense subgraphs (K-Core, K-Truss, etc.), while the general graph visualization methods mentioned above do not have the ability. Later we will show that even compared with some specific dense subgraph visualization tools [3, 85], our visualization method shows more advantages.

Figure 5.3 shows two applications of our visual platform: visualizing dense subgraphs (K-Cores, K-Trusses, etc) and visualizing the communities in social networks. In Figure 4.1(a), we use K-Core number [12] as a measure to visualize the K-Cores in a collaboration network, where the top part of high peaks contains dense K-Cores (we click the circled part of the high peak and the red box shows the details of a dense K-Core). In Figure 4.1(b), we use community score [81] as a measure to visualize the four communities in a DBLP network, where each major peak is a community, and the sub-peaks in a major peak indicate sub-communities. The top part of a peak contains key members of the community. We can also color the terrain using a second measure, so we can analyze the correlation between two different measures on the graph. More details of the two examples are in Section 4.4.

The work flow of our visual platform is as follows, we first convert the scalar graph into a tree structure which captures the distribution of scalar values over the graph. For scalar graphs with numerical measures defined on vertices, we convert the scalar graph into a scalar tree by adapting the contour tree algorithm in [20, 79]. For scalar graphs with numerical measures defined on edges, we design a novel efficient algorithm to construct an edge-scalar tree of the scalar graph that is subsequently visualized using the terrain
metaphor [27, 39, 40]. Besides visualizing a single numerical measure on a graph, we also propose methods to analyze and visualize correlation between multiple numerical measures on a graph.

In summary, our contributions are:

- We propose a novel visual platform in 3D space to visualize a graph whose vertices/edges are associated with numerical measures. The visual platform enables users to quickly capture how the scalar value is distributed in the graph.

- We propose a new concept – maximal $\alpha$-connected component, and use it to analyze a scalar graph. We demonstrate that for different numerical measures, maximal $\alpha$-connected components can be mapped to various graph patterns and motifs-of-interest (dense subgraphs, communities, K-Cores, K-Truss etc.). We also show that our visualization not only captures the distribution of K-Core/K-Trusses, but also their global relationships.
Based on our visual platform, we propose methods to analyze the relation between multiple numerical measures on one graph.

Finally, we empirically demonstrate that the visual platform is both general-purpose and scalable and can handle graphs with millions of nodes and edges.

4.2 Single Scalar Field

4.2.1 Vertex-based Scalar Graph

We define a vertex-based scalar graph $G(V, E)$ as a graph comprising edges $E$ and vertices $V$, where each vertex $v$ has one scalar value associated with it, denoted as $v.\text{scalar}$. In following we refer to vertex-based scalar graph as \textit{scalargraph} for short.

On scalar graphs we define a \textbf{maximal $\alpha$-connected component} as follows:

\textbf{Definition 9.} A connected component $C(V_C, E_C)$ of scalar graph $G(V, E)$ is a \textbf{maximal $\alpha$-connected component} if it satisfies following conditions:

(1) for every vertex $v \in V_C$, $v.\text{scalar} \geq \alpha$.

(2) for any vertex $v \in V_C$, if $v'$ is connected to $v$ and $v' \notin V_C$, then $v'.\text{scalar} < \alpha$.

(3) for any edge $e(v_1, v_2) \in E$, if $v_1 \in V_C$ and $v_2 \in V_C$, then $e(v_1, v_2) \in E_C$.

The reason we are interested in maximal $\alpha$-connected components is: they contain both connectivity and scalar value information, so the distribution of maximal $\alpha$-connected components in a graph reflects the distribution of the scalar values over the scalar graph. Also, when the scalar value is defined to be a specific attribute, a single maximal $\alpha$-connected component will be a subgraph pattern of particular interest. For example, when the scalar value of each vertex is its maximum K-Core number [85], a maximal $\alpha$-connected component will be a K-Core where $K = \alpha$, as will be explained later.
To analyze the maximal $\alpha$-connected components, we might be interested in the following questions: for a particular $\alpha$ value, how many maximal $\alpha$-connected components are there, and what are their sizes? What is the relationship between a maximal $\alpha$-connected component and another maximal $\alpha$-connected component? For example, when each maximal $\alpha$-connected component is a K-Core, we would be interested in the overall distribution of all the K-Cores – which area of the graph contains dense/sparse K-Cores? Is a K-Core connected/disconnected to another K-Core, or is it a subgraph of another K-Core? In the next section, we propose to use a tree structure to help users answer the questions quickly.

Since there are infinite number of $\alpha$ values, analyzing all maximal $\alpha$-connected components could be difficult. We explore the following properties of maximal $\alpha$-connected components to help reduce the problem size.

**Notation:** for two subgraphs $G'(V', E')$ and $G''(V'', E'')$ of scalar graph $G$, $G'$ is the same as $G''$ (denoted as $G' = G''$) if $V' = V''$ and $E' = E''$, $G'$ is a subgraph of $G''$ (denoted as $G' \subseteq G''$) if $V' \subseteq V''$ and $E' \subseteq E''$, $G'$ is connected to $G''$ (denoted as $G' \leftrightarrow G''$) if there is a vertex $v' \in V'$ and a vertex $v'' \in V''$ such that $v'$ is connected to $v''$.

**Definition 10.** For vertex $v$, the maximal $\alpha$-connected component of $v$ is the maximal $v$:scalar-connected component containing $v$, and it is denoted as $\text{MCC}(v)$.

**Theorem 2.** For any maximal $\alpha$-connected component $C$ in $G$, there is a vertex $v$ in $G$ such that $\text{MCC}(v) = C$.

**Proof Sketch:** Assume in maximal $\alpha$-connected component $C$, vertex $v$ has the minimum scalar value, then we can easily show that $C = \text{MCC}(v)$. □

**Theorem 3.** For two vertices $v$ and $v'$, if $v$:scalar = $v'$:scalar and $\text{MCC}(v)$ contains $v'$, then $\text{MCC}(v) = \text{MCC}(v')$. 

65
**Proof Sketch:** For every vertex \( v_i \) in \( MCC(v) \), we can show that in \( MCC(v) \) there is a path \( v_i \rightarrow ... \rightarrow v \rightarrow ... \rightarrow v' \), which starts with \( v_i \), passes through \( v \), and ends at \( v' \), and all the vertices on the path have scalar value greater than or equal to \( v'.scalar \), so \( v_i \) is in \( MCC(v') \). Similarly, we can show for every vertex \( v_j \) in \( MCC(v') \), \( v_j \) is in \( MCC(v) \). Thus \( MCC(v) = MCC(v') \). □

**Theorem 4.** For a maximal \( \alpha_1 \)-connected component \( C_1 \) and another maximal \( \alpha_2 \)-connected component \( C_2 \), if \( C_1 \leftrightarrow C_2 \) then \( C_1 \subseteq C_2 \) or \( C_2 \subseteq C_1 \).

**Proof Sketch:** Based on Theorem 2, there are two vertices \( v_1 \) and \( v_2 \) such that \( MCC(v_1) = C_1 \) and \( MCC(v_2) = C_2 \). Assume \( v_1.scalar \leq v_2.scalar \), since \( MCC(v_1) \leftrightarrow MCC(v_2) \), for any vertex \( v' \) in \( MCC(v_2) \), we can find a path \( v_1 \rightarrow ... \rightarrow v' \) connecting \( v_1 \) and \( v' \), and all the vertices on the path have scalar value greater than or equal to \( v_1.scalar \), so \( v' \) is in \( MCC(v_1) \). Thus \( MCC(v_2) \subseteq MCC(v_1) \). □

Theorem 2 and 3 shows that in a scalar graph \( G \), the number of distinct maximal \( \alpha \)-connected components is limited – no more than the number of vertices in \( G \). Theorem 4 shows the relationship between different maximal \( \alpha \)-connected components.

**4.2.2 Vertex Scalar Tree**

**Properties of Vertex Scalar Tree**

In this section, we introduce the vertex scalar tree (scalar tree for short) to analyze a scalar graph. Intuitively, the scalar tree is the same as the so-called split tree (a variant of contour tree) which is studied in computational geometry and applied to a piecewise-linear function on a 1-dimensional simplicial complex [20, 79]. Here we interpret scalar tree (split tree) from the perspective of the maximal \( \alpha \)-connected components, in order to better understand the properties of the scalar tree in the context of analyzing a scalar graph. We
note that this perspective also helps us easily extend the notion of scalar tree for edge-based scalar graphs (in Section 2.3).

A scalar tree is a tree whose node is associated with a scalar value, and the scalar tree $T$ of scalar graph $G$ has the following properties:

1. Every node in $T$ corresponds to a vertex in $G$ with the same scalar value, and vice versa (i.e. one-to-one correspondence).

2. Every maximal $\alpha$-connected component in $G$ corresponds to a subtree in $T$, and vice versa (i.e. one-to-one correspondence).

3. Assume a maximal $\alpha_1$-connected component $C_1$ corresponds to subtree $T_1$ in $T$, and another maximal $\alpha_2$-connected component $C_2$ corresponds to subtree $T_2$ in $T$, then $C_1$ is a subgraph of $C_2$ if and only if $T_1$ is subtree of $T_2$.

4. Assume a maximal $\alpha_1$-connected component $C_1$ corresponds to subtree $T_1$ in $T$, and another maximal $\alpha_2$-connected component $C_2$ corresponds to subtree $T_2$ in $T$, then $C_1$ and $C_2$ are not connected if and only if $T_1$ and $T_2$ are not connected.

**Notation:** In the following text, the scalar tree node corresponding to vertex $v$ is denoted as $n(v)$, and the vertex corresponding to scalar tree node $n$ is denoted as $v(n)$. The parent of tree node $n$ is denoted as $\text{parent}(n)$. The subtree corresponding to a maximal $\alpha$-connected component $C$ is denoted as $ST(C)$. A subtree $ST$ containing nodes $n_1, n_2, \ldots, n_k$ is denoted as $ST(n_1, n_2, \ldots, n_k)$, and a connected component $C$ containing vertices $v_1, v_2, \ldots, v_k$ is denoted as $C(v_1, v_2, \ldots, v_k)$.

In Figure 4.2 we use an example to illustrate the properties of scalar tree. Figure 4.2(a) is a scalar graph $G$, in which every vertex’s label is its scalar value. Figure 4.2(b) is a
correspondent scalar tree \( T \) of Figure 4.2(a), rooted at node \( n_9 \). Node \( n_i \) in Figure 4.2(b) corresponds to vertex \( v_i \) in Figure 4.2(a) (Property 1). In Figure 4.2(c), we extract all the maximal 2.5-connected components of \( G \): \( C_1(v_1, v_2, v_3, v_5) \) and \( C_2(v_4, v_6) \), and in Figure 4.2(b), their correspondent subtrees are: \( ST(C_1) = ST(n_1, n_2, n_3, n_5) \) and \( ST(C_2) = ST(n_4, n_6) \), this satisfies Property 2. We notice that \( C_1 \) and \( C_2 \) are not connected, and \( ST(C_1) \) and \( ST(C_2) \) are not connected either, this satisfies Property 4. We also observe that \( C_1 \) is a subgraph of a maximal 2-connected component \( C_3(v_1, v_2, v_3, v_4, v_5, v_6, v_7) \), and \( ST(C_1) \) is a subtree of \( ST(C_3) = ST(n_1, n_2, n_3, n_4, n_5, n_6, n_7) \), this satisfies Property 3.

![Scalar Graph G](image)

![Scalar Tree T](image)

![maximal 2.5-connected components](image)

Figure 4.2: Scalar Graph and Scalar Tree

From the properties above, we can see that scalar tree captures the distribution and relationship of all maximal \( \alpha \)-connected components in a scalar graph, so we can analyze the scalar tree instead of the scalar graph. This will facilitate the analysis of maximal \( \alpha \)-connected components, and another benefit is, it is easier to visualize a scalar tree than a scalar graph.
**Construct Vertex Scalar Tree**

Algorithm 13 constructs the scalar tree, it is actually the algorithm of [20, 79] to compute the split tree. We include it below for completeness. Algorithm 13 processes all the vertices in decreasing order of scalar values (line 1-3). If the current vertex \( v_i \) is connected to a previously processed vertex \( v_j \), but \( n(v_i) \) is not in the current subtree of \( n(v_j) \) (line 4-5), then we connect \( n(v_i) \) with the root of the current subtree of \( n(v_j) \) (line 6). Here \( root(n(v_j)) \) denotes the root node of the current subtree containing \( n(v_j) \). Note that now \( n(v_i) \) is parent of \( root(n(v_j)) \), so \( n(v_i) \) becomes the new root of the current subtree containing \( n(v_j) \).

---

**Algorithm 13 Constructing Scalar Tree**

**Require:** A scalar graph \( G(V, E) \).

**Ensure:** The scalar tree \( T \) of \( G \).

1: Sort vertices in decreasing order of scalar values, the sorted vertices are \( v_1, v_2, ... v_n \);
2: Create a tree node \( n(v_i) \) for each vertex \( v_i \);
3: for \( i = 1 \) to \( n \) do
4: for every neighbor \( v_j \) of \( v_i \) do
5: if \( j < i \) and currently \( n(v_i) \) and \( n(v_j) \) are not in the same subtree then
6: Connect \( n(v_i) \) to \( root(n(v_j)) \); \( \forall n(v_i) \) is parent

The running time of line 1 is \( O(|V| \times \log|V|) \). An efficient implementation of line 5 is the idea of "Union Find", which compares \( root(n(v_i)) \) and \( root(n(v_j)) \). The worst case running time is \( O(\log|V|) \) for line 5, and line 5 is executed \( O(|E|) \) times. So the worst case running time for Algorithm 13 is \( O(|E| \times \log|V| + |V| \times \log|V|) \). The amortized time for "Union Find" (line 5) is \( O(\alpha(n)) \), where \( \alpha(n) \) is inverse of Ackermann function, and is usually a small constant. So the amortized time cost of Algorithm 13 is \( O(|E| \times \alpha(n) + |V| \times \log|V|) \).
In the scalar tree $T$ generated by Algorithm 13, every node’s scalar value is greater than or equal to its parent’s scalar value. If we layout a scalar tree $T$ in such a way that the height of each node is the scalar value of the node, then we could get all the maximal $\alpha$-connected components for a particular $\alpha$ in the following way: draw a line with $\text{height} = \alpha$ to cross $T$, and each of the subtrees above the line corresponds to a maximal $\alpha$-connected component. For example, as Figure 4.2(b) shows, the two subtrees above red line $\text{height} = 2.5$ correspond to all the maximal 2.5-connected components.

When every vertex in the input scalar graph $G$ has a distinct scalar value, we can prove the following proposition for scalar tree $T$ generated by Algorithm 13:

**Proposition 1.** For every vertex $v$ in $G$, assume it corresponds to $n(v)$ in $T$, then the subtree rooted at $n(v)$ (denoted as $ST$) in $T$ corresponds to the MCC($v$).

**Proof Sketch:** Obviously $ST$ corresponds to a connected component $SG$. Since $n(v).\text{scalar}$ is the minimum scalar value in $ST$, every vertex in $SG$ has scalar value greater than or equal to $v.\text{scalar}$. If there is a vertex $v_i$ that connects to a vertex $v_j$ in $SG$, and $v_i.\text{scalar} > v.\text{scalar}$, then due to Algorithm 13, $n(v_i)$ and $n(v_j)$ will be in the same subtree $ST$, which indicates that $v_i$ is in $SG$. So $SG$ is a maximal $v.\text{scalar}$-connected component, and it is MCC($v$). $\square$

Based on Theorem 2, Theorem 4 and Proposition 1, we can easily prove that when every vertex in the input scalar graph has a distinct value, the tree generated by Algorithm 13 has the four properties of the scalar tree.

When some vertices in scalar graph have the same scalar value, Algorithm 13 may not guarantee Property 2, and we need to do some postprocessing.
**Postprocess Vertex Scalar Tree**

If some vertices in G have the same scalar values, the scalar tree generated by Algorithm 13 may not satisfy Property 2 – a subtree may not correspond to a maximal \( \alpha \)-connected component. For example, Figure 4.3(a) is a scalar graph, and Figure 4.3(b) is the scalar tree generated by Algorithm 13, in which \( n_i \) corresponds to \( v_i \) in the scalar graph. The subtree rooted at \( n_3 \) is \( ST(n_1, n_3) \), it corresponds to connected component \( C(v_1, v_3) \), however, \( C(v_1, v_3) \) is not a maximal \( \alpha \)-connected component.

![Figure 4.3: Postprocessing Scalar Tree](image)

The problem is caused by the fact that Proposition 1 does not hold when there are duplicate scalar values. To solve the problem, we use Algorithm 14 to postprocess the scalar tree \( T \) generated by Algorithm 13. The algorithm is based on the following proposition:

**Proposition 2.** For every tree node \( n \) in \( T \), assume it has an ancestor \( n_{\text{anc}} \) in \( T \), such that \( n_{\text{anc}}:\text{scalar} = n:\text{scalar} \), and \( \text{parent}(n_{\text{anc}}) \) is null or \( \text{parent}(n_{\text{anc}}):\text{scalar} < n_{\text{anc}}:\text{scalar} \), then the subtree rooted at \( n_{\text{anc}} \) (denoted as \( ST_{\text{anc}} \)) corresponds to the MCC(\( v(n) \)).

**Proof Sketch:** Obviously the subtree rooted at \( n_{\text{anc}} \) is MCC(\( v(n_{\text{anc}}) \)). Since subtree \( ST_{\text{anc}} \) contains node \( n \), MCC(\( v(n_{\text{anc}}) \)) contains vertex \( v(n) \). Because \( n_{\text{anc}}:\text{scalar} = n:\text{scalar} \),
$n\text{.scalar}$, from Theorem 3, we can get that $MCC(v(n_{\text{anc}})) = MCC(v(n))$. The proposition is true. □

Based on Proposition 2, in Algorithm 14, we merge the ancestor node with all its descendants with the same scalar value into a super node, and Proposition 1 still holds for the postprocessed tree.

In the example in Figure 4.3(b), Algorithm 14 will merge the nodes $n_3, n_4, n_5$ into a super node, and build a super tree (Figure 4.3(c)). Every subtree of the super tree will correspond to a maximal $\alpha$-connected component. After postprocessing, the scalar tree will still satisfy Property 2, 3, 4, but may not satisfy Property 1, because a super node may correspond to multiple vertices, however, this does not affect the further analysis.

Algorithm 14 only needs one pass of the scalar tree $T$, so the time complexity is $O(|V|)$, where $|V|$ is the number of vertices in the original scalar graph.

### 4.2.3 Edge-based Scalar Graph

Similar to the approach described for modeling scalar values on vertices, here we describe an approach for modeling scalar values on edges. We define an edge-based scalar graph $G(V, E)$ as a graph comprising edges $E$ and vertices $V$, where each edge $e$ has one scalar value associated with it, denoted as $e\text{.scalar}$.

Similarly, the maximal $\alpha$-edge connected component is defined below:

**Definition 11.** A connected component $C(V_C, E_C)$ of scalar graph $G(V, E)$ is a maximal $\alpha$-edge connected component if it satisfies following conditions:

1. for every edge $e \in E_C$, $e\text{.scalar} \geq \alpha$.

2. for any edge $e \in E_C$, if edge $e'$ shares a common vertex with $e$ and $e' \notin E_C$, then
Algorithm 14 Postprocess Scalar Tree

Require: The scalar tree $T$ generated by Algorithm 1.
Ensure: The super scalar tree $T_{\text{super}}$.

1: Create an array $\text{ancestors}$;
2: Assume the root of $T$ is $n_r$, $n_r\text._\text{super}_\text{node} = \text{new} \ Node()$;
3: $\text{ancestors}.\text{add}(n_r)$;
4: $T_{\text{super}}.\text{add}(n_r\text._\text{super}_\text{node})$;
5: for each node $n_{\text{anc}}$ in $\text{ancestors}$ do
   6: Create a queue $Q$;
   7: $Q.\text{push}(n_{\text{anc}})$;
   8: while !$Q.\text{empty}()$ do
      9: $n_q = Q.\text{pop}()$;
   10: $n_{\text{anc}}.\text{super}_\text{node}.\text{members}.\text{add}(n_q)$;
   11: for every child $n_c$ of $n_q$ do
      12: if $n_c.\text{scalar} = n_q.\text{scalar}$ then
         13: $Q.\text{push}(n_c)$;
      else
         14: $n_c.\text{super}_\text{node} = \text{new} \ Node()$;
         15: $\text{ancestors}.\text{add}(n_c)$;
         16: $T_{\text{super}}.\text{add}(n_c.\text{super}_\text{node})$;
         17: Connect $n_{\text{anc}}.\text{super}_\text{node}$ with $n_c.\text{super}_\text{node}$;
   18: end for
   end while

$e'.\text{scalar} < \alpha$.

(3) for any edge $e(v_1, v_2) \in E_C$, we have $v_1 \in V_C$ and $v_2 \in V_C$.

The edge scalar tree $T$ of edge scalar graph $G$ has the following properties:

1. Every node in $T$ corresponds to an edge in $G$ with the same scalar value, and vice versa (i.e. one-to-one correspondence).

2. Every maximal $\alpha$-edge connected component in $G$ corresponds to a subtree in $T$, and vice versa (i.e. one-to-one correspondence).

3. Assume a maximal $\alpha_1$-edge connected component $C_1$ corresponds to subtree $T_1$ in $T$, and another maximal $\alpha_2$-edge connected component $C_2$ corresponds to subtree $T_2$ in $T$, then $C_1$ is a subgraph of $C_2$ if and only if $T_1$ is subtree of $T_2$. 

73
4. Assume a maximal $\alpha_1$-edge connected component $C_1$ corresponds to subtree $T_1$ in $T$, and another maximal $\alpha_2$-edge connected component $C_2$ corresponds to subtree $T_2$ in $T$, then $C_1$ and $C_2$ are not connected if and only if $T_1$ and $T_2$ are not connected.

**Naive Method:** a straightforward way to build edge scalar tree is: first we convert edge-based scalar graph $G(V, E)$ to a dual graph $G_d(V_d, E_d)$—every edge in $G$ is converted to be a vertex in $G_d$, and if two edges in $G$ share a common vertex, their correspondent vertices in $G_d$ are connected. Then we apply Algorithm 1 to $G_d$, we can easily prove that the generated tree is an edge scalar tree of $G$.

The worst time complexity of the method is $O(|E_d| \times \log |V_d|)$. In dual graph $G_d$, we can get that $|V_d| = |E|$, and $|E_d| = O(\sum_{v \in V} \text{degree}(v)^2)$ so the time complexity is $O(\sum_{v \in V} \text{degree}(v)^2 \times \log |E| + |E| \times \log |E|)$. The time cost is high because of the bottleneck $O(\sum_{v \in V} \text{degree}(v)^2 \times \log |E|)$.

**Improved Method:** we propose a more efficient method (Algorithm 15) to construct edge scalar tree from edge scalar graph, and the time complexity is reduced to be $O(|E| \times \log |E|)$. In line 1, we sorted all the edges in descending order of scalar value. In line 2-3, we select the $\text{min}_id\_edge$ on vertex $v$ that has the minimum index. In line 6-9, we process edge $e_i$. Instead of checking all $e_i$’s neighbor edges (edges which share a common vertex with $e_i$) , we just need to check the $\text{min}_id\_edges$ of $e_i$’s two vertices (line 6-7). This is based on Proposition 3:

**Proposition 3.** If edge $e_j$ is a neighbor edge of $e_i$ ($i > j$), and they share the same vertex $v$, when processing $e_i$ in line 6-9 of Algorithm 15, root$(n(e_j))$ is the same as root$(n(v.min_id\_edge))$.  

74
Proof. Since \( i > j \), \( e_j \) will be processed before \( e_i \). When processing \( e_j \) in line 6-9 of Algorithm 15, \( n(e_j) \) will be connected \( \text{root}(n(v.min\_id\_edge)) \), which means \( n(e_j) \) and \( n(v.min\_id\_edge) \) will be in the same subtree thereafter. So when processing \( e_i \), \( \text{root}(n(e_j)) \) is the same as \( \text{root}(n(v.min\_id\_edge)) \). □

**Algorithm 15** Constructing Edge Scalar Tree

**Require:** An edge scalar graph \( G(V, E) \).

**Ensure:** The edge scalar tree \( T \) of \( G \).

1. Sort edges in decreasing order of scalar values, the sorted edges are \( e_1, e_2, \ldots e_n \).
2. for each vertex \( v \) in \( G \) do
3. set \( v.min\_id\_edge \) to be the edge on \( v \) that has the minimum index.
4. Create a tree node \( n(e) \) for each edge \( e \).
5. for \( i = 1 \) to \( n \) do
6. Assume \( e_i \) has two vertices \( v_1 \) and \( v_2 \), create an array:
    \[
    \text{min\_neighbors}[2] = \{v_1.min\_id\_edge, v_2.min\_id\_edge\};
    \]
7. for each edge \( e_m \) in \( \text{min\_neighbors} \) do
8. if \( m < i \) and currently \( n(e_i) \) and \( n(e_m) \) are not in the same subtree then
9. connect \( n(e_i) \) to \( \text{root}(n(e_m)) \) if \( n(e_i) \) is parent

The time complexity of line 1 in Algorithm 15 is \( O(|E| * \log|E|) \). For each edge \( e \), line 8 is executed \( O(1) \) times, so line 8 is executed a total of \( O(|E|) \) times, and the total running time of line 5-9 is \( O(|E| * \log|E|) \). The worst case running time of Algorithm 15 is \( O(|E| * \log|E|) \).

When some edges have the same scalar value, we can also use Algorithm 14 to post process the edge scalar tree.

### 4.2.4 Relation between maximal \( \alpha \)-(edge) connected component and Dense Subgraph

Dense subgraph is a connected subgraph in which every vertex is heavily connected to other vertices in the subgraph. K-Core [12] and K-Truss [72] (also called Triangle K-Core
in [85], DN-graph in [75]) are two dense subgraph patterns that draw much attention in recent works. The definitions of K-Core and K-Truss are as follows:

**Definition 12.** A K-Core is a subgraph in which each vertex participates in at least $K$ edges within the subgraph. The maximal K-Core of a vertex $v$ is the K-Core containing $v$ that has the maximum $K$ value, and the $K$ value of maximal K-Core of $v$ is denoted as $KC(v)$.

**Definition 13.** A K-Truss is a subgraph in which each edge participates in at least $K$ triangles within the subgraph. The maximal K-Truss of an edge $e$ is the K-Truss containing $e$ that has the maximum $K$ value, and the $K$ value of maximal K-Truss of $e$ is denoted as $KT(e)$.

In our paper we assume the K-Core and K-Truss are connected subgraphs. Now we prove the relationship between maximal $\alpha$-(edge) connected component and K-Core, K-Truss below.

**Proposition 4.** In a scalar graph $G$ where for any vertex $v$, $v.scalar = KC(v)$, a maximal $\alpha$-connected component in $G$ is a K-Core where $K = \alpha$.

**Proof.** Assume in a maximal $\alpha$-connected component $C$, vertex $v$ has the minimum scalar value. Based on the definition of K-Core, for every vertex $v'$ in the maximal K-Core of $v$, $KC(v') \geq KC(v)$, so the maximal K-Core of $v$ is a subgraph of $C$. Since $v$ is connected to at least $KC(v)$ vertices in its maximal K-Core, $v$ is connected to at least $KC(v)$ vertices in $C$. For every other vertex $v'$ in $C$, similarly we can get that $v'$ is also connected to at least $KC(v)$ vertices in $C$. So $C$ is a K-Core where $K = KC(v)$, since $KC(v) = v.scalar \geq \alpha$, C is also a K-Core where $K = \alpha$. 

**Proposition 5.** In an edge scalar graph $G$ where for any edge $e$, $e.scalar = KT(e)$, a maximal $\alpha$-edge connected component in $G$ is a K-Truss where $K = \alpha$. 

Proof. Assume in a maximal $\alpha$-edge connected component $EC$, edge $e$ has the minimum scalar value. Based on the definition of K-Truss, for every edge $e'$ in the maximal K-Truss of $e$, $KT(e') \geq KT(e)$, so the maximal K-Truss of $e$ is a subgraph of $EC$. Since $e$ participates in at least $KT(e)$ triangles in its maximal K-Truss, $e$ participates in at least $KT(e)$ triangles in $EC$. For every other edge $e'$ in $EC$, similarly we can get that $e'$ also participates in at least $KT(e)$ triangles in $EC$. So $EC$ is a K-Truss where $K = KT(e)$, since $KT(e) = e.scalar \geq \alpha$, $EC$ is also a K-Truss where $K = \alpha$.

When we define the scalar value of each vertex/edge to be $KC(v)/KT(e)$, the (edge) scalar tree will capture the distribution and relationships of K-Cores/K-Trusses in the graph.

4.2.5 Visualizing Scalar Trees via Terrain Metaphor

Scalar trees are usually not easy to visually interpret, especially when the size of the tree is too large. We adapt the terrain metaphor visualization technique to visualize the scalar trees [40]. In Figure 4.4 we use an example to illustrate how to convert the scalar tree in Figure 4.4(a) to terrain visualization in Figure 4.4(c). In Figure 4.4(b) we first layout all the tree nodes of Figure 4.4(a) in a 2D plane, every node $n_i$ is represented by a boundary $b_i$ in the 2D plane, and the area enclosed by the boundary $b_i$ is proportional to the number of nodes in subtree (not including $n_i$) rooted at $n_i$. To generate the 2D layout, we start traversing the tree from the root(bottom) node $n_9$, draw the outermost boundary $b_9$ to represent it. Then we move to $n_8$, and draw a boundary $b_8$ inside $b_9$. When we reach $n_7$, and draw boundary $b_7$, we find there are two subtrees rooted at $n_7$, so we split the area inside $b_7$ into 2 areas, and recursively layout each subtree in each area. When we reach
leaf nodes \( n_1, n_2, n_4 \), since the size of their subtrees is 0, their correspondent boundaries degenerate to be points.

To convert the 2D layout (Figure 4.4(b)) into a terrain visualization in 3D space (Figure 4.4(c)), we first escalate each boundary \( b_i \) in Figure 4.4(b) to the height of \( n_1, \text{scalar} \), and then draw a “wall” between neighboring boundaries. Finally we generate a terrain in Figure 4.4(c). We can color the terrain by assigning each vertex a color value, and since each “wall” is confined by two boundaries \( b_i \) and \( b_j \), we color the wall based on the color value of the vertex corresponding to \( b_i \) or \( b_j \).

To identify a subtree of the scalar tree in the terrain visualization, we locate the correspondent boundary \( b_r \) of the subtree root \( n_r \), and the terrain area within the boundary \( b_r \) corresponds to the subtree. In our paper, we define a \( \text{peak}_\alpha \) in terrain as below:

**Definition 14.** A \( \text{peak}_\alpha \) is the terrain area within a boundary whose height is \( \alpha \).

Since each \( \text{peak}_\alpha \) corresponds to a subtree in scalar tree, we can easily get the following properties of a \( \text{peak}_\alpha \):

1. A \( \text{peak}_\alpha \) corresponds to a maximal \( \alpha \)-connected component (denoted as \( C(\text{peak}_\alpha) \)) in the scalar graph and vice versa.

2. A \( \text{peak}_\alpha \) is contained in another \( \text{peak}_{\alpha'} \) if and only if \( C(\text{peak}_\alpha) \) is a subgraph of \( C(\text{peak}_{\alpha'}) \).

3. A \( \text{peak}_\alpha \) and another \( \text{peak}_{\alpha'} \) are not connected if and only if \( C(\text{peak}_\alpha) \) and \( C(\text{peak}_{\alpha'}) \) are not connected.

For example, the red peak in Figure 4.4(f) is a \( \text{peak}_5 \), which corresponds to the maximal 5-connected component (red nodes) in Figure 4.4(d), and the red peak in Figure 4.4(i)
is a peak, which corresponds to the maximal 3-connected component (red nodes) in Figure 4.4(g). One peak may contain some sub-peaks, which indicates its maximal $\alpha$-connected component contains other maximal $\alpha'$-connected components. For example, peak 5 in Figure 4.4(f) is contained in peak 3 in Figure 4.4(i), this indicates that the correspondent maximal 5-connected component in Figure 4.4(d) is a subgraph of the maximal 3-connected component in Figure 4.4(g). In a peak, the area of its bottom boundary indicates the number of vertices in its correspondent maximal $\alpha$-connected component.

To get all the maximal $\alpha$-connected components for a particular $\alpha$ value, we can use a 2D plane with height = $\alpha$ to cross the terrain in 3D space, and all the peaks above the plane correspond to all maximal $\alpha$-connected components. The benefit of using terrain visualization is, it captures the overall information of all maximal $\alpha$-connected components in one picture. Also, we could encode more information in the terrain by using colors to the terrain.

If the scalar graph $G$ is not a connected graph, our method will generate multiple scalar trees from the scalar graph, with each scalar tree corresponding to a connected component in $G$. However, the terrain visualization software only accept one single connected tree as input, so we use a super root $r_s$ to connect the roots of all the scalar trees. The scalar value of the super root is the minimum scalar value of all the roots.

To visualize a super tree where each super node may contain multiple nodes, the software provides two methods. The first method is to give each super node a weight to indicate how many nodes in it. The second method is to use an edge to represent a super node, the two end nodes of the edge connects the super node’s parent and children respectively, and
the members of super node are added as edge members. In experiments we use the second method because it preserves member information. Software [27] also provides some function to simplify a terrain when the terrain is too complex.

Figure 4.4: Terrain Visualization of a Simple Scalar Tree
4.3 Handling Multiple Scalar Fields

On some graphs, multiple scalar fields can be defined, which means there are multiple scalar values defined on each vertex. For example, a vertex v has degree value and KC(v) value. Users might be interested in how the multiple scalar fields correlate with each other. In this section, we propose two indexes to measure the correlation of two scalar fields on a graph. Then we can use terrain visualization to analyze the relationship between two scalar fields.

**Local Correlation Index:** Assume we have two scalar fields, \( S_i \) and \( S_j \), defined on a graph. Each vertex \( v \) has scalar values \( v Scalar_i \) and \( v Scalar_j \) in the two scalar fields. Some previous work [61] proposed a measure to compute the correlation of multiple scalar fields in continuous domain, we adapt their method and propose Local Correlation Index to measure the correlation of two scalar fields on local areas of a graph. Here local area is defined as k-hop neighborhood of each vertex v (denoted as N(v)), for all experiments we limit this to be 1-hop. The Local Correlation Index of \( S_i \) and \( S_j \) on N(v) is denoted as \( LCI_{S_i,S_j}(v) \). For each vertex \( v \), we compute \( LCI_{S_i,S_j}(v) \) as follows.

\[
\begin{align*}
\text{\( v Scalar_i \)} &= \frac{\sum_{u \in N(v)} u Scalar_i}{|N(v)|} \\
Cov_{ij}(v) &= \frac{\sum_{u \in N(v)} (u Scalar_i - v Scalar_i) \cdot (u Scalar_j - v Scalar_j)}{|N(v)|} \\
LCI_{S_i,S_j}(v) &= \frac{Cov_{ij}(v)}{\sqrt{Cov_{ii}(v)} \cdot \sqrt{Cov_{jj}(v)}}
\end{align*}
\]

\( LCI_{S_i,S_j}(v) \) is actually the correlation of the scalar values of \( S_i \) and \( S_j \) on v’s k-hop neighborhood. This method can easily be adapted to analyze edge-based scalar graphs.
**Global Correlation Index:** We can compute the Global Correlation Index (GCI) of scalar fields $S_i$ and $S_j$ on a graph by averaging the Local Correlation Indexes of all neighborhoods.

$$GCI_{S_i,S_j}(G) = \sum_{v \in V} LCI_{S_i,S_j}(v)/|V|$$

By comparing the Global Correlation Index and Local Correlation Index, we may identify some outlier neighborhood on which the correlation of scalar fields $S_i$ and $S_j$ is different from the overall correlation.

**Terrain Visualization:** To visualize the local correlation between two scalar fields, we can use $LCI_{S_i,S_j}(v)$ as a new scalar field, and draw the terrain. This will show us the overall distribution of $LCI_{S_i,S_j}(v)$ over the graph, and help us identify the area of the graph that the two scalar fields are positively/negatively correlated.

We can also visually capture the global correlation of scalar fields $S_i$ and $S_j$ through coloring terrain visualization. We use one scalar field $S_i$ to draw the terrain visualization, and use the other scalar field $S_j$ to color the terrain (see Figure 4.1(a)). Please note that this method can also visualize the relationship between a numerical attribute and a nominal attribute of vertices, by coloring the terrain based on the value of nominal attributes.

### 4.4 Evaluation

We seek to address questions about the effectiveness (qualitative) and efficiency of the our interactive network visualization platform in this section. We leverage a wide range of datasets from the network science community as noted in Table 4.1. Heights in our terrain visualization represent scalar measures of input scalar graph while color represents
intensity of the same measure (unless otherwise noted). The color ranges from red (most intense); yellow (intense); green (less intense); blue (least intense).

Table 4.1: Dataset Properties

<table>
<thead>
<tr>
<th>Dataset</th>
<th># Nodes</th>
<th># Edges</th>
<th>Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>GrQc</td>
<td>5242</td>
<td>14496</td>
<td>Collaboration</td>
</tr>
<tr>
<td>Wikivote</td>
<td>7115</td>
<td>103689</td>
<td>Voting</td>
</tr>
<tr>
<td>Wikipedia</td>
<td>1,815,914</td>
<td>34,322,831</td>
<td>Encyclopedia</td>
</tr>
<tr>
<td>Cit-Patent</td>
<td>3,774,768</td>
<td>16,518,947</td>
<td>Patent Citation</td>
</tr>
<tr>
<td>Amazon</td>
<td>3,348,685</td>
<td>92,687,72</td>
<td>Co-Purchase</td>
</tr>
<tr>
<td>Astro</td>
<td>17,903</td>
<td>196,972</td>
<td>Collaboration</td>
</tr>
<tr>
<td>DBLP(sub)</td>
<td>27,989</td>
<td>66,832</td>
<td>Collaboration</td>
</tr>
</tbody>
</table>

4.4.1 Visualizing Dense Subgraphs

Effectiveness: The visualization of dense subgraphs within graphs has been of much interest within the database and information visualization. Examples abound and include CSV plots [74], K-Core [3] and Triangle K-Core (K-Truss) [85] plots. Here we use our terrain visualization to visualize K-Cores and K-Trusses and compare with the previous methods.

We consider two datasets (GrQc, Wikivote) for this illustration. In Figure 4.5(a) and Figure 4.5(b), we use the traditional spring layout algorithm [31] to draw both networks – it is hard to say anything about the distribution of dense subgraphs using such a plot. Following the discussion in Section 4.2.4, we use $KC(v)$ as scalar value, and generate the terrain visualization of both networks in Figure 4.5(c) and Figure 4.5(d). Recollect that if the scalar value of every vertex $v$ is defined to be $KC(v)$, each maximal $\alpha$-connected component is a K-Core where $K = \alpha$. Thus in the terrain, each $peak_{\alpha}$ is a K-Core where $K = \alpha$. The distribution of K-Cores in the two datasets is obviously different (and not apparent in corresponding spring layouts). Figure 4.5(d) shows that there is one single high peak, which means the network has one densest K-Core, and K-Core density gradually
decreases to the neighboring vertices. Figure 4.5(c) shows that there are several high peaks within the GrQc network, which means there are several disconnected K-Cores with high K values (dense K-Cores).

Moreover Figure 4.5(c) clearly illustrates the hierarchical relationship among K-Cores. In the selected terrain area (the terrain area in dashed line), the red peak is placed on green and blue foundation, which means the dense K-Core is contained in some less dense K-Cores. This can be verified in Figure 4.5(e) (with our tool, a user can select a region, and drill down to draw the nodes and edges of the selected region as illustrated in Figure 4.5(e)), the red dense K-Core is surrounded by some green and blue vertices. The visualization of hierarchy is important, as it allows an analyst to derive high level insights on the connectivity that is not immediately obvious even in state of the art K-Core plots as shown in Figure 4.5(g) [3] for the GrQc network.

Also we can color the terrain using a second measure. In Figure 4.1(a), we color the terrain based on vertex degree (red/yellow/green/blue area indicates vertices with highest/high/low/lowest degrees), we can see that generally KC(v) is positively correlated with degree – vertices in dense K-Cores have high degrees, except a few outlier vertices that have relatively high degree but low KC(v) values (the yellow area at the bottom of the terrain). They are usually local hub nodes with sparse neighborhood.

We can illustrate the same principle when visualizing K-Trusses (used to understand triangle density) instead of K-Cores. Here each edge uses KT(e) as the scalar measure, and we use the edge-based scalar graph for visualizing the K-Trusses in GrQc dataset. The terrain visualization is in Figure 4.5(f) where high peaks indicate dense K-Trusses. To contrast, Figure 4.5(h) depicts a CSV plot, a state-of-the-art density plot leveraged within
Figure 4.5: Visualizing K-Cores in GrQc and wikiVote network
the database community [74, 85]. Again such visualization strategies do not reveal important hierarchical relationships (e.g. contains) among different K-Trusses. Also we note that our visual platform is a common and flexible framework which can render plots based on different scalar measures. While it is hard to show, end-users of our tool have found the ability to rotate, filter and extract details on demand (allowing the analyst to quickly identify regions of interest) to be extremely useful as well.

**Scalability:** We next examine the efficiency of Algorithm 13, 14 and 15. Since for every dataset there are duplicate scalar values, so the generated trees are all super trees. We test our methods on datasets of various sizes, and list the number of nodes in the final super (edge) scalar tree \( N_t \), time cost to construct the tree (tc) and visualize the tree (tv) in Table 4.2. Also we list the time cost of the naive method (using dual graph) to build edge-scalar tree (te). We can see the improved method (tc) is much faster than the naive method (te), especially on the Wikipedia dataset, the improved method is more than 300 times faster than the naive method.

Additionally, in Figure 4.6, we show the terrain visualizations of (edge) scalar tree of wikipedia and cit-Patent datasets. The peaks in Figure 4.6(a) 4.6(b) 4.6(c) 4.6(d) indicate dense K-Cores and K-Trusses in the network, we highlight the highest peaks in Figure 4.6(b) and Figure 4.6(c), and draw the details in Figure 4.6(e) and Figure 4.6(f). They are a K-Truss with \( K = 86 \) and a K-Core with \( K = 64 \).

### 4.4.2 Visualizing Communities and Roles

**Visualizing Community Affiliation:** We next illustrate the flexibility of the terrain visualization scheme on an important network science task – understanding community structures (see Figure 4.7). We use a subset of the DBLP network (DBLP(sub)) for this purpose, comprising authors who publish in the areas of Machine Learning, Data mining, Databases and
Figure 4.6: Visualizing K-Cores and K-Trusses
Table 4.2: Terrain Visualization Time Cost (sec)

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Scalar</th>
<th>N</th>
<th>tc</th>
<th>te</th>
<th>tv</th>
</tr>
</thead>
<tbody>
<tr>
<td>GrQc</td>
<td>KC(v)</td>
<td>869</td>
<td>0.0018</td>
<td>&lt;1</td>
<td></td>
</tr>
<tr>
<td>GrQc</td>
<td>KT(e)</td>
<td>728</td>
<td>0.0039</td>
<td>&lt;1</td>
<td></td>
</tr>
<tr>
<td>WikiVote</td>
<td>KC(v)</td>
<td>106</td>
<td>0.0017</td>
<td>&lt;1</td>
<td></td>
</tr>
<tr>
<td>WikiVote</td>
<td>KT(e)</td>
<td>44</td>
<td>0.053</td>
<td>&lt;1</td>
<td></td>
</tr>
<tr>
<td>Wikipedia</td>
<td>KC(v)</td>
<td>230</td>
<td>6.9</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Wikipedia</td>
<td>KT(e)</td>
<td>1,903</td>
<td>49.3</td>
<td>163.34</td>
<td>22</td>
</tr>
<tr>
<td>Cit-Patent</td>
<td>KC(v)</td>
<td>1,059</td>
<td>7.1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Cit-Patent</td>
<td>KT(e)</td>
<td>110,412</td>
<td>27.7</td>
<td>65.3</td>
<td>13</td>
</tr>
</tbody>
</table>

Information Retrieval. We apply a state-of-the-art overlapping (soft) community detection algorithm [81] on this dataset to detect four communities. Each author in the dataset is affiliated with a community score vector \((c_0, c_1, c_2, c_3)\) indicating how much it belongs to each community. To visualize the affiliation of a particular community \(i\), we use \(c_i\) as the corresponding scalar measure, and draw the terrain of the network. \(peak_{c_i}\) in the terrain indicates a connected component in which every vertex has \(c_i \geq \alpha\).

In Figure 4.7(a), we visualize community 1, in which most authors are database researchers. We highlight two peaks in the circle of Figure 4.7(a), and zoom in to get a clear picture of the two peaks on the right. We find that authors in the left peak include researchers Donald Kossmann, Divyakant Agrawal, Amr El Abbadi, Michael Stonebraker, Samuel Madden, and Joseph M. Hellerstein while authors in the right peak include Zheng Chen, Hongjun Lu, Jeffrey Xu Yu, Beng Chin Ooi, Kian-Lee Tan, Qiang Yang and Aoying Zhou. Since authors in both peaks have high community scores \((c_1)\), they can be seen as core members of the community although from different geographic areas. The fact that they are in two separate peaks indicates that authors in one peak do not work with authors in the other peak in the dataset. Similarly, we also observe subcommunities in another community (Figure 4.7(b)) largely comprising Machine Learning researchers. We also find
two peaks in the terrain, and authors in the left peak are \textit{Philip S. Yu, Christos Faloutsos, Michael I. Jordan, Stuart J. Russell, Daphne Koller, Sebastian Thrun, Wei Fan and Andrew Y. Ng}, who all work in United States, while authors in the right peak are \textit{Hang Li, Ji-Rong Wen, Tie-Yan Liu, Lei Zhang, Wei-Ying Ma, Qiang Yang and Yong Yu}, who are researchers in China.

Figure 4.1(b) visualizes the four communities together to give an overview of them.

**Visualizing Role and Community Affiliation:** Moving beyond community affiliation, the ability to uncover the roles of individual nodes (e.g. bridge, hub, periphery and whisker) within a network or community has received recent interest \cite{41, 57}. Here we examine how one may use the terrain to visualize the distribution of roles over a community. We leverage a recent idea to simultaneously detect communities and roles on large scale networks \cite{57}. For each vertex in the network the algorithm outputs a community affinity vector \((c_1, \ldots, c_m)\) and a role affinity vector \((r_1, \ldots, r_n)\).
As before we focus on a particular community (community \(i\)) and use the community score \((c_i)\) of each vertex to create terrain visualization. The peak in Figure 4.8(a) contains the vertices affiliated with one major community in Amazon co-purchase network. Instead of re-using the intensity of community score \((c_i)\) to color vertices we actually use the dominant role for each vertex (four roles is typical [41]) to color vertices. We assign each role a color, the “hub vertex” is green, the “dense community vertex” is blue, the “periphery vertex” is red, and the “whisker vertex” is white. Then we assign the color of roles to the terrain in Figure 4.8(a). From the terrain visualization, we can see that the vertices in the community have 3 roles, the hub vertex has the highest community score (green top), and below it is the blue portion, which means the “hub vertex” is surrounded by some “dense community vertices” in the network. The red part of the peak indicates that there are some “peripheral vertices” attached to the community. Since the community contains a small number of vertices, we can draw the details of the community using node-link visualization in Figure 4.8(b).
Using our tool we drill down focusing on nodes that are a part of the peak in Figure 4.8(b), and list them in Table 4.3. All nodes are books on Amazon. The green node is the book which has the highest salesrank and is focused on creativity (hub). Most of the blue nodes are books about creativity (densely connected), while the red nodes are books loosely relevant to creativity (periphery).

<table>
<thead>
<tr>
<th>Role</th>
<th>Book Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>green</td>
<td>Artist’s Way, The PA</td>
</tr>
<tr>
<td>blue</td>
<td>The ARTISTS WAY AT WORK THE : Riding the Dragon: Twelve Weeks to Creative Freedom</td>
</tr>
<tr>
<td>blue</td>
<td>Heart Steps: Prayers and Declarations for a Creative Life</td>
</tr>
<tr>
<td>blue</td>
<td>Inspirations: Meditations from the Artist’s Way</td>
</tr>
<tr>
<td>blue</td>
<td>Reflections on the Artist’s Way</td>
</tr>
<tr>
<td>blue</td>
<td>The Artist’s Way Creativity Kit</td>
</tr>
<tr>
<td>red</td>
<td>Writing From the Inner Self</td>
</tr>
<tr>
<td>red</td>
<td>Codes of Love : How to Rethink Your Family and Remake Your Life</td>
</tr>
<tr>
<td>red</td>
<td>Heal Your Wounds and Find Your True Self : Finally A Book That Explains Why It’s So Hard Being Yourself</td>
</tr>
</tbody>
</table>

4.4.3 Comparing Different Centralities

In this section we examine the use of our approach for understanding the relationship of two different measures of centrality across various nodes within a network. We will compare two centralities, degree centrality and betweenness centrality, as two scalar fields, $S_d$ and $S_b$.

We use the Astro Physics collaboration network, in which each author is a vertex, and each edge indicates a coauthorship between two authors. We first compute the Local Correlation Index of each vertex (as described earlier), and then compute the Global Correlation Index of the network, $GCI_{S_d, S_b} = 0.89$. This indicates that the overall correlation between degree centrality and betweenness centrality is highly positive.
In this case, we are interested in those vertices with negative LCI values, as they could be seen as outliers. We define an outlier score for each vertex $v$ as follows:

$$outlier\_score(v) = -LCI(v)$$

the vertex with the most negative $LCI(v)$ will have the highest outlier score. We use $outlier\_score(v)$ as scalar field to draw the terrain in Figure 4.9(a), and we color the terrain using $S_d$ (degree centrality), where red/yellow/blue indicates high/moderate/low degree. Here $Peak_\alpha$ denotes a connected subgraph in which every vertex has an outlier score greater than or equal to $\alpha$. We notice that most high peaks are blue, which indicates that the outlier vertices usually have low degree.

We drill down into the two peaks in the terrain within black and red circles, and pick the vertex at the top of each peak. Their 2-hop neighborhoods are displayed in Figure 4.9(b) and Figure 4.9(c) respectively. We pick these two vertices specifically because one is in high peak while the other appears to be in a broader but smaller peak. In both cases the vertices picked have high outlier score, which indicates the correlation between degree centrality and betweenness centrality is negative in their neighborhood. Actually the two vertices have relatively higher betweenness and lower degree centrality when compared to many of their neighbors. From Figure 4.9(b) and Figure 4.9(c) we can see the two vertices (in the circles) are like bridge nodes connecting multiple communities.
(a) Terrain of Astro Network based on Outlierness

(b) 2-hop neighborhood of vertex in black circle in Figure 4.9(a)

(c) 2-hop neighborhood of vertex in red circle in Figure 4.9(a)

Figure 4.9: Compare Degree and Betweenness Centralities
4.4.4 Extension to Visualizing Non-graph Data

Our terrain visualization can also be extended to visualize non-graph data. Here we apply it to visualize a federated medical insurance billing dataset that contains transactions of customers purchase patterns at various pharmacies. We illustrate the potential of our terrain visualization strategy in two ways.

Visualize the whole dataset: we convert all the records in the dataset to a weighted graph $G(V, E)$ – each vertex represents a record, and for every pair of two records we compute their similarity $s$, and connect their correspondent vertices in $G$ by an edge with weight $s$. Then we build an edge scalar tree of the edge weighted graph $G(V, E)$, and visualize it through terrain visualization in Figure 4.10.

![Figure 4.10: Terrain Visualization of Medical Dataset](image)

Every peak in the terrain visualization indicates a cluster of records are that have high similarity, and may reveal something interesting. For example, the peak in the red circle represents three transactions of the same patient at three different times at the same pharmacy store, it actually shows the patient refilled the medicine three times. The peak in
the black circle represents two transactions of the same patient who bought medicine at a pharmacy store in the same day, this might be duplicate records.

**Visualize the query results:** the idea is, we first convert all the records in the database to a graph $G(V, E)$ – each vertex represents a record, and if the similarity of two records is above a threshold, their correspondent two vertices are connected by an edge in $E$.

Then we issue an query to the database, and return a result set $R$ of records. Assume the records in $R$ correspond to a vertex subset $V_r$ in $V$, and all the one-hop neighbors of vertices in $V_r$ comprise a subset $V_n$. We use vertex set $V_r$ and $V_n$ to construct a subgraph $G'$ from $G$.

Finally we visualize the subgraph $G'$ through terrain visualization, the scalar value of each vertex is the number of its connections to vertices in $V_r$. The scalar value indicates how much it is related to the result set. Those records that are not in the result set, but closely related to the result set might be worth further analysis.

We issue a query to the medical dataset to retrieve all the transactions whose patient id is “33003282052”, and generate the terrain visualization in Figure 4.11. The blue color indicates the transactions in the result set ($V_r$) and red color indicates the transactions in the neighborhood set ($V_n$). There are two peaks in the terrain. The right peak contains the patient’s transactions purchasing medicine “WARFARIN SODIUM”. The left peak contains the patient’s transactions purchasing other medicine. So the terrain visualization separates the query results into two clusters.

What is more, in the right peak, we see a red band in the top of peak, the red color indicates the transactions in the red band do not match the query, but the fact that they have high
scalar value indicates that they are very similar to the queried patient’s transactions. Actually, the red band represents two other patients’ transactions purchasing medicine “WARFARIN SODIUM”. So the terrain visualization not only returns records matching the query, but also records that are close to query results.

4.5 User Study

To evaluate our terrain visualization method and compare with other methods in depth, we conduct the following user study on some real world graphs. All the 10 users participating in the user study are graduate students in the College of Engineering at the Ohio State University.
4.5.1 Tasks

The first part of user study is to evaluate the single scalar field visualization on three graphs (GrQc, PPI, DBLP). For each graph, we compute the K-Core value (KC(v)) of each vertex, and visualize each graph using following three different visualization methods: (1) Our terrain visualization which uses K-Core value as scalar value. (2) LaNet-vi which is a K-Core visualization tool [3]. (3) OpenOrd which is a multilevel graph layout method [51]. In the OpenOrd visualization we use each vertex’s color to represent its K-Core value.

For each visualization of each graph, we ask the 10 users to perform two tasks:

(Task 1) Identify the densest K-Core in the graph.
(Task 2) Identify the densest K-Core in the graph that are not connected to the previously detected K-Core in Task 1.

The second part of user study is to evaluate the multiple scalar field visualization on Astro dataset. In terrain visualization, we use betweenness centrality of each vertex as the first scalar value to generate the terrain, and use the degree centrality of each vertex as the second scalar value to color the terrain (Figure 4.13(a)). In the visualization generated by OpenOrd, we use vertex color to indicate betweenness centrality, and vertex size to indicate degree centrality (Figure 4.13(b)). We ask the 10 users to perform the following task:

(Task 3) Determine whether the betweenness centrality and degree centrality are positively or negatively correlated in the Astro dataset.

Task 1 and Task 2 are meaningful to data mining researchers who want to explore K-Cores in a graph. The densest K-Core usually indicates a significant group of closely related nodes in the graph, such as an important community in a social network. To look for the second K-Core for analysis, simply choosing the second densest K-Core could be
meaningless, because it might be heavily overlapped with the densest K-Core, and the two
K-Cores are actually the same group of closely related nodes. It is more meaningful to
identify the densest K-Core in the graph that are not connected to the previously detected
one, because such a K-Core would indicate a group of nodes with different meaning. In
Task 3 we do not compare with LaNet-vi, because it is specifically designed for visualizing
K-Cores, and cannot be adapted to visualize two centralities.

4.5.2 Results

For Task 1 and Task 2, we list the average time and accuracy of all users in Table 4.4
and Table 4.5. All the pictures generated by the three visualization methods on the three
datasets are in Figure 4.12. In each picture we label the K-Core to be identified in Task
1/Task 2 as K1/K2.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Terrain (accuracy)</th>
<th>Terrain (time)</th>
<th>LaNet-vi (accuracy)</th>
<th>LaNet-vi (time)</th>
<th>OpenOrd (accuracy)</th>
<th>OpenOrd (time)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GrQc</td>
<td>1</td>
<td>2.6</td>
<td>1</td>
<td>6.7</td>
<td>1</td>
<td>7.6</td>
</tr>
<tr>
<td>PPI</td>
<td>1</td>
<td>4.9</td>
<td>1</td>
<td>5.3</td>
<td>0.8</td>
<td>10.7</td>
</tr>
<tr>
<td>DBLP</td>
<td>1</td>
<td>4.6</td>
<td>0.8</td>
<td>6.6</td>
<td>1</td>
<td>10.9</td>
</tr>
</tbody>
</table>

Table 4.5: Time Cost(sec) and Accuracy of Finishing Task 2

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Terrain (accuracy)</th>
<th>Terrain (time)</th>
<th>LaNet-vi (accuracy)</th>
<th>LaNet-vi (time)</th>
<th>OpenOrd (accuracy)</th>
<th>OpenOrd (time)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GrQc</td>
<td>1</td>
<td>3.6</td>
<td>0.7</td>
<td>10.3</td>
<td>0.8</td>
<td>8.2</td>
</tr>
<tr>
<td>PPI</td>
<td>1</td>
<td>4.1</td>
<td>0.2</td>
<td>7.7</td>
<td>0.7</td>
<td>11.6</td>
</tr>
<tr>
<td>DBLP</td>
<td>1</td>
<td>5.1</td>
<td>0.8</td>
<td>8.5</td>
<td>0.9</td>
<td>9.8</td>
</tr>
</tbody>
</table>
In Table 4.4, we can see that all users successfully finished Task 1 by using terrain visualization. Two users failed by using LaNet-vi on DBLP dataset, and two users failed by using OpenOrd on PPI dataset, because the densest K-Core in the two visualizations are very small, and they did not notice it or they couldn’t correctly identify the K-Core value through the color. Although they zoomed in the two visualizations to see a larger picture, they lost the full context and only see portion of the picture, this lead them to choose incorrect densest K-Cores. Table 4.5 shows that all users successfully finished Task 2 by terrain visualization, some users failed by using LaNet-vi and OpenOrd. One reason is Task 2 requires not only to identify the K-Core value but also the connections to the previous K-Core, the LaNet-vi and OpenOrd both draw edges to indicate connections, and users need to check the edges carefully to determine whether two K-Cores are connected, it is time consuming and users might make mistakes. In both tasks, we can see from Table 4.4 and Table 4.5 that users spent least time on terrain visualization.

The result of Task 3 is presented in the Table 4.6, and the visualization pictures are in Figure 4.13. The result shows that users achieved higher accuracy and spent less time on terrain visualization. In the visualization generated by OpenOrd, some nodes are blocked by other nodes, this affects users to make right decision. One user also pointed out that it is easier to identify the outlier areas (circle in Figure 4.13(a)) in the terrain visualization than in the OpenOrd visualization.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Terrain (accuracy)</th>
<th>Terrain (time)</th>
<th>OpenOrd (accuracy)</th>
<th>OpenOrd (time)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Astro</td>
<td>0.9</td>
<td>9.1</td>
<td>0.7</td>
<td>11.9</td>
</tr>
</tbody>
</table>

Table 4.6: Time Cost(sec) and Accuracy of Finishing Task 3
Figure 4.12: Compare Visualizations on GrQc, PPI and DBLP datasets

4.6 Discussion

Comparison with previous works: Many previous works have been proposed to visualize graph datasets [71]. Gronemann et al. [37] and Athenstadt et al. [8] both use topographic
map/landscape to visualize hierarchical clustering structures within a graph – each mountain corresponds to a cluster. The difference is Gronemann et al. [37] use the mountain height to indicate the hierarchy level, while Athenstadt et al. [8] use the mountain height to indicate the numerical attribute value of each vertex. Bourqui et al.[16] visualize hierarchical clusters of weighted graphs by using Voronoi diagram. van Liere et al.[70] propose the GraphSplatting method to visualize a graph as a 2D splat field, and use color to encode vertex density. Didimo et al.[24] propose a fast method to layout graph by combining space-filling and fast force-directed methods. Batagelj et al.[11] propose a hybrid visualization method (combining node-link and matrix-based representations) to visualize graph clusters. Telea et al.[69] generate a concise representation of graph by clustering edges and bundling similar edges together, These methods focused on analyzing graph topology and can efficiently reveal the clusters and dense areas of the graph. Some other visualization methods focused on visualizing graph attributes, Bezerianos et al. [15] propose an interactive visual system, GraphDice, to let users explore social networks with multiple node/edge
attributes. Some work [44, 30] used matrix or histogram to analyze attribute information in graphs.

One major difference between our work and the previous work is, we propose to analyze the graph through the hierarchical structure (scalar tree) induced by the maximal $\alpha$-connected components. Scalar tree partitions the graph in a different way than the clustering methods used in previous works. It highlights how scalar values evolve from high values to low values over the graph, this is particularly useful for some applications, for example, checking how a community is expanded from its core members to peripheral members (Figure 4.7). If we view the scalar graph as a scalar field, scalar tree is actually a natural way to partition scalar graph, because it is a variant of contour tree, which has been commonly used to visualize and explore scalar fields [35, 10, 66, 78, 67, 56, 19]. Another benefit of using maximal $\alpha$-connected components is, based on different attributes, the maximal $\alpha$-connected component can represent different subgraph patterns, such as K-core, K-truss, subcommunity, which attracted much interest in recent works [12, 72]. Thus our terrain visualization could be easily adapted to show the distribution of K-Cores/K-Trusses/subcommunities over the graph, and reveal the topological relationship (containment, connection) among the K-Cores/K-Trusses/subcommunities. Most previous works aim to visualizing general clusters, and they can’t be easily adapted to visualize these special subgraph patterns (K-Core, K-Truss etc.)

**2D Treemap vs 3D Terrain:** Besides 3D terrain visualization, we can easily generate a 2D treemap of scalar graph using Denali software. In particular, we can set the height of all boundaries in the 3D terrain to be 0, or alternatively, just view the 3D terrain top-down (in the direction of the height), then we see the projection of the terrain on 2D plane, which is the 2D treemap. In the 2D treemap, red/yellow/green/blue indicate highest/high/low/lowest
value, so the red/yellow blocks in the 2D treemap indicate the subgraph areas with high scalar values. Here we use the GrQc dataset (KC(v) as scalar value) as an example to compare the 2D and 3D visualization in Figure 4.14(a) and Figure 4.14(b).

From the 2D visualization, we could clearly see how the red/yellow/green blocks are distributed over the graph, while the 3D visualization could be less clear because of occlusion. So the 2D visualization may give a better overview of the distribution of dense k-cores in the graph.

However, the rotation/zoom in(out) manipulation of 3D terrain could alleviate the occlusion problem. Also, using color to encode scalar value may not be as accurate as using height. For example, peak 1 and peak 2 in Figure 4.14(b) correspond to block 1 and block 2 in Figure 4.14(a), from the height we can see that the peak 2 is a little higher than the peak 1, but from the color we cannot tell difference between block 1 and block 2. Finally, note that using height allows us to color the terrain based on a different attribute (from the one that generates the terrain), which helps to get a picture of the correlation between two attributes (see Figure 4.13(a)). Since our terrain visualization can easily generate 2D treemap, we can put 2D treemap and 3D terrain side by side, so users can have a better understanding of the data.

In summary, compared with previous works and 2D visualization, the unique information conveyed by terrain visualization is:

- The terrain reveals the hierarchical structure of maximal $\alpha$-connected components within a graph (the maximal $\alpha$-connected component can represent subgraph patterns like K-core, K-truss, etc.).

- The volume of the peak of terrain indicates the size of cluster of nodes/edges represented by the peak.
Figure 4.14: 2D Treemap vs. 3D Terrain

- The color of the terrain could be used to indicate a different attribute.
Chapter 5: Visually Analyzing Dynamic Graphs through Bitstream Representation

5.1 Introduction

Analysis of dynamically changing networks is always a challenging task in the graph mining area. In this chapter we propose to represent all the snapshots of a dynamic graph using bitstreams. As Figure 5.1 shows, every edge is associated with a bitstream, and each bit in the bitstream corresponds to a snapshot. The value of each bit (‘1’ or ‘0’) indicates whether the edge occurs or not in the corresponding snapshot. So the bitstreams actually encode the temporal information of all the edges. This idea was proposed in some previous work [36], and in our work we use the bitstreams to analyze the dynamic graph more in depth from the following two aspects:

1. Detecting Periodicities in the Dynamic Graph: The bitstream of each edge can be considered as a time series signal, then we can apply time series analysis methods to analyze it. In our work, we apply two popular methods (Discrete Fourier Transform and Autocorrelation) to each bitstream to identify major periods in it, and then generate a cumulative plot to reveal major periods in the whole dynamic graph. Based on the detected major periods, we propose an algorithm to detect communities that appear periodically in the dynamic graph.
2. Detecting Topologically and Temporally Correlated Edges: Many methods have been developed to detect clusters of vertices/edges that are topologically close. To analyze a dynamic graph, however, only considering topological features is insufficient. We are interested in clusters of vertices/edges that are not only topologically but also temporally correlated (i.e. co-occur in many snapshots). In our work, we convert the original dynamic graph to a weighted graph, and apply terrain visualization to it. The peaks in the terrain will represent clusters of topologically and temporally correlated edges in the original dynamic graph. Based on the terrain visualization, we develop a method to detect “dense blocks” in the bitstreams – a “dense block” usually indicates a dense community that frequently appears during a certain period.

5.2 Bitstream

A dynamic graph can be considered as a discrete sequence of snapshots, where two consecutive snapshots are separated by a time interval. The snapshot at time $t$ is represented as $G_t(V_t, E_t)$ where $V_t$ represents the set of nodes and $E_t$ represents the set of edges occurring at time $t$. For an edge $e$ that appears at least in one snapshot, it is represented as a bitstream $(b_1, b_2, b_3...b_i...b_n)$, where $b_i$ indicates the presence of $e$ at $i$th snapshot– ‘1’ means ‘present’ and ‘0’ means ‘not present’. In this manner, the temporal aspect of the dynamic graph can be captured by bitstreams of all edges.

Figure 5.1 is an example of dynamic graph and its bitstreams. Figure 5.1(a) shows four snapshots of a dynamic graph, and Figure 5.1(b) shows an aggregated graph of the dynamic graph – every edge in the aggregated graph appears at least in one snapshot of the dynamic graph, and beside each edge is its bitstream.
Notation: in the following, we will use $B_e$ to represent a bitstream of edge $e$, and use $B_{e,i,j}$ to denote a sub-bitstream of edge $e$, which indicates $e$’s presence from snapshot $i$ to snapshot $j$.

5.3 Periodicity Detection

Many real world datasets exhibit periodicity. In this section, we focus on detecting major periods in a dynamic graph, and then use the detected periods to further detect periodic communities.

5.3.1 Detecting Major Periods

We use two methods to identify periodicity in a single bitstream, one is Discrete Fourier Transform (DFT), the other is Autocorrelation. Then we aggregate the periodicities of all bitstreams, and the aggregated result reveals the major periods in the dynamic graph. Some previous work has applied the similar idea to detect periodicities in a time series dataset [55].
**Discrete Fourier Transform**: Discrete Fourier transform (DFT) is a popular method in digital signal processing. It decomposes a signal (time domain) into its constituent frequency components (frequency domain), each component with an magnitude and phase. The magnitude of each frequency component indicates how much the signal is consisted of that frequency. The formula of DFT is provided in Equation 5.1, where $x$ is the time domain signal with $N$ samples, $x(0), x(1), x(2)...x(N-1)$, and $X$ is the frequency domain signal also with $N$ samples, $X(0), X(1), X(2)...X(N-1)$. Each $X(k)$ is a complex number, which contains the magnitude and phase of the corresponding frequency $1/k$. $X(0)$ is the DC component of the frequency domain signal, which is meaningless for periodicity mining, so we ignore it.

$$X(k) = \sum_{n=0}^{N-1} x(n)e^{-j2\pi kn/N}, \; k = 0, 1, ..., N - 1$$

(5.1)

In our case, we first apply Discrete Fourier transform (DFT) to each bitstream, and get the magnitude for each frequency component. Then we sum the magnitude of each frequency component across all the bitstreams, and get a cumulative DFT plot. Since period is the reciprocal of frequency, we convert the frequency in the cumulative DFT plot to periods for easier interpretation. The high peaks in the cumulative DFT plot usually correspond to major periods in this dynamic graph.

In real world, most signals are non-periodic, but DFT assumes that the signal is periodic. This assumption may lead to some errors in the result, which is called “spectral leakage” [50]. To reduce the “spectral leakage” error, one common way is to multiply the original signal by a window function. In our experiments we apply the Hanning window function [38] to each bitstream, and other window functions have similar results.
The naive way to compute Discrete Fourier transform (DFT) has complexity \(O(N^2)\), and a more efficient Fast Fourier transform (FFT) algorithm will compute DFT with complexity \(O(N \log N)\).

**Autocorrelation**: Autocorrelation is another method to detect periodicity in a signal, and it is effective when the signal is noisy. For a discrete signal \(x\) (e.g. a bitstream), the autocorrelation \(R\) at lag \(k\) is:

\[
R(k) = \sum_{n=0}^{N-1} x(n)x(n-k), k = 0, 1, 2...N - 1
\]

(5.2)

For a bitstream, what \(R(k)\) essentially computes is, it counts how many ‘1’s reappear after \(k\) snapshots. Intuitively, the larger \(R(k)\) is, the more ‘1’s reappear after \(k\) snapshots, the more likely period \(k\) is a major period of the bitstream. Like cumulative DFT plot, we generate a cumulative Autocorrelation plot by summing each \(R(k)\) across all the bitstreams.

The naive way to compute Autocorrelation at all lags is of complexity \(O(N^2)\), where \(N\) is the length of the bitstream \(x\). An efficient way to compute Autocorrelation is by using Fast Fourier transform (FFT):

\[
R = FFT^{-1}(FFT(x') \cdot FFT(x))
\]

(5.3)

Here \(x'\) is the reverse of bitstream \(x\), \(FFT^{-1}\) is the reverse FFT, and \(R(k)\) is the Circular Autocorrelation at lag \(k\). There is one little problem with Circular Autocorrelation, when \((n - k) < 0\), Circular Autocorrelation will take \(x(n - k)\) as \(x((n - k)modN)\), while in our case we need \(x(n - k)\) to be 0 if \((n - k) < 0\). This problem can be solved by a common method called “zero padding” [47]: that is appending \(N - 1\) bits of ‘0’s to the end of the bitstream \(x\), and then applying Equation 5.3 to \(x\). We only need the first \(N\) elements of \(R\).
computed by Equation 5.3, and they are exactly the same as the N elements of R computed by Equation 5.2.

**Comparison**: One weakness of Discrete Fourier Transform (DFT) is, DFT will miss some “long” periods. The output frequency of DFT will be 1/N, 2/N, 3/N, ... (N-1)/N, which correspond to periods N, N/2, N/3... N/(N-1), it will miss all periods between N and N/2. Autocorrelation will include all the lags ranging from 1 to N-1, so it will overcome this problem.

One problem with Autocorrelation is, for a periodic signal with period k, the values of R(2k), R(3k)... will be close to the value of R(k), and it might be confusing whether the period is k, or 2k, 3k and so on. DFT is less affected by this problem, so we suggest to couple DFT with Autocorrelation to identify the major periods. We will illustrate this in our experiments.

### 5.3.2 Detecting Periodic Communities

After revealing the major periods in the dynamic graph, we could further analyze the graph to detect periodic communities. In our work, a periodic community is defined as a community that appears periodically at snapshots \( s_0, s_0 + p, s_0 + 2 \times p... s_0 + (k - 1) \times p \), here \( p \) is called the *period* and \( k \) is called the *length* of the periodic community.

We use Algorithm 16 to detect periodic communities. In line 1, based on the major periods detected, we pick a major period \( p \) to our interest and set the minimum length \( k \) of the periodic community.

In the following steps we use a sliding window to identify periodic communities. In our algorithm, a window \( W^k_p(s_0) \) is defined as a set of periodic snapshots \( \{ s_0, s_0 + p, s_0 + 2 \times p... s_0 + (k - 1) \times p \} \). For window \( W^k_p(s_0) \), its period is \( p \), length is \( k \), and \( W^k_p(s_0).start = s_0 \).
In line 3 we initialize the sliding window. In step 4, Extract-PeriodicEdges(W) checks each edge in the dynamic graph, if it appears in all the snapshots in window W, we add the edge to set \( E_{pre} \). In step 5, we apply a graph clustering algorithm to the subgraph induced by edges in \( E_{pre} \), and detect a set of communities \( C_{pre} \). In step 6, PostCheckCommunities(W, C) (details in Algorithm 17) will post-check all the communities in C: for community \( c \in C \), if a portion (above a threshold \( \alpha \)) of its edges appear together at a snapshot \( s' \) that \( W.start < s' < W.end \) but \( s' \notin W \), then the community is considered to be non-periodic, and will be removed from C.

In line 7-18, we move the sliding window forward by p snapshots in each iteration, until reaching the last snapshot. In each iteration we detect communities in the current window. In line 8 we set the current window. In line 9, we select the edges (\( E_{cur} \)) that appear at all snapshots in the current window. In line 10, we detect communities in current window by updating the communities in previous window, this is through a dynamic clustering algorithm, and the benefit is it will maintain correspondence between communities in two windows. The input to the dynamic clustering algorithm is the previous graph (\( E_{pre} \)), previous cluster results (\( C_{pre} \)) and the current graph (\( E_{cur} \)). In line 11, we need to post check the communities detected in the current window. In step 12-16, we compare every community detected in the previous window to every community in the current window. If a community \( c \) in the previous window is matched with (similar to) a community \( c' \) in the current window, then we merge \( c \) with \( c' \) and the merged community still stays in \( C_{cur} \). Otherwise if there is no community in current window that matches community \( c \), it indicates the community \( c \) does not continue to exist in current window, so we output it.

In the following we give an informal proof of the correctness of Algorithm 16. Assume there is a periodic community appears at snapshots \( s_i, s_i + p, s_i + 2 * p \ldots s_i + (j - 1) * p \)
where \( j \geq k \). When the sliding window in Algorithm 16 moves to window \( W^k_p(s_i) \), the community will be detected. As the sliding window moves forward, the periodic community will be detected in each of the following windows until the sliding window reaches snapshot \( s_i + (j - 1) \cdot p \). Thus such a periodic community and the period in which it exists can be discovered. If the community not only appears at periodic snapshots \( s_i, s_i + p, \ldots s_i + j' \cdot p, s_i + (j' + 1) \cdot p \ldots s_i + (j - 1) \cdot p \), but also appears at snapshot between \( s_i + j' \cdot p \) and \( s_i + (j' + 1) \cdot p \). When the sliding window moves to window \( W^k_p(s_i + j' \cdot p) \), the community will be recognized as non-periodic by the post check method. So Algorithm 16 will not generate false positive results.

**Algorithm 16** Detect the Periodic Communities

1. Pick the period \( p \) and the minimum length \( k \) of the periodic communities;
2. for \( \{s = 0; s < p; s + +\} \) do
3. \( W^k_p(s) = \{s, s + p, s + 2 \cdot p, \ldots s + (k - 1) \cdot p\}; \)
4. \( E_{pre} = \text{ExtractPeriodicEdges}(W^k_p(s)); \)
5. \( C_{pre} = \text{Clustering}(E_{pre}); \)
6. \( PostCheckCommunities(W^k_p(s), C_{pre}); \)
7. for \( \{s_{new} = s + p; s_{new} + (k - 1) \cdot p < N; s_{new} + = p\} \) do
8. \( W^k_p(s_{new}) = \{s_{new}, s_{new} + p, \ldots s_{new} + (k - 1) \cdot p\}; \)
9. \( E_{cur} = \text{ExtractPeriodicEdges}(W^k_p(s_{new})); \)
10. \( C_{cur} = \text{UpdateClustering}(E_{pre}, E_{cur}, C_{pre}); \)
11. \( PostCheckCommunities(W^k_p(s_{new}), C_{cur}); \)
12. for each community \( c \) in \( C_{pre} \) do
13. if there is a community \( c' \) in \( C_{cur} \) that \( |V_c \cap V_{c'}|/|V_c \cup V_{c'}| > \beta \) then
14. Merge \( c \) to \( c' \);
15. else
16. Output community \( c \).
17. \( E_{pre} = E_{cur}; \)
18. \( C_{pre} = C_{cur}; \)
19. Output communities in \( C_{pre}; \)
Algorithm 17 PostCheckCommunities(W, C)
1: for each community \( c(V_c, E_c) \) in set \( C \) do
2:   for each snapshot \( s \) that \( W.start < s < W.end \) and \( s \notin W \) do
3:     \( E^s_c = \{ e | e \in E_c \text{ and } e \text{ appears at snapshot } s \} \);
4:     if \( |E^s_c|/|E_c| > \alpha \) then
5:       Remove community \( c \) from set \( C \);

5.4 Detecting Topologically and Temporally Correlated Edges

In this section, we propose two methods to detect edges that are both topologically and temporally correlated in a dynamic graph. These correlated edges usually comprise a cluster that occurs often in a dynamic graph. For example, in a dynamic collaboration network, the subgraph might indicate a group of people that often work together.

The first method is a visualization method, it will highlight clusters of topologically and temporally correlated edges. However, it cannot guarantee “density” of the cluster, and there might be noisy nodes within a cluster. Then we propose a second method to detect “dense block” within a cluster. A “dense block” indicates a dense subgraph that appears frequently within a period. It is an extension of the “dense subgraph” idea from static graphs, and captures “density” in time domain.

5.4.1 Terrain Visualization Method

The basic idea of our visualization method is: we build a weighted graph in which every node corresponds to an edge in the dynamic graph. If two edges \( e \) and \( e' \) in the dynamic graph are topologically correlated, their correspondent nodes \( n \) and \( n' \) will be connected with a weighted edge \( e_{n,n'} \), and the weight of \( e_{n,n'} \) indicates how much \( e \) and \( e' \) are temporally correlated. Thus the weighted graph encodes topological and temporal correlation among edges in the dynamic graph. Finally we visualize the weighted graph.
using terrain visualization, and the topologically and temporally correlated edges will be highlighted as peaks in the terrain.

The details of the idea are as follows:

**Topological Correlation**: in our work, two edges in the dynamic graph are topologically correlated if they share a triangle in the aggregated graph of the dynamic graph. This is based on the observations from social science: if edge \((a, b)\) and \((a, c)\) are related, there might be a relation between \((b, c)\).

**Temporal Correlation**: every edge in the dynamic graph is associated with a bitstream, and we compute the temporal correlation between two edges as the Jaccard similarity between their associated bitstreams. Intuitively, if two edge co-occur in many snapshots, their temporal correlation will be high.

In Figure 5.2, the example shows how we convert a dynamic graph (Figure 5.2(a)) into a weighted graph (Figure 5.2(b)). Every node \(XY\) in Figure 5.2(b) corresponds to an edge \((X, Y)\) in Figure 5.2(a). For example, node \(af\) and \(ef\) in Figure 5.2(b) correspond to edge \((a,f)\) and edge \((e,f)\) respectively, and the weight of edge \((af, ef)\) is the Jaccard similarity of bitstreams associated with edge \((a,f)\) and edge \((e,f)\), which is 4.

**Terrain Visualization**: In this section we apply the terrain visualization method to visualize the weighted graph converted by dynamic graph. Intuitively, the peaks/valley in the terrain will represent areas in the weighted graph which have relatively high/low edge weights.

Figure 5.3(a) is a terrain visualization of the weighted graph in Figure 5.2(b). To illustrate the properties of terrain visualization, first we give the definition of *maximal \(\alpha\)-edge connected component* as follows:
Definition 15. In a weighted graph $G(V, E)$ where each edge is associated with a weight, a connected component $C(V_C, E_C)$ of $G(V, E)$ is a maximal $\alpha$-edge connected component if it satisfies following conditions:

1. for every edge $e \in E_C$, $e.weight \geq \alpha$.
2. for any edge $e \in E_C$, if edge $e'$ shares a common vertex with $e$ and $e' \notin E_C$, then $e'.weight < \alpha$.
3. for any edge $e(v_1, v_2) \in E_C$, we have $v_1 \in V_C$ and $v_2 \in V_C$.

In Figure 5.2(b), the connected components C1/C2/C3 (in the green/red/brown circle) are maximal $\alpha$-edge connected components where $\alpha = 4, 5, 4$ respectively. We are usually interested in maximal $\alpha$-edge connected components with high $\alpha$ value, because they usually represent clusters of edges in the dynamic graph that are highly temporally correlated. For example, in Figure 5.2(b), $C_2$ in the red circle (a maximal 4-edge connected
Figure 5.3: An Example of Terrain Visualization

component) represents a cluster of edges (b, c), (b, d), (b, g), (c, d), (c, g), (d, g), (d, h) in Figure 5.2(a), and all the edges co-occur in at least 4 snapshots.

In the terrain visualization of the weighted graph, we can easily get all the maximal $\alpha$-edge connected components with a specific $\alpha$. We just need to use a plane with $height = \alpha$ to cut the terrain, and every peak above the plane corresponds to a maximal $\alpha$-edge connected component (one to one correspondence). For example, in Figure 5.3(a), we first use plane whose $height = 5$ to cut the terrain, there is only one peak (peak 1 in Figure 5.3(b)) above the plane, and peak 1 corresponds to $C_1$ in Figure 5.2(b). Then we use a plane with $height = 4$ to cut the terrain, and there are two peaks (peak 2 and peak 3 in Figure 5.3(b)) above the plane, and they correspond to $C_2$ and $C_3$ in Figure 5.2(b).

In summary, to visually identify topologically and temporally correlated edges, we first convert the dynamic graph to a weighted graph, then generate a terrain visualization of the weighted graph, and use a plane with $height = \alpha$ to cut the terrain. Every peak above the terrain will correspond to a maximal $\alpha$-edge connected component in the weighted graph,
and thus represent a cluster of edges in the dynamic graph that are topologically connected and temporally correlated. The higher $\alpha$ value we use, the more temporally correlated edges we can get.

There are some other properties of the peaks:

1. The thickness of the peak represents the number of edges in the correspondent maximal $\alpha$-edge connected component. For example in Figure 5.3, peak 2 corresponds to $C_2$ and it contains 13 edges while peak 3 corresponds to $C_3$ and it contains 1 edge. Peak 2 is much thicker than peak 3 since it contains many more edges.

2. If the two peaks are not connected in the terrain, their correspondent maximal $\alpha$-edge connected components are not connected. For example in Figure 5.3, peak 2 and peak 3 are not connected in the terrain, and in Figure 5.2(b) $C_2$ and $C_3$ are not connected either.

3. If one peak is contained in another peak, then the correspondent maximal $\alpha$-edge connected component of the former peak is contained in the correspondent maximal $\alpha$-edge connected component of the later peak. For example in Figure 5.3, peak 1 is contained in peak 2, and in Figure 5.2(b), $C_1$ in contained in $C_2$.

5.4.2 Detecting Densest Block within a Cluster

The terrain visualization method could detect clusters of topologically and temporally correlated edges in a dynamic graph. However, the subgraph comprised by such a cluster of correlated edges may contain vertices that are “loosely” connected to the cluster. For example, the subgraph in Figure 5.4(a) is a cluster of correlated edges detected by the terrain visualization (corresponds to peak 2 in Figure 5.3), and inside the subgraph nodes
b, c, d, g are connected to each other while node h is only connected to d, so h is loosely connected to the cluster. These “loosely” connected vertices sometimes are outliers to the cluster, and make the cluster less meaningful and hard to interpret.

![Diagram of a cluster of correlated edges, a dense block, and a weighted graph](image)

**Figure 5.4: An Example of Dense Block**

In this section, we develop a method to detect a dense subgraph in a cluster of correlated edges. In a dense subgraph every node is densely connected with other nodes so there is usually no outliers. Moreover, different from dense subgraph defined on static graph, our method will also identify a period in which the dense subgraph frequently occur. So the nodes in the dense subgraph that are not only *topologically coherent* but also *temporally coherent*. For example, in Figure 5.4(a), our method will identify the dense subgraph comprised by red edges, and find the period in which the dense subgraph occurs frequently is snapshot 1 to snapshot 4. If we use bitstreams to represent the dense subgraph in Figure 5.4(a) from snapshot 1 to snapshot 4, it is a block filled with many ’1’ (Figure 5.4(b)), and we call such a pattern “dense block”. Intuitively, the “dense block” indicates a dense subgraph that frequently occur during a certain period p, and it has following properties:
1. Every edge in the subgraph is topologically correlated with many other edges in the subgraph.

2. During the period p, most edges in the subgraph co-occur many times.

3. Every edge in the subgraph occurs frequently in the period p.

In the following we give a more formal definition of the problem.

The definition of a block is:

**Definition 16.** 

$$ BLOCK_{E_b,i,j} = \{ B_{e,i,j} \mid e \in E_b \}, $$

Where $E_b$ is a set of edges, and $B_{e,i,j}$ represents the sub-bitstream of edge $e$ starting from snapshot $i$ to snapshot $j$.

Then we define two types of density of a block, the first is correlation density, denoted as $CorrDen(BLOCK_{E_b,i,j})$.

$$ CorrDen(BLOCK_{E_b,i,j}) = \frac{\sum_{e,e' \in E_b} Corr(B_{e,i,j}, B_{e',i,j})}{|E_b|} $$

(5.4)

In Equation 5.4, $Corr(B_{e,i,j}, B_{e',i,j})$ is to compute the correlation value between two edges $e$ and $e'$. If $e$ and $e'$ are not topologically correlated (i.e. share a triangle), $Corr(B_{e,i,j}, B_{e',i,j})$ is 0. Otherwise, $Corr(B_{e,i,j}, B_{e',i,j})$ is the Jaccard similarity of sub-bitstreams $B_{e,i,j}$ and $B_{e',i,j}$. Correlation density actually captures correlation among edges in the block.

The second type of block density is frequency density, denoted as $FreqDen(BLOCK_{E_b,i,j})$.

$$ FreqDen(BLOCK_{E_b,i,j}) = \frac{\sum_{e \in E_b} Freq(B_{e,i,j})}{|E_b| \ast (j - i + 1)} $$

(5.5)

In Equation 5.5, $Freq(B_{e,i,j})$ is to compute how many times edge $e$ appears from snapshot $i$ to snapshot $j$. So $Freq(B_{e,i,j})/(j - i + 1)$ is the frequency of edge $e$’s appearance from
snapshot i to snapshot j, and the frequency density actually computes the average frequency of all the edges’ appearance.

Finally we define the overall density of a block as

\[
Density(BLOCK_{E_b,i,j}) = CorrDen(BLOCK_{E_b,i,j}) * 
\]

\[
FreqDen(BLOCK_{E_b,i,j})
\]

(5.6)

Our problem is to find the densest block among a cluster of edges, it is defined as:

**Problem 1.** Given a set of edges E and every edge’s bitstream from snapshot 1 to snapshot N, find a block from the set of blocks \{BLOCK_{E_b,i,j} | E_b \subseteq E, |E_b| \geq k and 1 \leq i \leq j \leq N\} that has the maximum overall density (Equation 5.6).

Here we specify a minimum number of k edges within the densest block, because sometimes the block is meaningless if there are too few edges within it. To solve Problem 1, our idea is to iterate over all possible periods, and detect the densest block during each period. Finally we return the densest block among all densest blocks detected.

In the following we focus on detecting the densest block in a cluster of edges E during a period from snapshot i to snapshot j. We first convert the cluster of edges into a weighted graph \(G_{i,j}\), where every node \(n\) in the weighted graph corresponds to an edge \(e\), and \(n.weight = \text{Freq}(B_{e,i,j})\). If two edges \(e\) and \(e'\) in E share a triangle with a third edge in E (topologically correlated), their correspondent nodes \(n\) and \(n'\) will be connected with a weighted edge \(e(n, n'),\) and \(e(n, n').weight = Corr(B_{e,i,j}, B_{e',i,j}).\) For example, Figure 5.4(c) is the weighted graph converted by cluster of edges in Figure 5.4(a) from snapshot 1 to snapshot 4. For any subgraph \(G'\) in the weighted graph, we define the following density

\[
Density(G') = \frac{\sum_{e \in G'} e.weight}{|V|} * \frac{\sum_{v \in G'} v.weight}{|V| * (j - i + 1)}
\]

(5.7)
We can easily see that there is a one-to-one correspondence between $BLOCK_{E_b,i,j}$ where $E_b \subseteq E$ and subgraph in the weighted graph $G_{i,j}$, and the density of the block (Equation 5.6) is equal to the density of its correspondent subgraph (Equation 5.7). If we want to find the densest block with at least k edges, it is equivalent to find the densest subgraph with at least k vertices in the weighted graph.

In the following we prove it is NP hard problem.

Claim 6. In a weighted graph where each edge and vertex is associated with a weight, find the densest subgraph with at least k vertices (density is defined in Equation 5.7) is a NP hard problem.

Proof. The densest-at-least-k-subgraph problem is proved to be NP-hard in [46]. The density of subgraph $S(V_s, E_s)$ is defined as $\frac{|E_s|}{|V_s|}$ in [46], In our problem we use a different definition of density (Equation 5.7). If we set every edge weight to be 1, and every vertex weight to be $(j - i + 1)$, the density defined in Equation 5.7 is equal to the density defined in [46], then the densest-at-least-k-subgraph problem in [46] reduces to be our problem. So our problem (Problem 1) is a NP hard problem. □

We propose a simple approximate method (Algorithm 18) to detect the densest subgraph with at least k vertices in a weighted graph. It is a greedy algorithm – in every iteration we remove the node that will make the remaining graph has the maximum density, and store the remaining graph and its density. Finally we return the remaining graph that has the maximum density.

Here we give an intuitive explanation about the effectiveness of our method. For the densest block detected by our algorithm, its high correlation density will guarantee that every edge will be topologically correlated with many other edges, and co-occur with many
Algorithm 18: Detect the densest subgraph in a weighted graph

Require: A weighted Graph $G$ and parameter $k$.
Ensure: The densest subgraph in $G$.

1: $S_0 = G$;
2: $d_0 = \text{density}(G)$;
3: $i = 1$;
4: while $G$ contains more than $k$ vertices do
5: find vertex $v$ from the current vertices in $G$ that subgraph $G - \{v\}$ has the maximum density;
6: $S_i = G - \{v\}$;
7: $d_i = \text{density}(S_i)$;
8: $i = i + 1$;
9: $G = G - \{v\}$;
10: Find the subgraph $S_i$ with the maximum density $d_i$, return $S_i$;

other edges during the period. Also its high frequency density will guarantee that every edge appears frequently during the period. So the densest block detected by Algorithm 18 will represent a dense subgraph that appears frequently in a certain period.

5.5 Experiments

5.5.1 Datasets

The datasets used in our experiments are introduced below.

**Enron:** The Enron dataset (http://www.cs.cmu.edu/enron/) contains emails sent by Enron corporation accounts. Each node represents an email account and each edge represents an email. The dataset has 9652 vertices and 17317 edges, and lasts 1352 days. The interval between consecutive snapshots is 1 day.

**Facebook:** The Facebook dataset contains the wall posts from the Facebook New Orleans users. Each node is a user, and an edge means one user posted on the other user’s wall. The dataset has 63891 vertices and 193494 edges, and lasts 1592 days. The interval between consecutive snapshots is 1 day.
DBLP: The DBLP dataset is a coauthor network, where each node is an author and an edge indicates coauthorship between authors. It contains 1347526 nodes and 2013724 edges, and lasts from 1938 to 2006. The interval between consecutive snapshots is 1 year.

Congress: The Congress dataset is a cosponsorship network. Each node is a congressman (Senator or House Representative), and an edge indicates that one congressman cosponsored a bill which is proposed by another congressman. It contains 1718 nodes and 346625 edges, and lasts from 1971 to 2004. The interval between consecutive snapshots is 1 year.

5.5.2 Periodicity Mining

The cumulative DFT plots and cumulative Autocorrelation plots of the Enron, DBLP, Facebook, and Congress datasets are presented in Figure 5.5, Figure 5.6, Figure 5.7 and Figure 5.8. Please note that, when applying DFT to bitstreams, the maximum frequency that can be represented by DFT is the Nyquist frequency, which is $\frac{1}{2\Delta}$ ($\Delta$ is the interval between consecutive snapshots). So the minimum period that we can detect for each dataset is, 2 days (Enron), 2 days (Facebook), 2 years (DBLP), 2 years (Congress).

In Figure 5.5(a), we can see two major peaks in the cumulative DFT plot of Enron dataset, the highest peak corresponds to $period = 7$, which indicates that many bitstreams have 7 days as major periods. When referring to the cumulative Autocorrelation plot in Figure 5.5(b), we find that the peaks occur at 7, 14, 21... which are all multiples of 7, so it verifies that the major period of Enron dataset is $period = 7$ days.

Figure 5.7(a) is the cumulative DFT plot of Facebook dataset, we can see one peak occurs at $period = 7$, and in its cumulative Autocorrelation plot (Figure 5.7(b)), we can see some peaks occur at 7, 14, 21... which indicates that $period = 7$ is a major period in
Facebook dataset. Another peak in cumulative Autocorrelation plot (Figure 5.7(b)) occurs at 365, which indicates that $\text{period} = 365$ is also major period in Facebook dataset, this period is missed in cumulative DFT plot, because in the output of DFT, it does not have a frequency corresponds to $\text{period} = 365$.

In Figure 5.6(a) and Figure 5.6(b), we do not see any peaks, so there are no significant periods in DBLP dataset. Similarly we do not see peaks in Congress dataset (Figure 5.8(a) and Figure 5.8(b)).

After identifying the major periods in these datasets, we apply Algorithm16 to detected periodic communities in Enron dataset. Here we listed a few periodic communities in Enron dataset in Table5.1. Every community is formed in the same way: someone first sent an email to other members, and other members responded to the email. So we list the subject the first email of each community at each snapshot. People in Community 1 play a game together every Monday. In community 2, people bet on the NCAA tournament games, and they updated the scores every week. In community 3, people attended a meeting every week.

<table>
<thead>
<tr>
<th>Communities</th>
<th>Times</th>
<th>Email Subjects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Community 1</td>
<td>Mon, 24 Apr 2000</td>
<td>Game Tonight</td>
</tr>
<tr>
<td></td>
<td>Mon, 1 May 2000</td>
<td>GAME TONIGHT @ 7:45</td>
</tr>
<tr>
<td></td>
<td>Mon, 8 May 2000</td>
<td>GAME TONIGHT @ 8:45</td>
</tr>
<tr>
<td></td>
<td>Mon, 15 May 2000</td>
<td>GAME TONIGHT @ 9:45</td>
</tr>
<tr>
<td></td>
<td>Mon, 22 May 2000</td>
<td>GAME TONIGHT @ 9:45</td>
</tr>
<tr>
<td>Community 2</td>
<td>Mon, 12 Mar 2001</td>
<td>NCAA tournament</td>
</tr>
<tr>
<td></td>
<td>Mon, 19 Mar 2001</td>
<td>Sweet sixteen scores</td>
</tr>
<tr>
<td></td>
<td>Mon, 26 Mar 2001</td>
<td>Pool Update</td>
</tr>
<tr>
<td>Community 3</td>
<td>Mon, 30 Apr 2001</td>
<td>5/3 EWS Budget Mtg @ 11:00 a.m. in EB30C2</td>
</tr>
<tr>
<td></td>
<td>Mon, 7 May 2001</td>
<td>EWS Budget Mtg @ 11:00 a.m. in EB32C2</td>
</tr>
<tr>
<td></td>
<td>Mon, 14 May 2001</td>
<td>EWS Budget Meeting</td>
</tr>
</tbody>
</table>
Figure 5.5: Cumulative DFT/Autocorrelation Plots for Enron
Figure 5.6: Cumulative DFT/Autocorrelation Plots for DBLP
Figure 5.7: Cumulative DFT/Autocorrelation Plots for Facebook
Figure 5.8: Cumulative DFT/Autocorrelation Plots for Congress
5.5.3 Detecting Topologically and Temporally Correlated Edges

**Congress dataset:** In the congress dataset, since some congressmen work as both senators and house representatives, the aggregated congress cosponsorship network is a connected graph. The terrain visualization of congress cosponsorship network is in Figure 5.9(a). We first use two high planes to cut the terrain to get two peaks (peak 1 and peak 2), and they represent two clusters of edges that are highly temporally correlated. Peak 1 represents the cluster of edges in Figure 5.9(b), and the bitstream of each edge is in Figure 5.9(d). These edges co-occur in most snapshots from year 1971 to 2004, and they comprise a cluster which indicates a group of senators that work together for many years. These senators are actually the longest serving members of US senate, and most of them in Figure 5.9(b) (except Ted Stevens) are from Democratic party. Peak 2 represents the cluster of edges in Figure 5.9(c), and the associated bitstream are is in Figure 5.9(e), these edges also co-occur in most snapshots from year 1971 to 2004. The subgraph indicates a group of longest-serving house representatives that work together for many years, and all of them are from Democratic party.

Then we use a lower plane to cut the terrain, and generate two other peaks (peak 3 and peak 4), which indicate two larger clusters of temporally correlated edges (Figure 5.9(f) and Figure 5.9(g)). These edges are less temporally correlated than cluster 1 and cluster 2, and contain some loosely connected vertices. So we detect the densest block in each of them. The edges and subbitstreams of the densest block in cluster 4 are shown in Figure 5.9(i) and Figure 5.9(j), and the block starts from year 1975 to year 2004. We could see that during the period (1975-2004), every edge of the dense block appears frequently, and the dense subgraph appears in many snapshots. Although in some snapshots a few edges in the dense subgraph were missing, most of its edges still appear together. This indicates a
Figure 5.9: Congress Dataset
strong topological and temporal coherence between the nodes in the dense subgraph. The dense subgraph in Figure 5.9(i) is actually comprised by a few senior house representatives, and all of them are from Democratic party. It indicates a close relationship among them from year 1975 to year 2004.

In Figure 5.9(h), we show the edges of the densest block in peak 3, and the block starts from year 1979 to year 2004. Due to limitation of space, we do not show the subbitstreams of the block. Similarly to the previous dense block, every edge appears most of the snapshots during the period, and indicates a close relationship among the nodes. The subgraph is comprised by a few senior senators (most of them are from Democratic party), and it indicates a close relationship among them from year 1979 to year 2004.

**DBLP dataset:** The terrain visualization of DBLP network is in Figure 5.10(a). We first use two planes to cut the terrain, and generate two peaks which represent two clusters of highly temporally correlated edges. Peak 1 represents the cluster of edges in Figure 5.10(b), and their bitstreams are in Figure 5.10(d). In Figure 5.10(d) we only show the bitstreams from year 1986 to 2006, because all edges in the cluster do not appear before 1986. We can see that all the edges co-occur in most years from year 1986 to 2006. They are actually a group of researchers working on Computational Geometry, and Micha Sharir is the center of the cluster, he worked with other four researchers for many years.

Peak 2 represents the cluster of edges in Figure 5.10(c), and their bitstreams are in Figure 5.10(e). In Figure 5.10(e) we only show the bitstreams from year 1990 to 2006, because no edge in the cluster appears before 1990. All the edges in the cluster co-occur in most years from year 1990 to 2006. They are actually a group of researchers working on digital testing, and they published a lot of research papers together for many years.
Figure 5.10: DBLP Dataset
Then we lower down the plane (the lowest black line in Figure 5.10(a)) to cut the terrain, and generate many more peaks. Each peak still represents a cluster of temporally correlated edges, but less temporally correlated than cluster 1 and 2. We detect the densest block in each cluster, then select the two densest blocks with highest overall density, and present their details below.

The edges in the first densest block are in Figure 5.10(f), and all the edges comprise a dense subgraph. Every edge appears in 5 consecutive years from 2002 to 2006 (i.e. the dense subgraph appears every year from 2002 to 2006). Actually the dense block indicates a group of authors working together on “Ensembl genome database” project for 5 years (from 2002 to 2006), and each year they publish one or more papers on the project.

The edges in the second densest block are in Figure 5.10(g), and all the edges comprise a clique (dense subgraph). Every edge appears in 5 consecutive years from 2002 to 2006 (i.e. the dense subgraph appears every year from 2002 to 2006). Actually the dense block indicates a group of authors working together on “Saccharomyces genome database” project for 5 years (from 2002 to 2006), and each year they publish one or more papers on the project.
Chapter 6: Concluding Remarks

Large scale graph visualization is becoming increasingly complicated with the widening gap between the amount of data that has to be visualized, available pixel space and cognitive abilities of humans in-the-loop. To address this challenge, in this thesis we proposed a framework which incorporates the following three components.

In the first component, we define and introduce the notion of a Triangle K-Core, a simple topological motif and demonstrate how to extract such structures efficiently from both static and dynamic graphs. We empirically demonstrate on a range of real-world datasets that this motif can be used as a proxy for probing and visualizing relevant clique-like structure from large dynamic graphs and networks. Finally, we discuss a method to extend the basic definition to support user defined clique template patterns with applications to network visualization, correspondence analysis and event detection on graphs and networks. Based on Triangle K-Cores, we also develop a parallel method to compute approximate cliques from large graphs.

In the second component, we develop a visualization method to analyze scalar graphs. We demonstrate that our visualization method could reveal overall distribution of attribute values over a graph, and highlight interesting areas (dense subgraphs, social communities, etc.). It can also be extended to analyze relationship between multiple attributes. We
evaluate our system on a range of real-world data, and show the effectiveness and scalability of our system.

In the third component, we represent all the snapshots of a dynamic graph using bit-streams. The bitstreams actually encode the temporal information of the edges. We first apply two popular time series analysis methods (Discrete Fourier Transform and Autocorrelation) to bitstreams to identify major periods in a dynamic graph. Based on the detected major periods, we design an algorithm to detect communities that appear periodically in the dynamic graph. We also identify the topologically and temporally correlated edges by using the terrain visualization, and use “dense block” to detect dense subgraphs that frequently occur in a certain period.

6.1 Limitation

As with almost all such efforts our solutions have both implicit and explicit limitations. Here we outline the important ones and subsequently describe possible avenues of future work that can address them.

The first limitation of our work is, we do not consider directionality in the three components of our work. Directionality exists in many graph datasets, such as email networks, citation networks. In our work, we simply ignore the directionality associated with graphs, this might lose useful information in some cases. For example, in a directed graph, one dense subgraph could have most directed edges pointing to a small portion of its nodes, and another dense subgraph could have incoming/outgoing edges uniformly distributed over its nodes. The two subgraphs could carry different meanings, and ignoring directionality would make us unable to distinguish between the two different types of dense subgraphs. One possible solution is, we could design different template patterns (like the
template patterns in Section 2.4) to represent dense subgraphs with different directionali-
ties. Another possible solution is, we could convert a directed graph into a undirected graph
through symmetrization [60], thus different directed subgraph patterns would correspond
to different subgraphs in the converted graph. Then we could apply our methods on the
converted undirected graph.

The second limitation of our work is, we do not handle some special types of graph,
such as hypergraphs – a graph in which an edge can connect any number of vertices. In
some cases, it is more appropriate to model a dataset as hypergraphs than a normal graph.
For example, if we model a co-authorship network as a hypergraph, people working on the
same paper could be linked by one hyperedge. If we model it as a normal graph, we need to
connect every pair of people participating in the paper. Modeling through normal graph is
problematic, because it is hard to tell from the normal graph whether these people work on
one single paper or each pair of them work on a different paper. Our current methods could
not handle the hypergraphs appropriately. One possible solution is, we can abstract the
nodes connected by an hyperedge into a super node, and construct a super graph comprised
by super nodes. Then we could apply our methods on the super graph.

There are also specific limitations for each component of our work:
1. For sparse graphs with few triangles structures, there are no significant dense subgraphs
in the graph, so the CSV density plot employed by our method would be flat and low, and
no interesting formation could be extracted from the density plot.
2. The major concern with our terrain visualization work is, the occlusion problem of 3D
visualization may affect users’ performance. We tried to alleviate the problem through
adding the user interaction features (rotation, zoom in/out etc.). Also, when visualizing
complicated scalar trees, jagged edges might bring noise to the visualization, one solution
is to simplify the part of terrain that is too complicated. The simplification might lose some
information, but users can capture the main information and would not be mislead by noise
effects.

3. For the bitstream representation of dynamic graph, the effectiveness of our methods
is dependent on the sampling rate and discretization of the dataset (i.e. the interval be-
tween consecutive snapshots), an inappropriate sampling rate could make our method fail
to identify the meaningful periods or clusters.

6.2 Future Work

6.2.1 Matrix Representation of Dynamic Graphs

Matrix-based graph visualization has been used in several works[42, 33]. The idea is
illustrated in Figure 6.1, which is a matrix-based representation of a small graph[42]. It
orders vertices along X and Y axis, and puts a white box in cell(a, b) and cell(b, a) if there
is an edge between vertex a and vertex b. The blocks of white boxes indicate communities
in the graph. We plan to use the matrix-based graph visualization for analyzing changes in
dynamic graphs.

Changes in Dynamic Graphs are usually related to the underlying graph topological
structures, especially community structures. Here we characterize three types of changes
related to community structures.

1. The major change in communities: this type of change occurs to most community mem-
bers. For example, the yellow part in Figure 6.1(b) indicates the edges represented by
yellow boxes are removed, this might indicate that those members are leaving the commu-
nity

2. The change in connections between communities. This type of change occurs to two
communities. For example, the green part in Figure 6.1(b) indicates that edges connecting two communities are removed, so the two communities will be less connected.

3. The outlier change in communities. This type of change occurs to very few members of a community while most other members stay stable, such as the red part in Figure 6.1(b).

We plan to use analytic methods first to identify these changes, and then visualize them using different colors in the matrix, so users can clearly which type of changes occur in which community.

There is a question that needs more investigation: the nodes/edges might participate in different types of changes in the old and new graph. For example, the vertices leaving one community (major change) might join other communities (outlier change). How to identify relations between changes in the old and new graph? One solution is draw different matrix representations of the graph before and after the change, and link the changing nodes/edges in the two matrix representations, so we may identify the types of the change and their relations in the old and new graph.

6.2.2 Efficient Storage of Dynamic Graphs Using Bitstreams

We propose an idea to use hierarchical bitstreams to store dynamic graphs, and the hierarchical structure can support efficient query and analytic operations on dynamic graphs. Figure 6.2 illustrates the idea. The lowest level nodes in Figure 6.2 correspond to all the bitstreams of edges. We first divide all the bitstreams into clusters, and then compute a representative bitstream for each cluster. The representative bitstream is in the second level.
The value of each bit in the representative bitstream could be the majority value of the corresponding bit of the bitstreams in the cluster. We can still cluster the second level nodes and get higher level representative node (root node).

Each representative bitstream is associated with a table, recording the differences between it and the bitstreams in its cluster. For example, node ‘g’ is a representative bitstream of cluster \{a, b, c, d\}, and its table is the red box left to ‘g’. The length of each bitstream is of length 3 so the table has 3 entries. The 1st entry contains the nodes that differ from ‘g’ in 1st bit (left-most bit), it is ‘c’. The 2nd entry is empty because all nodes have the same 2nd bit value. The 3rd entry stores ‘b’ and ‘d’ because their 3rd bit are different from the representative bitstream.

In the following we explain the benefits of using this hierarchical representation.

1. This representation reduces storage of all the snapshots. In the example of Figure 6.2, we just need to store the data in the red box. The compression rate is dependent on the similarity among bitstreams in the same cluster.
2. We can easily query a snapshot at a given time point. For example, in Figure 6.2, if we want to query 2nd snapshot, we first start from the root, and record its 2nd bit value. Then we traverse down to second level nodes, and compute the 2nd bit values for them. After that we traverse down to the lowest level, and compute their 2nd bit values. In fact, based on our representation, the computation can be done by only checking the 2nd entry in all red tables, so we only need check three times in this example, the cost is very low. Finally we output all the edges with 2nd bit equaling 1.

3. We can easily update the representation when new snapshots/edges come. When a new snapshot is added, we will add a new entry to the tables, a new bit to the root node bitstream. When a new edges is added, we will find its nearest representative bitstream, and add it to the corresponding cluster. We may need to re-compute the clustering when a large number of snapshots/edges are added.
4. This representation can accelerate detecting temporal patterns in bitstreams. For example, if we want to count the frequency of a pattern "11*" in Figure 6.2, we just need to check the representative nodes, when checking representative node ‘g’ and its associated table, we find that it is of pattern "11*", and only c differs from it in the first 2 bits, then we know that except c, all the bitstreams in its cluster satisfy pattern "11*”. This shows that our representation can accelerate temporal pattern mining.

5. We can also make use of this representation to help detect structural patterns. Figure 6.3 is a structural pattern from paper [14], the number on each edge denotes the relative time point when the edge is established. The task is to mine all the structures of such pattern in all the snapshots. We can make use of our representation to prune the data, and then detect the structural patterns. The pruning idea is, we find that the left-most edge should have temporal pattern “011”, the 3 middle edges (dashed lines) should have temporal pattern “001”, and the right-most edge should have temporal pattern “111”, we prune all the bitstreams that do not satisfy the 3 patterns, and use retained edges to build graphs, and detect the structural pattern.

Figure 6.3: Local Structural Pattern
Bibliography


