# CMGV: A UNIFIED FRAMEWORK FOR COMPLEXITY MANAGEMENT IN GRAPH VISUALIZATION

A THESIS SUBMITTED TO

THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR

THE DEGREE OF

MASTER OF SCIENCE

IN

COMPUTER ENGINEERING

By

Osama Zafar

August 2023

CMGV: a unified framework for complexity management in graph visualization
By Osama Zafar
August 2023

We certify that we have read this thesis and that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Ugur Dogrusoz(Advisor)

Fazlı Can

İsmail Hakkı Toroslu

Approved for the Graduate School of Engineering and Science:

Orhan Arıkan
Director of the Graduate School

# ABSTRACT

## CMGV: A UNIFIED FRAMEWORK FOR COMPLEXITY MANAGEMENT IN GRAPH VISUALIZATION

Osama Zafar
M.S. in Computer Engineering
Advisor: Ugur Dogrusoz
August 2023

In today's era of technological revolution, the sheer volume of data being produced poses a significant challenge for analyzing relational data of such scale, particularly in terms of visual analysis. Graphs provide an effective way of organizing and representing relational data, with nodes representing entities. In contrast, edges representing relationships, a comprehensive and intuitive view of complex large-scale data is created. A well-represented visualization of complex graphs allows users to understand relationships, uncover new insights, and discover hidden patterns. To this end, we introduce a complexity management framework for effectively analyzing large-scale relational data represented as graphs. Existing methods for managing graph complexity work independently and may lead to inconsistencies and confusion consecutively applied. The Complexity Management Graph Visualization framework (CMGV) presents a novel approach integrating commonly used complexity management techniques while ensuring the preservation of the user's mental map through a specialized layout algorithm. The framework introduces an intuitive Graph Complexity Management Model (CMGM) for both graph representation and complexity management.

CMGV supports commonly utilized complexity management tasks, including filtering, hiding, showing, collapsing, and expanding graph elements. Importantly, CMGV is designed to be independent of the rendering method and can be seamlessly integrated with different graph rendering libraries. This is possible through an extension that synchronizes the graph models between the rendering library and CMGM. Our experiments performed on randomly generated graphs verify that CMGV flawlessly performs consecutive graph complexity management operations, leaving the user graph intact, and outperforms existing complexity management solutions in terms of both runtime and generally accepted graph

layout criteria. It is fast enough to be used in interactive applications with small to medium-sized graphs.

# ÖZET

## TÜRKÇE BAŞLIK



Osama Zafar
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Danışmanı: Ugur Dogrusoz
Temmuz 2023


Günümüzün teknolojik devrim çağında, üretilen verilerin hacmi, özellikle görsel analiz açısından, bu tür ölçekteki ilişkisel verileri analiz etmek için önemli bir zorluk teşkil etmektedir. Çizgeler, varlıkları temsil eden düğümlerle ilişkisel verileri düzenlemenin ve temsil etmenin etkili bir yolunu sağlar. Tersine, ilişkileri temsil eden kenarlar, karmaşık büyük ölçekli verilerin kapsamlı ve sezgisel bir görünümünü oluşturur. Karmaşık çizgelerin iyi temsil edilen bir görselleştirmesi, kullanıcıların ilişkileri anlamalarına, yeni içgörüler ortaya çıkarmalarına ve gizli kalıpları keşfetmelerine olanak tanır. Bu amaçla, çizgelerle temsil edilen büyük ölçekli ilişkisel verileri etkili bir şekilde analiz etmek için bir karmaşıklık yönetimi çerçevesi sunuyoruz. Çizge karmaşıklığını yönetmek için mevcut yöntemler bağımsız olarak çalışır ve art arda uygulanan tutarsızlıklara ve karışıklığa yol açabilir. Karmaşıklık Yönetimi Çizge Görselleştirme çerçevesi (CMGV), özel bir yerleştirme algoritması aracılığıyla kullanıcının zihinsel haritasının korunmasını sağlarken yaygın olarak kullanılan karmaşıklık yönetimi tekniklerini entegre eden yeni bir yaklaşım sunar. Çerçeve, hem çizge temsili hem de karmaşıklık yönetimi için sezgisel bir Çizge Karmaşıklık Yönetimi Modeli (CMGM) sunar.

CMGV, çizge öğelerini filtreleme, gizleme, gösterme, daraltma ve genişletme dahil olmak üzere yaygın olarak kullanılan karmaşıklık yönetimi görevlerini destekler. Daha da önemlisi, CMGV, işleme yönteminden bağımsız olacak şekilde tasarlanmıştır ve farklı çizge işleme kütüphaneleriyle sorunsuz bir şekilde entegre edilebilir. Bu, işleme kütüphanesi ve CMGM arasında çizge modellerini senkronize eden bir uzantı aracılığıyla mümkündür. Rastgele oluşturulmuş çizgeler üzerinde gerçekleştirilen deneylerimiz, CMGV'nin kullanıcı çizgesini olduğu gibi bırakarak ardışık çizge karmaşıklığı yönetimi işlemlerini kusursuz bir şekilde gerçekleştirdiğini ve hem çalışma zamanı hem de genel kabul görmüş çizge düzeni kriterleri açısından mevcut karmaşıklık yönetimi çözümlerinden daha iyi performans gösterdiğini doğrulamaktadır. Küçük ve orta boyutlu çizgeler içeren etkileşimli uygulamalarda kullanılabilecek kadar hızlıdır.

*Anahtar sözcükler*: Bilgi görselleştirme, çizge görselleştirme, çizge karmaşıklık yönetimi, görsel analiz, çizge görselleştirme yazılımı.

# Acknowledgement

I would like to express my heartfelt gratitude to Prof. Dr. Uğur Doğrusöz, for his invaluable motivation, guidance, and support throughout my MS journey. Working with him was one of the best decisions I have ever made.

I would like to express my gratitude to Prof. Dr. Fazlı Can and Prof. Dr. İsmail Hakkı Toroslu for graciously agreeing to be members of my thesis jury. My sincere thanks also go to my senior and mentor, Dr. Hasan Balcı, for his invaluable guidance and unwavering support throughout this journey. Additionally, I would like to acknowledge The Scientific and Technological Research Council of Turkey (TÜBİTAK) for their financial support that made this thesis possible.

Special thanks to my valuable friends Hamza Islam, Najeeb Ullah, Badar Munir, Hilal Khan, Talha Ejaz, Haider Raheem, Syed Asad Shah, Muhammad Jafar, Umer Saeed, Faizan Ahmad, Zain Ali, Zubair Abdullah Khan, Shahzaib Shah, and many others, for making this difficult journey easy and enjoyable for me. I would also like to extend my appreciation to all the faculty members, graduate students, and staff in the Office of International Students for their kindness and warm welcome.

Last but not least, I would like to express my deepest gratitude to my parents, Zafar Iqbal Mughal and Khalida Zafar, and my sisters Anosha Zafar, Sawera Zafar, and Aiman Zafar, who have always supported me throughout my life.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Motivation

Today's world is a global village filled with technological marvels from all walks of life. Since the dawn of the World Wide Web, the everyday use of electronic gadgets has grown exponentially. With the increased use of technology in our daily life, the data produced by these devices has also grown, which coined the term "big data" and made data analysis an extremely challenging endeavor. When it comes to big data analysis, it is necessary to convert it into a comprehensive format. It is known that visual data is much easier to explore and understand compared to tabular or numerical data [6]. The field of information visualization plays a vital role here. *Information visualization* is the art of representing data insightfully and visually so that users can easily understand it. Visualizations come in many shapes and forms, such as charts, histograms, heatmaps, and graphs, each suitable for representing a specific data type. Most real-world data contains relational connections among entities, which can be represented and visualized using graphs. When it comes to relational data, graphs provide an effective way of representation. Data elements become nodes, and edges connecting them represent relations, creating a comprehensive yet intuitive view of complex, large-scale data. *Graph visualization* is used in numerous domains, from biological pathways to social networks and architecture diagrams (Figure 1.1 through Figure 1.3).

Relational data can exhibit hierarchical structures, grouping of entities, and classification of data, which happens with most real-world datasets, as in Figure 1.1 and Figure 1.3. *Compound graphs* [7] are highly valuable for representing and managing complex data. With compound graphs, multiple layers of information can be organized and visualized effectively, allowing for a comprehensive understanding of the underlying complexity.

Although graphs are very effective in representing relational data, but when it comes to dealing with large-scale graphs, the complexity can quickly become overwhelming. A graph complexity management model is vital in organizing and visualizing different relational graph data because it helps make complex graphs easier to understand. The management model simplifies the visualization by reducing clutter and highlighting the most important information. This improves clarity, making identifying patterns and connections within the data easier. By managing complexity, the model enables scalability, allowing for the visualization of massive graphs. It also supports exploration and navigation, helping users delve into the data at different levels of detail.

Another important requirement in the graph visualization of relational data is the arrangement of nodes and edges (i.e., the graph's layout). A good layout is essential since a poor one may mislead or confuse the user, and manual layout adjustments can take up to 25% of a typical user's time [8]. Effectiveness of layout is generally decided by some commonly accepted metrics such as the number of edge crossings, average edge length, and symmetry maximization. One such metric is maintaining the user's mental map after performing complexity management operations. This is an important metric to consider as it highly affects the user's graph exploration experience.

There have been numerous works conducted on graph complexity management [9]; however, the studies have been done independently, focusing on unique

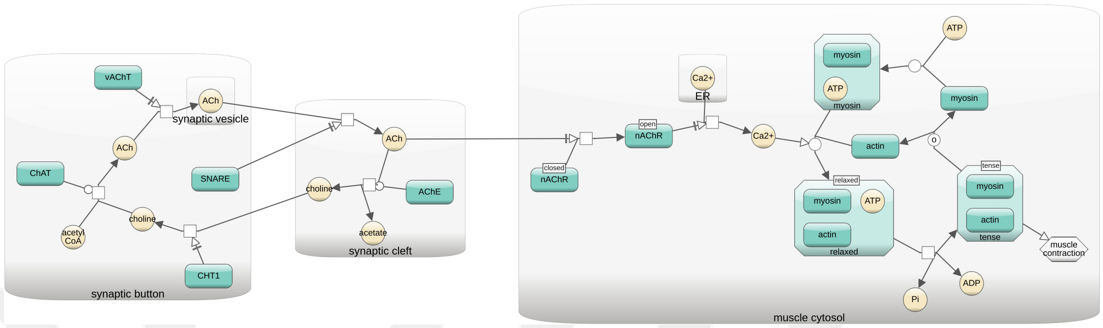



Figure 1.1: A sample biological pathway - neuronal muscle signaling [1]

Figure 1.2: A sample social network of Twitter followers [2]

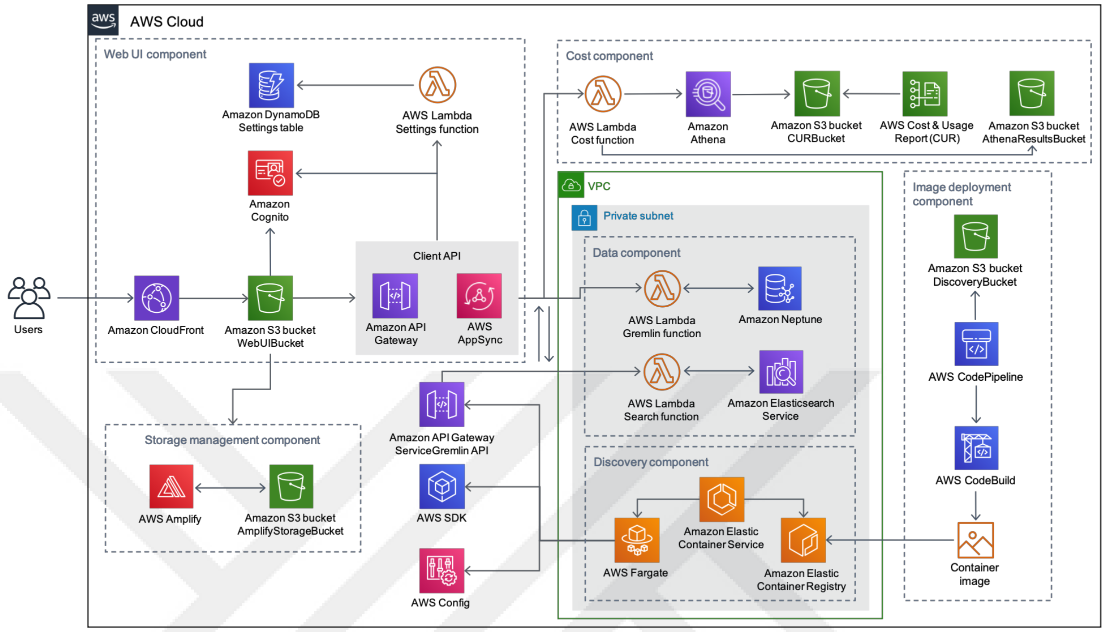



Figure 1.3: Architecture diagram of AWS Perspective, a visualization tool for cloud workloads. [3]

complexity management problems. Complexity management at different levels of graphs, aimed at reducing the size of graph visualization and making it comprehensive, is facilitated by expand/collapse operations of compound graphs [9]. Another extensively studied problem involves removing nodes and edges from a graph based on user-defined criteria for graph exploration and analysis. This task is accomplished through filter/unfilter operations and hide/show operations [9]. While these complexity management techniques are well-curated, they are developed independently and do not consider each other's actions during graph processing, leading to inconsistent results. Moreover, these techniques are also individually developed for rendering libraries without any standard algorithm in mind, which can result in varying outcomes for the same operation on different rendering libraries.

Also, numerous works have been done on simple and compound graph layouts [10, 11]. However, the existing approaches suffer from certain weaknesses, such as inadequate support for compound structures, particularly in maintaining the user's mental map when dealing with the expansion operation for compound nodes. This aspect is crucial for most real-life applications.

In this thesis, we present a new framework for complexity management in graph visualization called CMGV to overcome these deficiencies of existing management techniques and layout algorithms.

## 1.2 Contribution

CMGV is a novel framework designed to tackle the complexities of analyzing large-scale relational data in graph form. By using the new complexity management graph model (CMGM), CMGV unifies various complexity management techniques, enables seamless collaboration between these techniques, and enhances their overall effectiveness. A key focus of CMGV is ensuring the preservation of the user's mental map by employing a specialized layout algorithm that carefully adjusts the graph layout following each complexity management operation. By doing so, CMGV safeguards the user's understanding of the data and enhances the analysis of intricate relational information, all while maintaining the integrity of the user's mental map.

The rest of the thesis is structured as follows: Chapter 2 explains fundamental concepts related to graphs, layout algorithms, and the mathematical concepts used in this thesis as well as a comprehensive review of relevant literature. Chapter 3 presents the details of the CMGM (graph model) and CMGV (framework), while Chapter 4 provides an evaluation of the experimental results. Chapter 5 concludes the thesis by discussing potential future research directions.

# Chapter 2

# Basics & Related Work

## 2.1 Graphs

A *graph* $G = (V, E)$ is an abstract structure consisting of a non-empty set of vertices $V$ and a set of edges $E$ that connect pairs of nodes, $(u, v)$ where $u, v \in V$. The *source* and *target* nodes of an edge are said to be *adjacent* to each other, and the edge is considered *incident* to both nodes. The number of edges incident upon a node is called the *degree* $d(v)$ of that node. If the edges of a graph are assigned numerical values (weights), it is called a *weighted graph.* The *density* (D) of an undirected graph $G$ with $n$ vertices and $m$ edges can be calculated using the formula:

$$D = \frac{2m}{n \times (n-1)}$$

A *compound graph* is a graph that consists of multiple components, each of which is a graph on its own. To be formal, a compound graph $G = (V, E, F)$ contains a node set $V$, an adjacency edge set $E$, and an inclusion edge set $F$ [12]. The inclusion graph $T = (V, F)$ is a rooted tree, defined by nodes $V$ and inclusion edges $F$ (Figure. 2.1). Each edge $(u, v) \in F$ denotes that $u$ is a *parent node* of

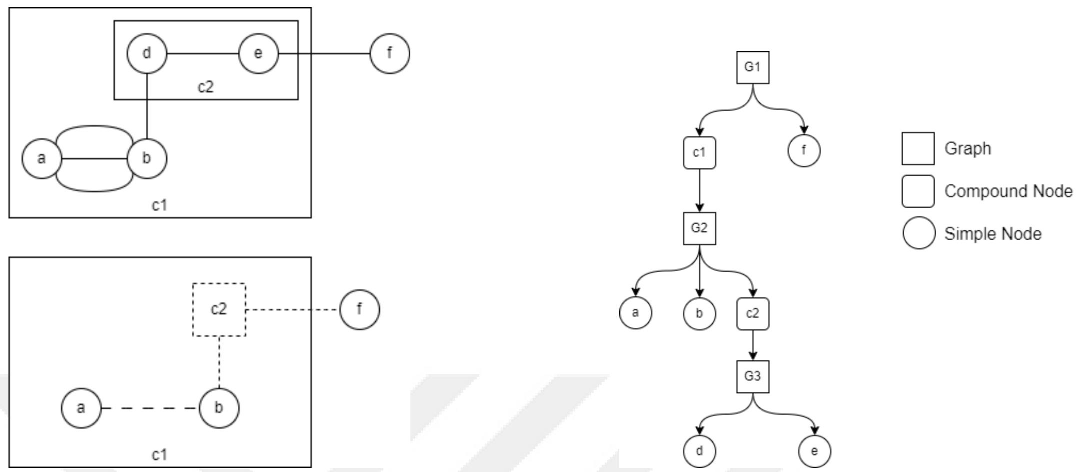



Figure 2.1: A compound graph $G = (V, E, F)$, where $V = \{a, b, d, e, f, c1, c2\}$, $E = \{e_1 = \{a, b\}, e_2 = \{a, b\}, e_3 = \{a, b\}, \{b, d\}, \{d, e\}, \{e, f\}\}$ and $F = \{(c_1, a), (c_1, b), (c_1, d), (c_2, e), (c_1, c_2)\}$, containing two compound nodes $c_1$ and $c_2$, and two inter-graph edges $\{b, d\}$ and $\{e, f\}$ (top left), its inclusion tree or nesting hierarchy (right) and the same compound graph where the compound node $c2$ and edges $e_1$, $e_2$ and $e_3$ were collapsed and since original edges no longer have both ends in the graph, they are represented with proxies called *meta edges* (bottom left)

$v$ (or $v$ is in the *child graph* or subgraph *nested* within a *compound node* $u$). A dedicated subgraph called the *root graph* contains all nodes $u$ such that $(u, v) \in F$ and $u$ is *not* in any child graph. Adjacency edges $\{u, v\} \in E$ are called *intra-graph* edges when both $u$ and $v$ are in the root graph or the same child graph, *inter-graph* when end nodes belong to different subgraphs. A key distinction between a compound graph and related variants such as hierarchically clustered graphs is that the former allows edges to connect with its compound nodes, while the latter does not. Compound graphs are employed to depict different levels of groupings or abstractions in data and are particularly useful for managing complexity through the use of expand-collapse operations [9] (Figure. 2.1).

A *meta edge* is a proxy representation of an edge or a set of edges between a pair of nodes that cannot be directly displayed in the visible graph. It arises as a result of complexity management operations, either directly or indirectly. For instance, a meta edge is generated by collapsing a set of edges between a pair of nodes, which is a direct consequence of the edge collapse operation. Similarly, a meta edge is formed when a compound node is collapsed, and it possesses an inter-graph edge, which is an indirect outcome of the node collapse operation.

A *path* in a graph is a non-repeating sequence of nodes connected by edges. A *connected* graph is a graph in which there is a path between every pair of vertices. A special case of a connected graph is called a *tree*, where any two nodes are connected by exactly one path. A tree in which a node is assigned as the *root* is a *rooted tree*.

A graph is called a *directed* graph if its edges have a direction. A path that starts and ends at the same node is called a *cycle*. A cycle in a directed graph with all edges in the same direction is called a *directed cycle*. A directed graph with no directed cycles is called a *directed acyclic graph (dag)*. A directed graph is*weakly connected* if its undirected version is connected.

Graphs are typically presented using pictorial representations. A poorly laid-out graph may lead to confusion for the user, whereas a well-organized one can provide aesthetic appeal and enhance the user's comprehension of the underlying data. While the criteria for a good layout may be subjective, some generally accepted criteria include minimizing the number of edge-to-edge crossings, minimizing the total edge length, maintaining uniform edge length throughout the graph, and minimizing the drawing area [13].

## 2.2 Graph Layout

*Graph layout* is the arrangement of nodes and edges organized in a graph to make it usable, understandable, and visually pleasing. This arrangement can be achieved using a function that maps each node to a distinct point in 2D/3D space and each edge to a Jordan curve in the same space, where endpoints correspond to the positions of the respective end nodes of the edge [13]. The quality of a layout is subjective, but there are some agreed-upon metrics for a good layout, including symmetry, minimal edge crossings, uniform edge lengths, and evenly spaced vertices [14]. A good layout algorithm also considers the size of nodes to avoid overlaps. In compound graphs, child nodes are placed inside their parent nodes' boundaries.

To create a layout, typically, nodes are initially given random positions and then adjusted to improve the layout. If a user is satisfied with the node positions and wants to make minor improvements, an *incremental layout* can be used to refine the existing layout while maintaining the user's mental map.

There are different types of graph layout algorithms, such as force-directed, spectral, hierarchical, orthogonal, and circular algorithms. Some are suitable for general purposes, while others are better suited for specific types of graphs (Figure 2.2). In this context, force-directed and spectral layout algorithms are of particular interest. Force-directed algorithms produce aesthetically pleasing layouts for both simple and compound graphs, while spectral algorithms are known for their speed.

### 2.2.1 Force-Directed Layout Algorithms

The force-directed layout, also known as spring embedder, is a widely used technique in automatic graph visualization [5], which draws inspiration from the real world by applying physical analogies to create visually appealing graphs. The concept originated from Eades' research [15], which employed metal rings to represent nodes that repel each other and springs to represent edges that exert forces on the rings proportional to their deviations from the "ideal" length.

To generate a layout, the nodes are initially assigned random positions. Then, a physical simulation is conducted, where the system is released, and the nodes move according to the cumulative forces acting upon them. After multiple repeated iterations, the system gradually reaches a stable state, minimizing the overall energy in the system (Figure 2.3). The run-time complexity of Eades' approach is $\mathcal{O}(|V|^2 + |E|)$ per iteration, where $\mathcal{O}(|V|^2)$ for the calculation of the repulsion forces and $\mathcal{O}(|E|)$ for the calculation of attractive (spring) forces.

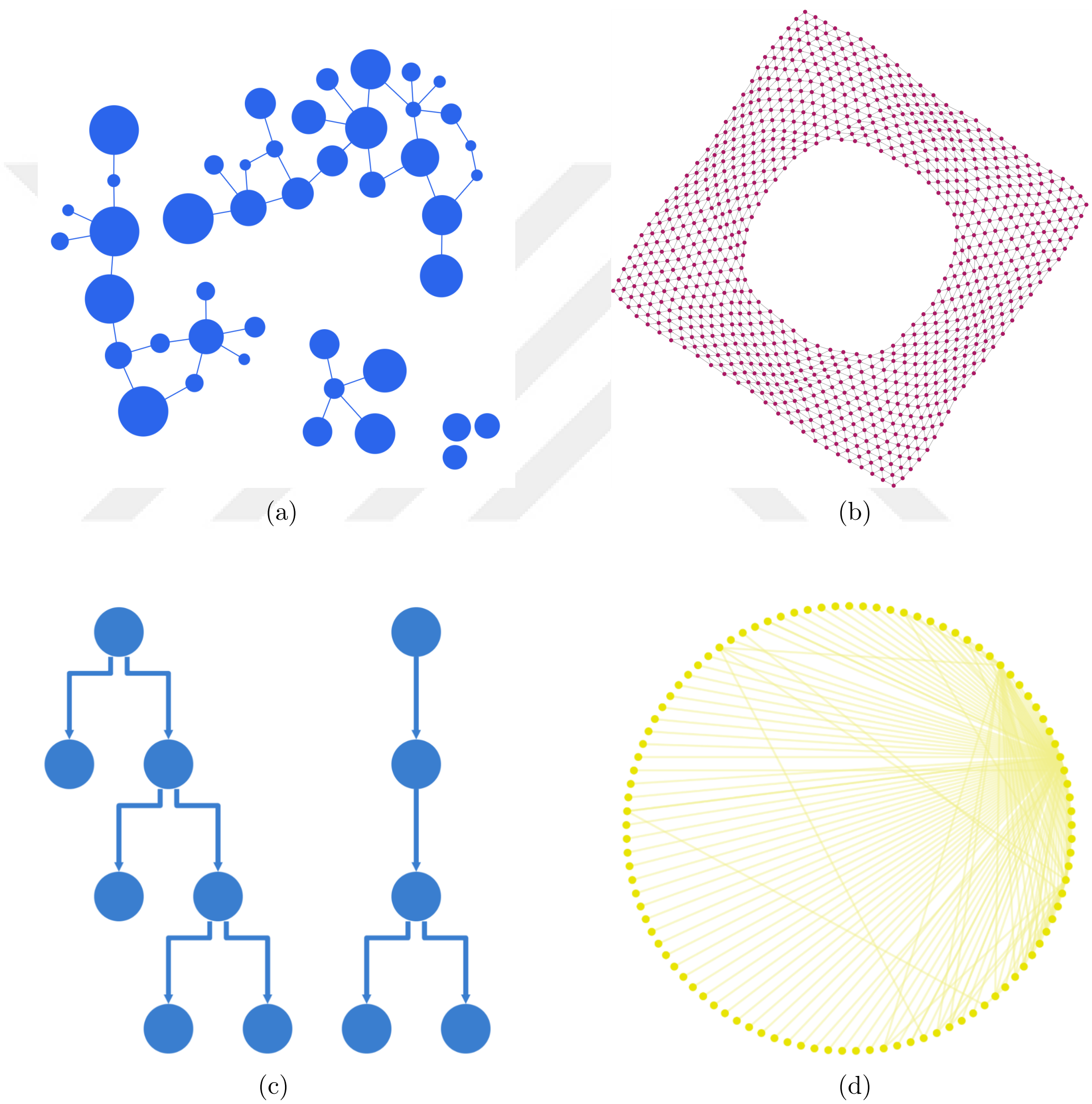



Figure 2.2: Sample results of various graph layout algorithms: a) force-directed layout, b) spectral layout, c) hierarchical layout with orthogonal edges, and d) circular layout [4]

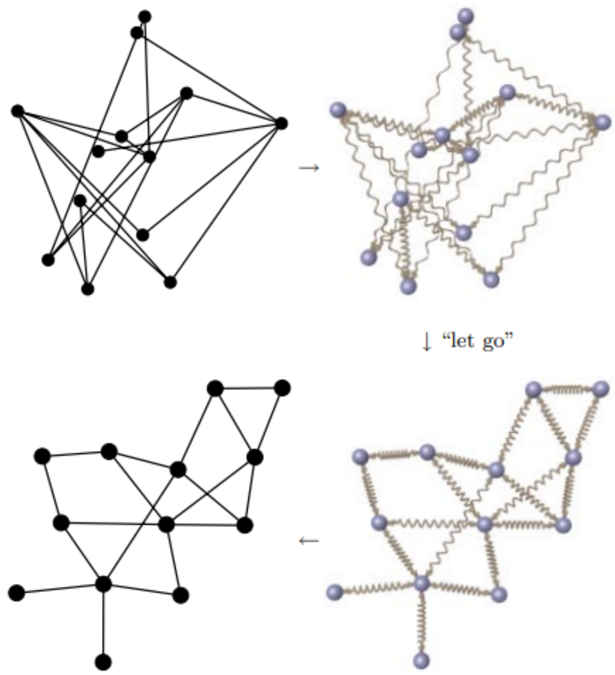



Figure 2.3: An illustration of the force-directed approach [5]

Two notable studies were influenced by Eades' force-directed approach. Kamada and Kawai [16] employed spring forces between all pairs of nodes, assuming that each pair of nodes is connected by a spring, and the optimal lengths of these springs were determined based on the distances between the node pairs in the graph. This approach involved minimizing the system's total energy by iteratively finding new positions for the nodes through the solution of partial differential equations.

In the second study, Fruchterman and Reingold [17] built upon Eades' approach and made improvements to it. They introduced repulsion forces between every pair of nodes and utilized a formula for spring forces similar to Hooke's law. Additionally, their algorithm aimed to achieve a uniform distribution of nodes on the drawing canvas. The equations used here for repulsion and attractive forces are as follows:

$$f_r(d) = -k^2/d$$

$$f_a(d) = d^2/k$$

where $d$ is the Euclidean distance between node pairs and $k$ is a constant that represents the ideal distance desired between nodes and calculated as

$$k = C\sqrt{\frac{area}{|V|}}$$

where *area* is the area of drawing canvas, $C$ is an experimentally found constant and $|V|$ is the number of nodes. The approach used here is called simulated annealing, which minimizes the energy of a system. The process begins with a high *temperature*, and as the nodes move during each iteration, the temperature gradually decreases. After multiple iterations, the system eventually reaches a stable state where the movement of the nodes ceases.

### 2.2.2 Measuring Effectiveness of Graph Layout

To evaluate the effectiveness of any layout algorithm, it is essential to record and analyze specific metrics that capture the quality of the resulting visualizations. These metrics include edge crossings, node overlaps, total edge length, and total area for static (non-incremental layout), whereas orthogonal ordering (the relative positions of the nodes about each other within the layout) and neighborhood proximity (a measure of separation between nodes within the layout) may be used for incremental layout. In our lab, we have developed a specialized extension called "cytoscape-layvo" [18] that enables the measurement of these metrics for graph layout and visualization.

While the calculation of most of the above-mentioned metrics is straightforward for most of them (e.g., node overlaps is calculating the sum of all the node-to-node overlaps for all the nodes in a given layout), there are some complex ones as well (e.g., neighborhood proximity and orthogonal ordering).

Neighborhood proximity is a metric that quantifies the distance or closeness between nodes in a graph layout. It measures how far apart or how closely positioned the nodes are relative to each other in the visual representation of the graph. To calculate the neighborhood proximity of node pairs *(a, b)* from graphs $G_1$ to $G_2$, distances $d_1$ and $d_2$ between the two nodes in the two graphs are used.

The change in their distances would be $\Delta d = |d_1 - d_2|$. Then a proximity score for each such pair of nodes is calculated using a function as follows.

$$f(\Delta d) = \frac{1}{1 + \frac{\Delta d}{l}}$$

where $l$ is the ideal edge length. For this to work nicely, distances are calculated from clipping point to clipping point (not center to center) as the layout algorithms strive to output the ideal edge length between clipping points. The resulting score will be in the range of 0 to 1, where 1 is the high level of neighborhood proximity in graphs $G_1$ and $G_2$.

Orthogonal ordering is the measure of the relative position of nodes in relation to each other within the layout, calculated as the polar angle $\theta$. To calculate the orthogonal ordering score, polar angles $\theta_1$ and $\theta_2$ for each pair of nodes are calculated in graphs $G_1$ and $G_2$. To measure polar angle $\theta$ for any node pair *(a, b)*, the position of node $a$ is taken as the center of the plane, and the angle is measured to the location of node $b$ (See Figure 2.4).

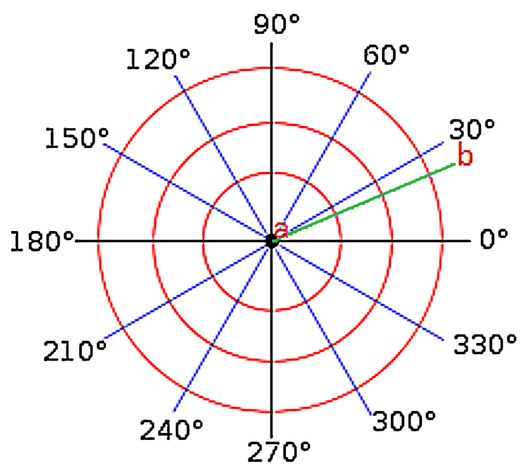



Figure 2.4: Polar angle between nodes $a$ and $b$ is about 20 degrees.

For similar graphs, the difference between the angles $\Delta\theta = |\theta_1 - \theta_2|$ in the two graphs $G_1$ and $G_2$ should be close to 0. Suppose $a, b \in V(G_1)$ and $a, b \in V(G_2)$, and $\theta_1$ and $\theta_2$ are the polar angles from $a$ to $b$ in the two graphs $G_1$ and $G_2$, respectively. Then, orthogonal ordering can be calculated using the following function

$$f(\Delta\theta) = 1 - \frac{\Delta\theta}{180}$$

where all $\theta$ values are calculated in degrees.

By comparing and averaging these metric values over multiple iterations, one can quantify the effectiveness of an algorithm in producing visually appealing and informative layouts for evolving networks.

## 2.3 Graph Complexity Management

Graph complexity refers to the challenges associated with the analysis and processing of graphs. The complexity of a graph arises due to several factors such as size, density, and structure. When confronted with a vast number of nodes and edges, effective complexity management becomes imperative. Large-scale graphs, which can range from hundreds to millions of nodes and edges, pose considerable challenges in terms of exploration and analysis. The *density* of a graph and levels of nesting in the compound graph can introduce complex patterns, which make analysis extremely challenging.

Complexity management aims to organize the graph structure and reduce the size of graph visualization to make it comprehensive and understandable for the end users, which is facilitated by operations like expand/collapse, filter/unfilter, or hide/ show [9]. The fundamental principle of any complexity management operation is that it can be undone, and the graph can be restored to its previous state. All complexity operations are designed in pairs, a sort of do and undo pair, such as the collapse/expand operations. Collapse operations replace the compound node and its child graph with a single proxy node and expand operation replaces the proxy node with the original compound node and its child graph, thus restoring the graph back to its previous state (Figure 2.5). Filter/unfilter and hide/show operations work similarly in pairs. The first of these operations removes nodes and edges from the graph based on some user-defined criteria, while the latter operations restore the nodes removed.

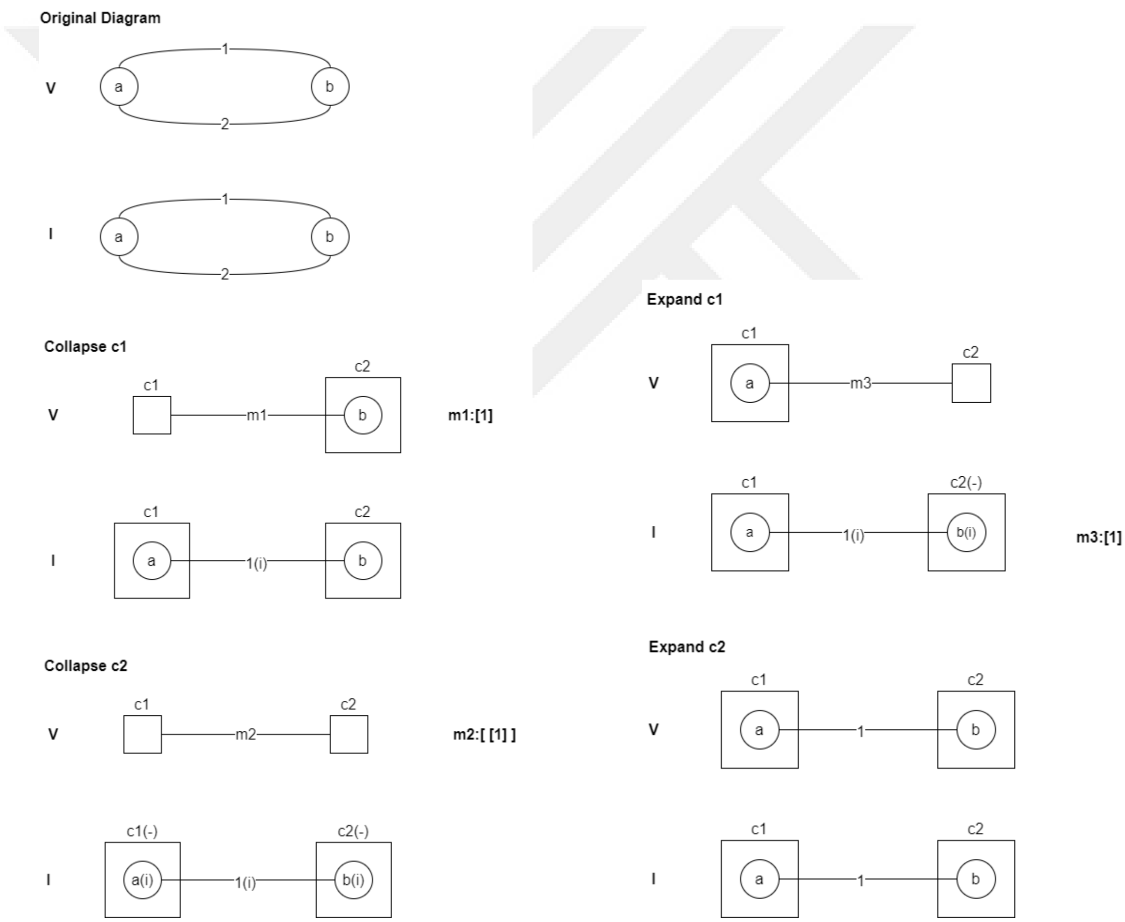



Figure 2.5: A series of complexity management operations where $c1$ and $c2$ are first collapsed and then expanded in the reverse order.

By applying complexity management techniques, these challenges posed by graph complexity can be overcome, enabling more efficient analysis, better utilization of computational resources, and improved decision-making based on the insights extracted from the graph data.

## 2.4 Related Work

*Force-directed* algorithms, commonly referred to as spring embedders, are the most popular and easy-to-use graph layout methods. These algorithms mimic physical systems governed by fundamental laws of physics, making them intuitive and effective for graph visualizations. By treating graph nodes as particles with electrical charge and edges as springs, force-directed algorithms create an interactive layout where nodes repel each other when they come too close, while edges attract their endpoints when they are too far apart [13][17]. The ultimate goal of these algorithms is to iteratively move the particles, i.e., the nodes, towards a stable state where all forces are balanced or the system's energy is minimized. Among the force-directed algorithms, many employ a temperature concept inspired by the simulated annealing technique. This temperature aids in swiftly reaching a local minimum of the energy function. An exemplary force-directed layout algorithm that supports compound nodes is fCoSE [11] (a rapid graph layout algorithm that supports compound graphs, capable of arranging large, complex graphs efficiently while taking into account various graph layout constraints, providing a fast and effective visualization solution.), whose details are given in Section 2.2.1.

In interactive visualization applications graphs alter and evolve. When dealing with evolving graphs, where nodes and edges are added, removed, or modified, a static layout becomes insufficient. *Incremental layout* is the method of calculating a new layout while taking into account any current locations [13]. Instead of starting from scratch each time, this method builds upon the previous layout, leveraging the user's familiarity with the graph's structure. By employing sophisticated mathematical techniques, it tries to minimize edge crossings, optimize node placements, and enhance the overall legibility of the resulting graph drawings. An incremental layout aims to clean up the drawing while maintaining the user's mental map, which alludes to the nodes' present locations.

One of the key advantages of incremental layout is its ability to improve the user experience in interactive visualization applications. As the graph evolves, the incremental layout algorithm dynamically adjusts the position of nodes and edges, resulting in a more organized and intuitive representation. This enables users to observe changes in the graph more easily and make informed decisions based on the updated information.

The need to view ever-larger sets of relational data has increased the demand for more sophisticated complexity management methods. The complexity management issue in the setting of large graphs has been the subject of many studies [12][9]. Unwanted information can be concealed or ghosted and subsequently revealed upon request, according to some strategies. Some methods make use of lenses, such as fish eye lenses, which enlarge a particular area of the drawing to enable users to zoom in on a particular area of a vast network. Others, which support multiple views and nesting and use expand-collapse operations to manage complexity, rely on techniques for building clusters based on the graph-theoretic properties of the given dataset, if not any domain-specific information.

There are many challenges when it comes to creating an effective graph layout. One such challenge is incrementally adjusting the layout to expand a collapsed compound node. An expanding compound node takes a lot of space depending on the number of nodes and edges in its child graph. To make accommodations for a compound node, surrounding the nodes need to be moved, which disturbs the whole graph and potentially destroys users' metal map. To overcome this, our lab has previously developed a layout algorithm [9] which employs the fisheye algorithm. Inspired by the fisheye lens used in photography, where the center of the image appears magnified while the edges are compressed, the fisheye algorithm for graph layout distorts the graph to emphasize the central region while compressing the outer regions. It creates empty space by taking the expanding compound node as the central region and compressing the surrounding region. The algorithm works far better than a plain incremental layout but has its own limitation and drawbacks. Firstly, there is no sophisticated size estimation for the expanding compound node, which distorts the graph unnecessarily too much; with accurate size estimation expansion, the resulting distortion becomes proportional to the size of the expanding compound. Also, the layout algorithm [9] saves the position of the compound node before the collapse and then performs the fisheye algorithm to space at that position which works well for consecutive expand and collapse but does not account for node movement in between the two operations. This oversight creates a translation problem, which results in distortion and displacement of nodes in both the current and previous location of the expanding compound node. This much movement makes it challenging to safeguard users' mental maps. In cases where the expanding compound node is in the child graph of another compound node, then the algorithm fails to take hierarchy into consideration and compresses the entire surrounding graph to make space for expansion. In case of multiple expansions (expand all operation expands on the level at a time), the algorithm performs layout for expand operation, which is costly.

In-depth studies on complexity management often fail to address implementation concerns, such as preserving the user's mental map when modifications are made to the graph's topology and geometry. More importantly, because these complexity management studies were conducted in isolation, the techniques they produced did not account for the adjustments made by other complexity management operations. It is crucial to realize that complexity management strategies are more frequently combined to explore graphs and analyze relational data in interactive visual applications in the real world. When these complexity management methods operate independently, it leads to potential inconsistencies, incorrect behaviors, and inefficient utilization of computing resources, which are all big issues for real-world applications.

# Chapter 3

# Framework and Methods

CMGV is a novel framework designed to address the complexities in management with the utmost efficiency. Our work is dedicated to crafting a complexity management solution that effectively analyzes large-scale relational data represented as graphs. By unifying various complexity management techniques, it ensures seamless cooperation, with careful adjustments to the graph layout after each operation to safeguard the user's mental map. Compound graphs, a critical aspect of complexity management and abstraction, play a fundamental role in representing most of large-scale relational data. CMGM allows users to confidently navigate the intricacies of their data landscape while ensuring clarity and efficiency. It mainly consists of three components (Figure 3.1):

- CMGM is the unique graph model that harmoniously integrates commonly used complexity management techniques, enabling them to collaborate seamlessly. It also comes equipped with specialized layout algorithms that diligently preserve the user's mental map after each complexity management operation.
- A specialized complexity management extension tailored for the user's graph rendering library (RL).
- A graph rendering library (RL).

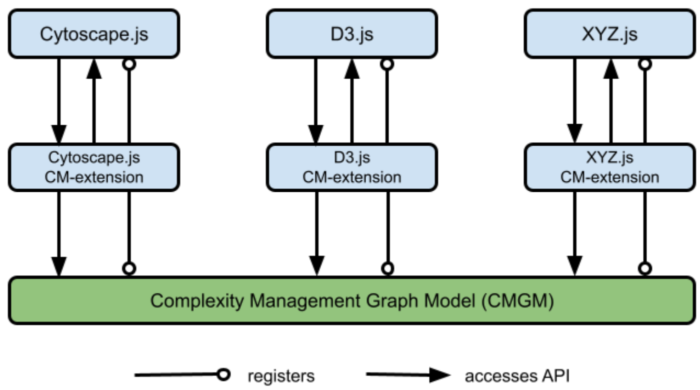



Figure 3.1: An illustrative diagram presenting the architecture of an application leveraging CMGV framework

## 3.1 Unified Graph Model

Our first step involves creating a novel complexity management graph model (CMGM) designed to facilitate the most frequently employed complexity management operations on graph elements:

- *filter* / *unfilter* a set of graph elements,
- *hide* / *show* a set of graph elements,
- *collapse* / *expand* contents (i.e., child graph) of a compound node or multiple edges between a pair of nodes.

CMGM has been meticulously crafted to possess a key characteristic: its independence from any specific rendering layer. This unique quality allows CMGM to seamlessly integrate with various graph rendering libraries (RLs). The integration between RL and CMGM is achieved through a specialized extension of

RL, enabling a smooth flow of information to maintain synchronization between the graph models of both components. The graph model of CMGM has been thoughtfully designed to support compound graphs, drawing inspiration from the model utilized in [12].

In CMGM, the representation of a graph on the renderer's side involves the use of two compound graphs, effectively segregating the viewable graph within RL. This ensures that the user's visual experience remains undisturbed by complexity management operations. The two graphs are named the *visible graph* and the *main graph*, each serving a specific purpose in the widely adopted model-view architecture. The *visible graph* contains the elements that are currently visible to the user. On the other hand, the *main graph* encompasses all the graph elements, including those that might have been temporarily removed due to complexity management operations. This model-view architecture enables a seamless flow of information between CMGM and RL. The *main graph* acts as a foundational model that serves as the basis for both view graphs: the *visible graph* within CMGM and the graph rendered by RL. By adopting this approach, CMGM ensures that complexity management operations are performed on the *main graph*, preserving the integrity and consistency of the data. Simultaneously, the *visible graph* provides the user with a clear and uncluttered view of the graph elements, enhancing the overall user experience.

### 3.1.1 Internal Structure of CMGM

Complexity Management Graph Model (CMGM) is built on a robust and well-organized internal architecture (proposed in Figure 3.1) that ensures its efficient functioning. At its core, CMGM comprises several essential components translated into a set of interconnected classes (see Figure 3.3) that work together to enable seamless complexity management operations on graph-based data.

`ComplexityManager` class serves as a crucial Facade responsible for mediating communication between the CMGM core and other components through a well-defined API. This class plays a vital role in ensuring seamless interaction and synchronization between CMGM and the graph of RL. The API provided by the `ComplexityManager` class includes a set of functions that cater to two main categories of operations, the ones utilized to keep CMGM in sync with any topological changes that occur on the renderer's side, such as adding or removing nodes, reconnecting an edge, or changing the parent of a node (via `addNode`, `removeNode`, `reconnect` and `changeParent`, respectively) and the ones related to the complexity management operations such as filtering, hiding and collapsing nodes (via `filter`, `hide` and `collapseNodes`, respectively).

As mentioned earlier, CMGM maintains two distinct instances of the `GraphManager`: one for the visible graph and another for the main graph. These graphs play a critical role in managing the elements, differentiating between those that are currently visible and those that are temporarily hidden due to complexity management operations. This differentiation is achieved through the use of an `isVisible` flag associated with each element. The `GraphManager` instances are interconnected through the `siblingGraphManager` instance variable, enabling seamless communication and data sharing between the visible and main graphs. When a complexity management operation necessitates the removal of an element from the visible graph, the corresponding element's `isVisible` flag in the main graph is reset, indicating that it is no longer visible. On the other hand, if an element needs to be moved to the visible graph, its `isVisible` flag in the main graph is set accordingly. Additionally, the element is copied from the main graph to the visible graph to ensure its presence in the visible view.

Within CMGM, each compound node and the root graph contain a nested graph represented by `Graph` objects. These `Graph` objects serve as containers that store the nodes and edges within them, while also maintaining a reference to their parent (compound) node. In the visible graph, each `Graph` object knows of its corresponding `siblingGraph` in the main graph and vice versa, except for the invisible objects in the main graph. Invisible objects in the main graph know of their corresponding `siblingGraph` only if their parent node is not collapsed in the visible graph. This awareness facilitates seamless communication between the two graphs, ensuring precise synchronization when there are changes in the visibility states of elements.

`Node` and `Edge` classes are subclasses of `GraphObject` class inheriting common properties like `ID`, `owner`, `isVisible`, `isFiltered` and `isHidden`. The latter three instance variables are mainly used in the main graph to manage nodes and edges. For `Node` objects, in addition to the common properties, each instance stores its incident edges, child graph, and a flag named `isCollapsed`. The `isCollapsed` flag indicates whether the node is currently collapsed or expanded. On the other hand, `Edge` instances, apart from the shared properties, store references to their source and target nodes. Additionally, each `Edge` object includes a flag named `isInterGraph`, indicating whether the edge represents an inter-graph edge.

The meta edge class is a subclass of the `Edge` class and is responsible for representing edges that become invisible as a result of a collapse operation, either on a compound node or a set of edges that are collapsed together. These meta edges store their original edge(s) in a unique data structure, as they may represent a nesting hierarchy of edges corresponding to the nesting history (Figure 3.2 & Figure 3.4). It is important to note that a meta edge can only exist within the visible graph and is not present in the main graph. Each `GraphObject` within CMGM maintains a reference to its owner `Graph`, which in turn is associated with its owner `GraphManager`. Additionally, each `GraphManager` has access to its corresponding `ComplexityManager` via its owner instance variable.

In the main graph, if a graph element (node or edge) is marked as filtered or hidden, it is also automatically marked as invisible. This ensures that elements that are filtered or hidden are not visible to the user. For a node to become visible in the visible graph, certain conditions must be met. Firstly, the node itself should not be marked as filtered or hidden. Additionally, none of its ancestors (parent nodes) should be marked as filtered, hidden, or collapsed. Similarly, for an edge to become visible in the visible graph, it must meet specific criteria. The source and target nodes of the edge should be present in the visible graph. Additionally, the edge itself should not be marked as filtered or hidden to be visible. Lastly, the edge should not be part of a meta edge in the visible graph. There will be four hash maps to keep track of the different elements and their correspondence in the graph, including nodes, edges, and meta edges. One of these maps, called

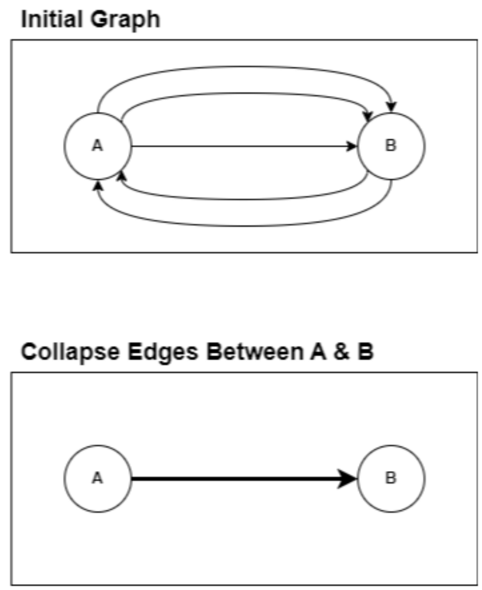



Figure 3.2: A scenario illustrating an edge collapse operation leading to the formation of a singular meta edge.

`edgeToMetaEdgeMap`, keeps track of all the edges by a meta edge ID of the parent meta edge, if any.

### 3.1.2 Operations Provided by CMGM via API Functions

API functions in our Complexity Management Framework can be classified into four categories. The first category encompasses topological changes, while the remaining categories involve complexity management operations. The first category includes functions such as `addNode`, `addEdge`, `reconnect`, and `changeParent`. These functions play a crucial role in maintaining synchronization between the renderer's side and CMGM, ensuring that changes are precisely reflected. In the second category, we find `filter` and `unfilter` functions, allowing users to remove or restore graph elements based on user-specified filtering rules, which can be defined based on the properties of the graph elements. The third category comprises functions like `hide`, `show`, and `showAll`, which have similar functionality to `filter` and `unfilter` functions but can be applied without predefined

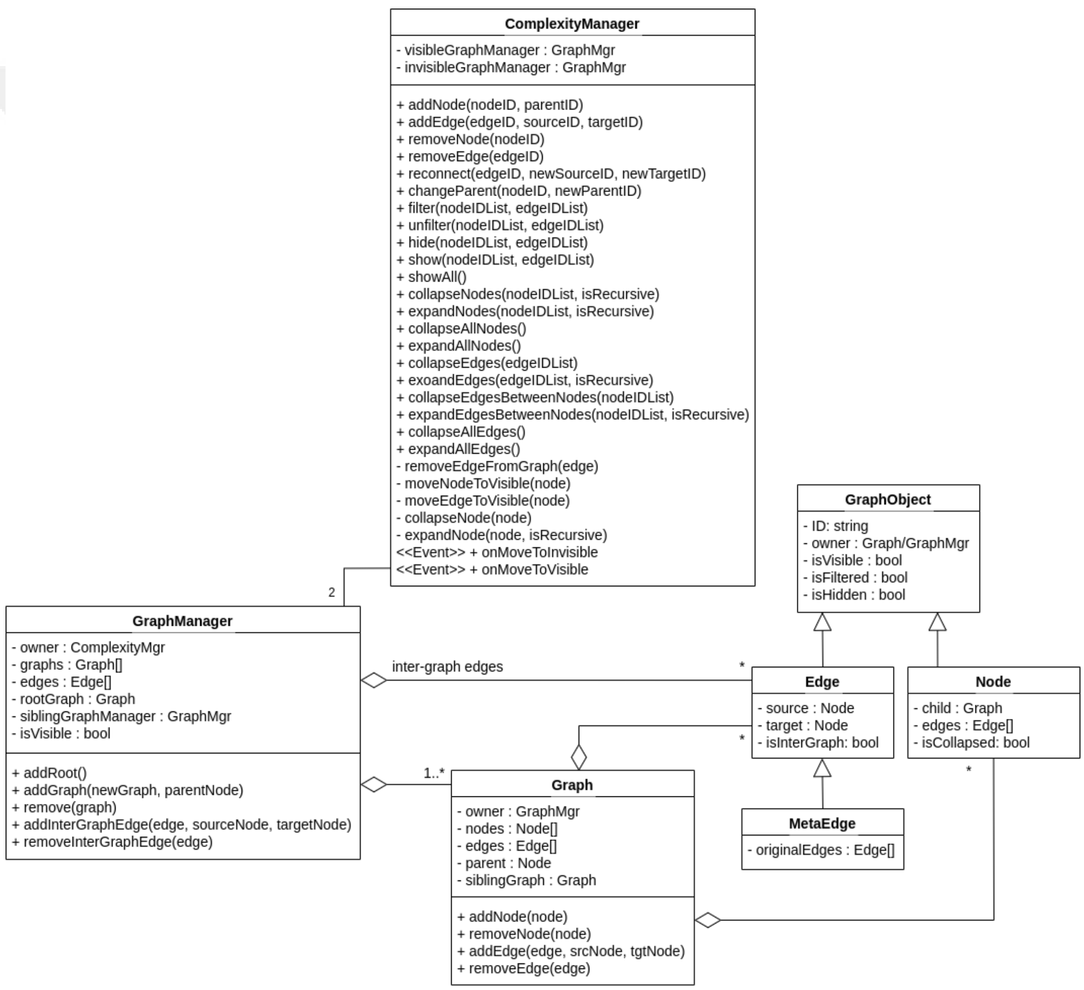
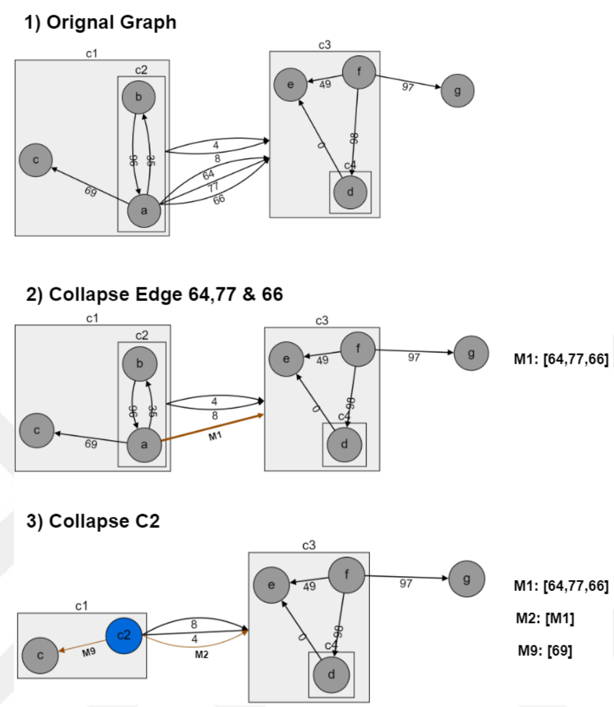
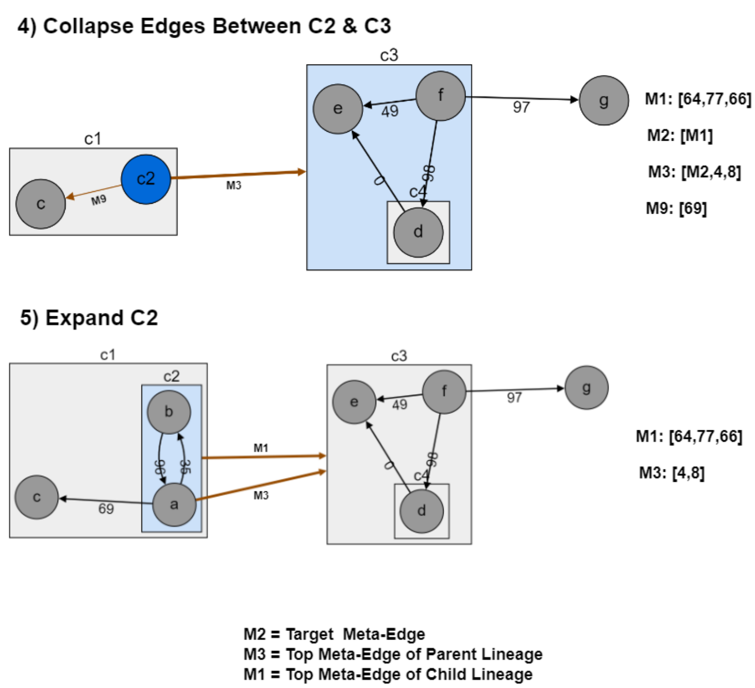



Figure 3.3: A UML class diagram illustrating the structure of CMGM.




(a)




(b)

Figure 3.4: An illustration depicting a combination of edge and node collapse/expand operations to visualize the manipulation of meta edges.

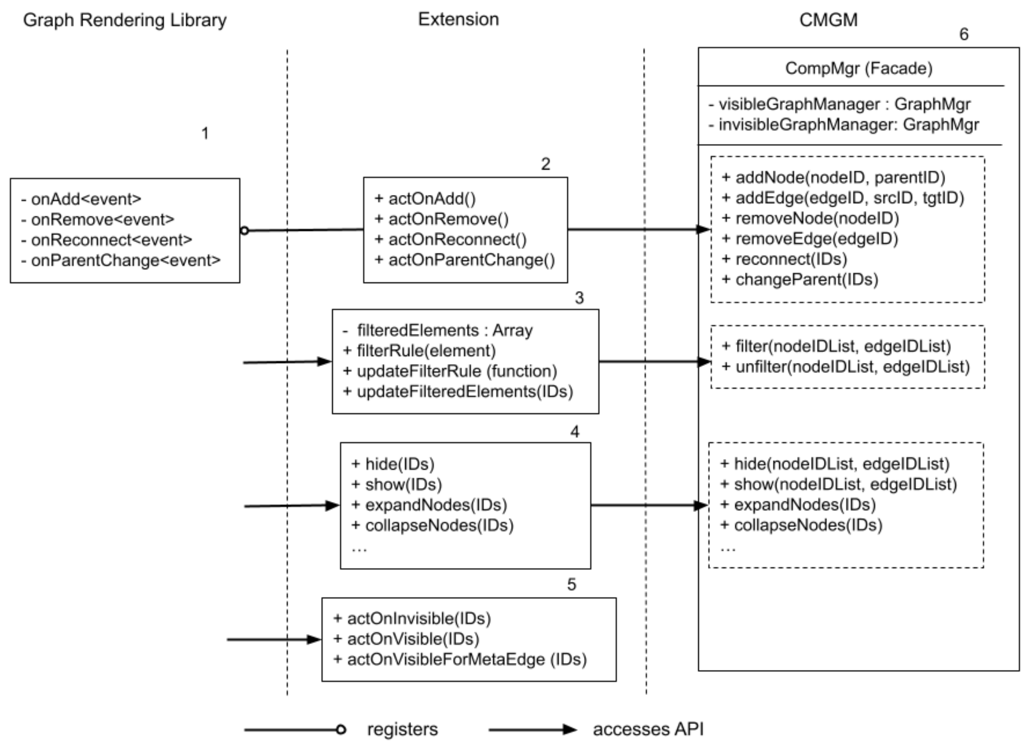



Figure 3.5: An activity diagram depicting the sequence of communication between different components within CMGM.

rules. These functions enable users to selectively add or remove sets of nodes from the graph.

When updating the filtering status of a node, it is essential to consider its hidden status, and vice versa, especially when moving it between the visible and main graphs. For example, if a node is both filtered and hidden, simply unfiltering it would not make it visible if its hidden status is still true. Similarly, showing it alone would not be enough if its filtering status remains true.

The last category involves functions related to node and edge collapse/expand operations, such as `collapseNodes`, `expandNodes`, `collapseAllNodes`, `expandAllNodes`, `collapseEdges`, `expandEdges`, `collapseEdgesBetweenNodes`, `expandBetweenNodes`, `collapseAllEdges`, and `expandAllEdges`. These operations enable users to reduce the graph's size by collapsing a compound node (which removes child elements of the collapsing compound node from the visible graph) or by representing multiple edges between a node pair as one edge (meta edge). Later, these removed elements can be restored as needed by expanding a collapsed node or edge.

## 3.2 Communication Between Graph RL and CMGM

Communication between RL and CMGM will be done through a tailored extension for RL, ensuring synchronization of the graphs in both components (see Figure 3.1). Whenever a user initiates a change in topology, such as adding/removing graph elements, modifying edge source/target, changing a node's parent, or performing a complexity management operation, these changes will be reflected in both views. Hence, it is crucial to establish a well-defined communication structure between the graph model of the renderer and CMGM. The communication structure design is outlined as follows (Figure 3.5):

- RL will provide events triggered by topological changes occurring in the renderer's graph model. These events include the addition/removal of graph elements (node/edge), changes in the source/target of an edge, and modifications to the node's parent (Box 1 in Figure 3.5).
- When events in Box 1 in Figure 3.5 will be triggered they will call their respective function in Box 2 in Figure 3.5, which include `actOnAdd`, `actOnRemove`, `actOnReconnect` and `actOnParentChange` function. Each action function in Box 2 (Figure 3.5) should be registered to the corresponding event in RL (e.g., `actOnAdd` function into `actOnAdd` event of Box 1 in Figure 3.5). To put it in simpler terms: whenever an event occurs in Box 1 (Figure 3.5), and this event leads to a change in the renderer's graph

model, it will then call the corresponding function in Box 2 (Figure 3.5). The action function applies the changes made on RL side that will synchronize CMGM with RL's graph model by using the API provided through the Facade class (Box 6 in Figure 3.5) of CMGM. Once the synchronization process is complete, it will proceed to invoke the `updateFilteredElements` function (explained in detail below) in order to ensure that the currently filtered elements are kept up to date. To example, whenever a new graph element is introduced to the graph model of RL, the `onAdd` event will be triggered, subsequently leading to a call to the `actOnAdd` function. When the `actOnAdd` function is called, it calls the `addNode/addEdge` function, which adds a new graph element into CMGM's graph model. Following this, the `updateFilteredElements` function is called, as the newly added elements might need to be filtered or even impact the filtered status of other graph elements. It's crucial to follow a top-down approach when adding new graph elements to maintain a well-structured graph and avoid any potential errors. This means adding elements from parents to children and nodes before edges. By doing so, we can ensure that there are no missing graph elements, such as a parent or an end node. Conversely, when it comes to removing graph elements from the CMGM graph model, it's best first to remove edges and then to remove nodes for the simplicity of the operations.

- The extension includes a `filterRule` function, which allows users to customize and define specific filtering rules according to their preferences. For example, they can choose to filter edges with the type "relationship" or nodes with lengths greater than 10. When applied to an element in the graph, the `filterRule` function determines whether the element should be filtered or not. If it meets the criteria, the function returns *true*; otherwise, it returns *false*. Additionally, the extension provides an API function called `updateFilterRule`, allowing users to update the `filterRule` function at any point during the application's flow (Box 3 in Figure 3.5).

- The `filteredElements` array will store the IDs of the filtered graph elements (Box 3 in Figure 3.5).

- The extension will include an additional function called `updateFilteredElements` function, which will get triggered whenever the user makes changes to the `filterRule` function using the `updateFilterRule` option or when specific events in RL are activated. When alterations are made to the `filterRule` function, it can render the currently filtered elements unfiltered or vice versa. When one of the events in RL is triggered, the corresponding action function also requires the `updateFilteredElements` function to be called (as explained above). This is because, for example, a newly added element might satisfy the existing `filterRule`; hence, it should be promptly filtered upon addition. On the other hand, a removed element may still exist in the `filteredElements` array, and to maintain consistency in operations, it needs to be removed from this array. The `updateFilteredElements` function applies the `filterRule` function on each graph element. Initially, it identifies the new list of IDs for elements that should be filtered; then it compares this new list with the `filteredElements` array. If elements need to be added to `filteredElements`, the `updateFilteredElements` function will include them in the array and utilize the `filter` function in the CMGM API to carry out the filtering process. Conversely, if there are elements that need to be removed from `filteredElements`, the `updateFilteredElements` function will exclude them from the array and utilize the `unfilter` function in the CMGM API to undo the filtering (Box 3 in Figure 3.5).

- Every element in the main graph will have an `isFiltered` flag with default value *false*. `filter` function is triggered by `updateFilteredElements` function with a set of graph elements. The `filter` function removes the given elements from the visible graph, sets their `isFiltered` flags to `true`, and marks them as `invisible` in the main graph. While filtering a graph element some adjacent elements may also need to be removed from the main graph without being exclusively filtered. For instance, if a node is filtered, its incident edges also become invisible. Thus, the function will remove these edges from the visible graph and mark them invisible. Similar in reverse applies the `unfilter`, the `unfilter` function will set the `isFiltered` flags of these elements to *false* and then transfer the invisible elements to the visible graph based on certain rules (explained in Section 3.1.1 in detail).

Similar to filter function, the `unfilter` function may also need to transfer some other graph elements to the visible graph. If a node gets unfiltered, its incident edges, which had turned invisible during the filtering process, must be restored and made visible again. In such a case, these graph elements will be moved back to the visible graph, but their `isFiltered` flags will remain unchanged. Finally, the `invisible` flag will be removed from the transferred graph elements.

Complexity management operations like `hide/show` and `expand/collapse` (Box 4 in Figure 3.5) can make graph elements visible or invisible. In such cases, the complexity management functions in the extension API will call the corresponding functions of CMGM (Box 6 in Figure 3.5). For instance, the `expandNodes` function in Box 4 in Figure 3.5 will trigger its corresponding `expandNodes` function in Box 6 in Figure3.5. Complexity management functions such as `hide` and `show` perform their respective operations which may include moving some elements from to visible graph or making some elements invisible by removing from the visible graph and marking them invisible in the main graph.

- `actOnInvisible`, `actOnVisible` and `actOnVisibleForMetaEdge` are action functions defined in the extension side (Box 5 in Figure 3.5), when an element is marked as *invisible* in the invisible graph, `actOnInvisible` function places the element in the renderer's side to the scratchpad of the renderer, then temporarily turns off the `onRemove` event (Box 1 in Figure 3.5) and removes the element from the renderer's side. Lastly, it will turn on the `onRemove` event (Box 1 in Figure 3.5). In order to avoid triggering any default or user-defined remove actions (which should only be triggered by when the user removes any graph element), the framework turns off `onRemove` event temporarily. Conversely, when an element is added to a visible graph the `actOnVisible` function removes the element from the scratchpad of the renderer side, turns off the `onAdd` event (Box 1 in Figure 3.5) temporarily and adds the element to the renderer's side. It will turn the `onAdd` event (Box 1 in Figure 3.5) back on. Same goes for `actOnVisibleForMetaEdge` function except it is designed to explicitly add the meta edges to the renderer's side instead of basic graph elements (nodes or edges). The purpose of having a separate function for meta edges is to deal with the unique structure of meta edge's data. Meta edges do not exist in the main graph of CMGM, so a new instance needs to be created before it can be added to the visible graph and render side.

## 3.3 Complexity Management Operations

As mentioned above, our framework supports three sorts of complexity management operations. Filter/unfilter, and hide/show, involve changing the visibility, filter, and hidden flag of graph elements in order to make them visible or invisible on the renderer side. Expand and collapse operations, on the other hand, may use meta edges, which are proxy representations of edges that cannot be directly represented in the visible graph. We will focus on these operations in the following section, specifically on the creation and destruction of *meta edges*.

### 3.3.1 Collapsing Nodes or Edges

There are two ways meta edges can be created. The first one is through an operation called *edge collapse*, where multiple edges between the same pair of nodes are combined into a single meta edge (refer to Figure 3.2). The second way is through function *node collapse* (where a compound node is collapsed, and its contents, child graph, are no longer visible). When the node collapse operation is performed, the child nodes of the collapsing compound are removed from the visible graph, and either the source or target node of each intergraph edge (edge that has both end nodes in different graphs, in this case, one inside the collapsing compound node and one outside) will be removed. Consequently, a meta edge is created as a proxy representation between the compound node and the original end node outside the collapsing compound (see Figure 2.1 for an example).

In CMGM, each meta edge maintains a list known as the "*original edges list*", which contains the IDs of all the edges it represents. This relationship can be visualized as a parent-child relationship, where the meta edge serves as the parent and the represented edges serve as children. These children may themselves be other meta edges, giving rise to a hierarchical relationship that we refer to as a "*meta edge lineage*". It's important to note that the length of the original edges list signifies the type of meta edge depending on how the meta edge was created. Meta edges formed through a node collapse operation have the original edges list of length one (representing a single intergraph edge); while, the meta edges created through edge collapse operation have the original edges list of length two or more. The first kind of meta edges can not be expanded via `expandEdges` operation. This variation in the length of the original edges list can be used during the implementation process to distinguish a node collapse meta edge from other meta edges within a meta edge lineage.

There is an exception to the rule of creating a new meta edge for an inter-graph edge, which occurs during node collapse. If a new meta edge is created for an intergraph meta edge, instead of creating a new meta edge to represent an existing intergraph meta edge, either the source or target of the existing meta edge is changed to be the collapsing node. This is done as an optimization measure to reduce both the number of meta edges created and the lengths of *meta edge lineages*.

Algorithm 1 is a function to collapse a compound node by removing its child graph and creating meta edges if needed. The algorithm iterates over each child node and then their respective edges and removing them from graph and introducing meta edges if needed (both of which are constant time operations); so, the runtime complexity is expected to be linear, O(V + E), where V is the number of nodes and E is the number of edges in the graph. Similarly, Algorithm 2 is a function to collapse a group of edges between a node pair into a single meta edge by removing edges from the visible graph and creating the meta edge. The algorithm iterates over each edge that is part of the meta edge, makes it visible and after all edges are processed it deletes the meta edge (which is a constant time operation). The runtime complexity is expected to be linear with respect to the number of edges, i.e., O(E) where E is the number of edges in the graph.

### 3.3.2 Expanding Nodes or Edges

Expand node operation (opposite of collapse node), brings back all child nodes (not filtered and not hidden) of the expanding compound node to the visible graph, then brings back their edges. Some of these edges were intergraph edges which got represented by meta edges; thus, to bring them back, the corresponding meta edges need to be destroyed and all references to them need to be removed.

As mentioned above, a meta edge can be part of a *meta edge lineage*; so when destroying a meta edge, it is essential that all its references be removed from its meta edge lineage. In order to do that, the position of *node-collapsed meta edge* in the meta edge lineage is to be determined, which can be done by starting with the simple intergraph edge (edge to be brought back to the visible graph) and going upwards in the lineage until the meta edge is found (*target meta edge*). Then, the *target meta edge* is removed from the lineage, thereby dividing the lineage into two separate parts: the first lineage will start from the immediate child of the *target meta edge*, and the second one will start from the *top meta edge* of said lineage and will end at the parent of the *target meta edge*. The second lineage will need a slight reconfiguration after the loss of a member. The following are the possible scenarios:

1. The *target meta edge* is *top meta edge* and has no parent (i.e., a single-member lineage).
2. The *target meta edge* has a parent but has only one child; with one gone, the parent meta edge is unnecessary. In this case, the sibling of the *target meta edge* is added to the original edge list of its grandparent meta edge. If there are none, then it will be a single-member lineage.
3. The target edge has multiple siblings. Then, no reconfiguration is necessary.

After making necessary changes to the lineage, the *target meta edge* gets destroyed and the *top meta edges* of both lineages are added to the visible graph (assuming they are not filtered or hidden and not already in the visible graph).

To bring back an edge to a visible graph, both end nodes must be visible; if one of the end nodes is not visible, then a meta edge is created between the visible end node and the visible parent of the invisible end node. In case both end nodes are invisible, a meta edge is created between the lowest visible parents in the hierarchy of both invisible end nodes.

Algorithm 3 is a function to expand a compound node by making its child graph visible and updating the relevant meta edge accordingly. The algorithm is expected to have a linear runtime complexity as expand node function makes all children nodes and edges visible. It iterates over each child node and then their respective edges making them visible. Thus expected runtime complexity is O(V + E). Similarly, Algorithm 4 is a function to expand a meta edge by making the original edges visible depending upon their filter and hidden status. As stated, the function iterates over all the edges that are part of expanding meta edge and so the runtime complexity is expected to be linear with respect to the number of edges (i.e., O(E)).

**Algorithm 1**

**function** COLLAPSENODE(Node $v$)
  $invisibleNodeList \leftarrow []$
  $invisibleEodeList \leftarrow []$
  $G \leftarrow v.child$
  $visited \leftarrow$ empty set
  $stack \leftarrow$ empty stack
  $vGM \leftarrow visibleGraphManager$
  $stack$.PUSH$(v, null)$
  **while** $stack \neq \emptyset$ **do**
    $(current, parent) \leftarrow stack$.POP$()$
    **if** $current \notin visited$ **then**
      $visited$.add$(current)$
      $invisibleNodeList \leftarrow invisibleNodeList \cup current.ID$
      **for each** $incidentEdge \in current.edges$ **do**
        $invisibleEodeList \leftarrow invisibleNodeList \cup incidentEdge$
        **if** $incidentEdge$.ISINTERGRAPHEDGE$()$ **then**
          Create a meta edge for the $incidentEdge$
      **for each** $neighbor$ of $current \in G$ **do**
        **if** $neighbor \notin visited$ **then**
          $stack$.PUSH$(neighbor, current)$
  $vGM$.REMOVEGRAPH$(node.child)$
  $nodeInInvisible \leftarrow GM.nodesMap$.GET$(node.ID)$
  $nodeInInvisible$.SETISCOLLAPSED$(true)$
  **for each** $nodeID$ in $invisibleNodeList$ **do**
    $nodeInInvisible \leftarrow GM.nodesMap$.GET$(nodeID)$
    $nodeInInvisible$.SETISVISIBLE$(false)$
  **for each** $edgeID$ in $invisibleEodeList$ **do**
    $edgeInInvisible \leftarrow GM.edgesMap$.GET$(edgeID)$
    $edgeInInvisible$.SETISVISIBLE$((false)$

**Algorithm 2**

```
function CollapseEdges(List edgeIDList)
  GM ← mainGraphManager
  vGM ← visibleGraphManager
  firstEdge ← vGM.edgesMap.get(edgeIDList[0])
  sourceNode ← firstEdge.getSource()
  targetNode ← firstEdge.getTarget()
  metaEdge ← createMetaEdge(edgeIDList)
  invisibleEodeList ← []
  for each edgeID ∈ edgeIDList do
    if edgeID ∉ edgeToMetaEdgeMap.keys then
      invisibleEodeList ← invisibleEodeList ∪ edgeID
    removeEdgeFromGraph(edgeID)
  for each edgeID ∈ invisibleEodeList do
    edge ← vGM.edgesMap.get(edgeID)
    edge.setIsVisible(false)
```

**Algorithm 3**

```
function ExpandNode(Node v, boolean isRecursive)
  GM ← mainGraphManager
  vGM ← visibleGraphManager
  nodeInInvisible ← GM.nodesMap.get(v.ID)
  vGM.addGraph(emptyGraph, v)
  v.child ← nodeInInvisible.child
  nodeInInvisible.child ← v.child
  childrenNodes ← getDescendantsInOrder(v)
                        ▷ this function returns a sorted list of descendants.
  markedChildren ← []
  for each child ∈ childrenNodes do
    if (!child.isFiltered ∧ !child.isHidden ∧ !child.isCollapsed)∨
       (child.isFiltered ∧ !child.isHidden ∧ child.isCollapsed ∧
       isRecursive) then

      reportedEdges ← moveNodeToVisible(v)
                        ▷ report all edges (to be removed) and meta edges
                          (to be removed & to be added) to Cytoscapejs side
                          using global variables
      if child.child then
        expandNode(child, isRecursive)
    else if !child.isFiltered ∧ !child.isHidden ∧ !isRecursive ∧
       child.isCollapsed then
      reportedEdges ← moveNodeToVisible(v)
                        ▷ report child node, all edges and meta edges to Cy-
                          toscapejs side using global variables.
      markedChildren ← markedChildren ∪ child
  for each n ∈ markedChildren do
    for each e ∈ n.edges do
      if e ∈ GM.edgeToMetaEdgeMap.keys then
        T ← getTopMetaEdge(e)
        if T.source ∉ v.child.edges ∨ T.target ∉ v.child.edges then
          if T.source.owner = T.target.owner then
            T.source.owner.addEdge(T, T.source, T.target)
          else
            GM.addInterGraphEdge(T, T.source, T.target)
                        ▷ report T (to be added) to Cytoscapejs side using
                          global varriables
```

**Algorithm 4**

```
function ExpandEdges(List edgeIDList, boolean isRecursive)
  GM ← mainGraphManager
  vGM ← visibleGraphManager
  for each ID ∈ edgeIDList do
    M ← vGM.edgeToMetaEdgeMap.get(ID)
    S ← M.source
    T ← M.target
    for each e ∈ M.orignalEdgesList do
      if e.ID ∈ vGM.metaEdgeMap.keys then
        if S.owner = T.owner then
          S.owner.addEdge(e, S, T)
        else
          vGM.addInterGraphEdge(e, S, T)
        if isRecursive then expandEdges(e.ID, isRecursive)
      else
        eINV ← GM.edgesMap..get(e.ID)
        if !eINV.isFiltered ∧ !eINV.isHidden then
          eINV.setIsVisible(true)
          if S.owner = T.owner then
            S.owner.addEdge(e, S, T)
          else
            vGM.addInterGraphEdge(e, S, T)
```

Figure 3.6, Figure 3.7, and Figure 3.8 present scenarios to illustrate how the CMGM graph models change as a combination of various complexity management operations are applied. Algorithm 1 - 4 are used to execute respective collapse and expand operations in the above given scenarios.

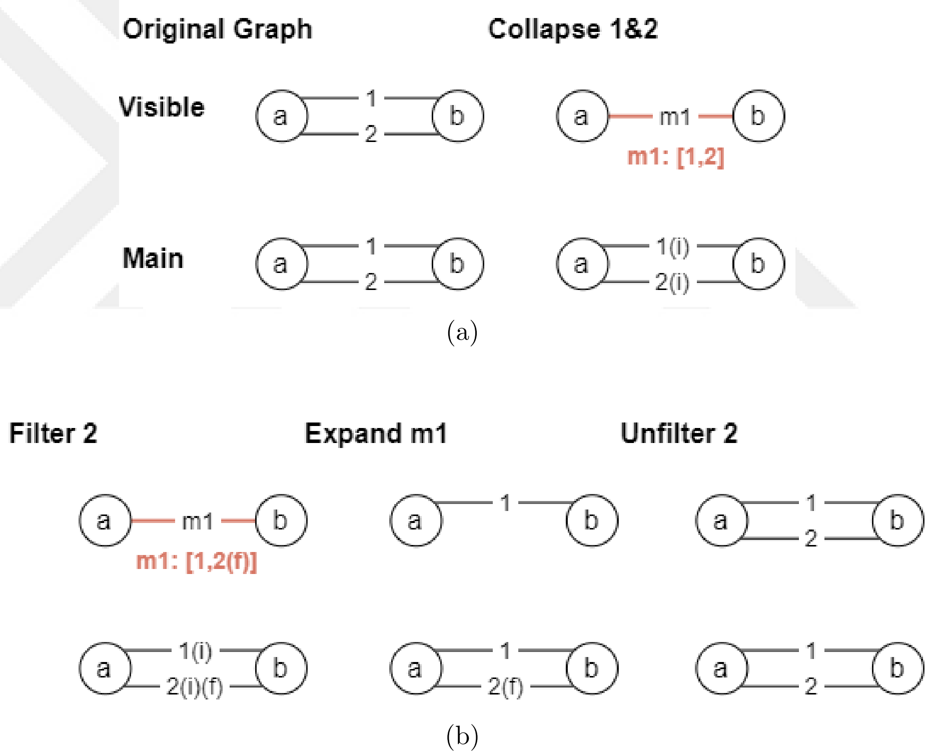



Figure 3.6: A scenario illustrating a sequence of edge collapse & filter/unfilter operations to show how visible and main graphs change.

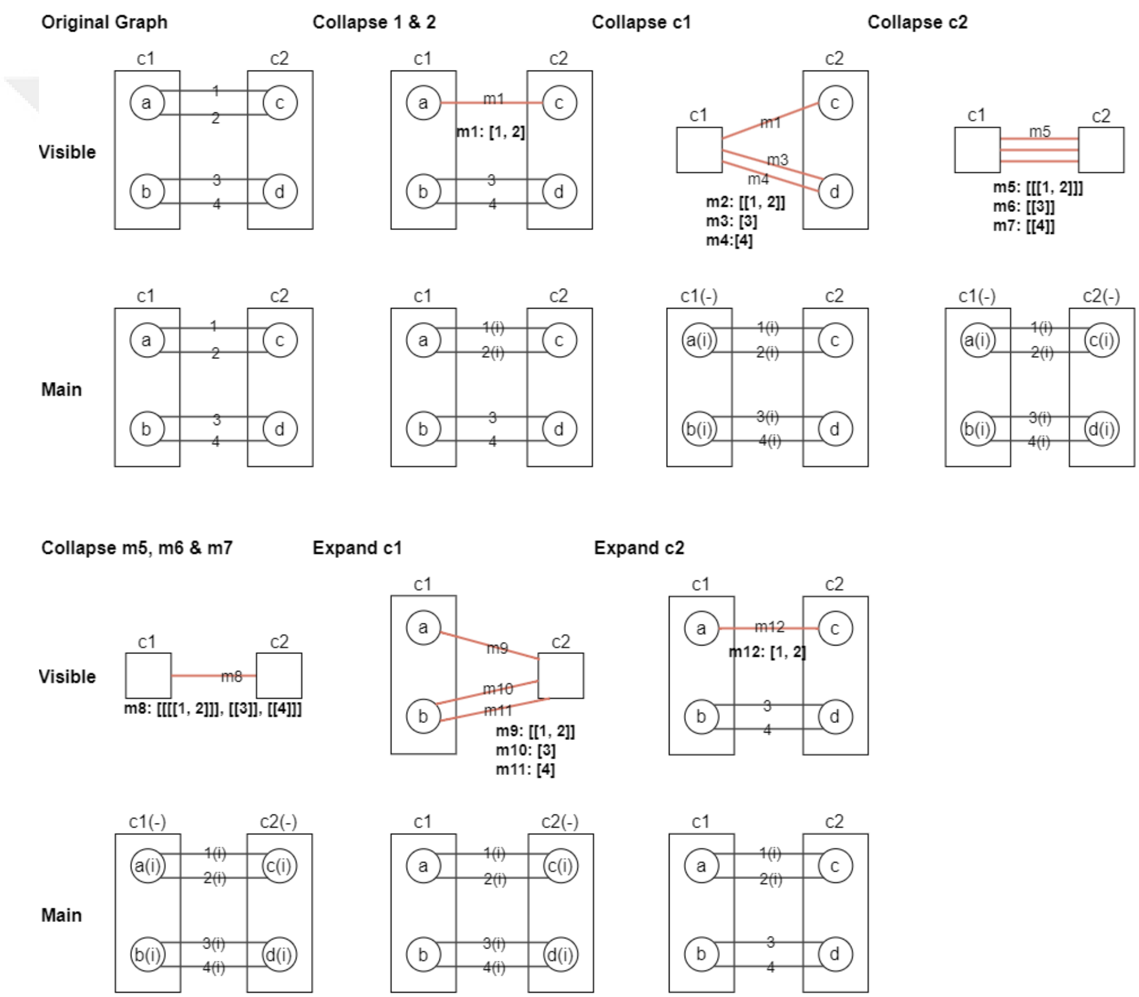



Figure 3.7: A scenario illustrating a sequence of node and edge collapse/expand operations to show how visible and main graphs change.

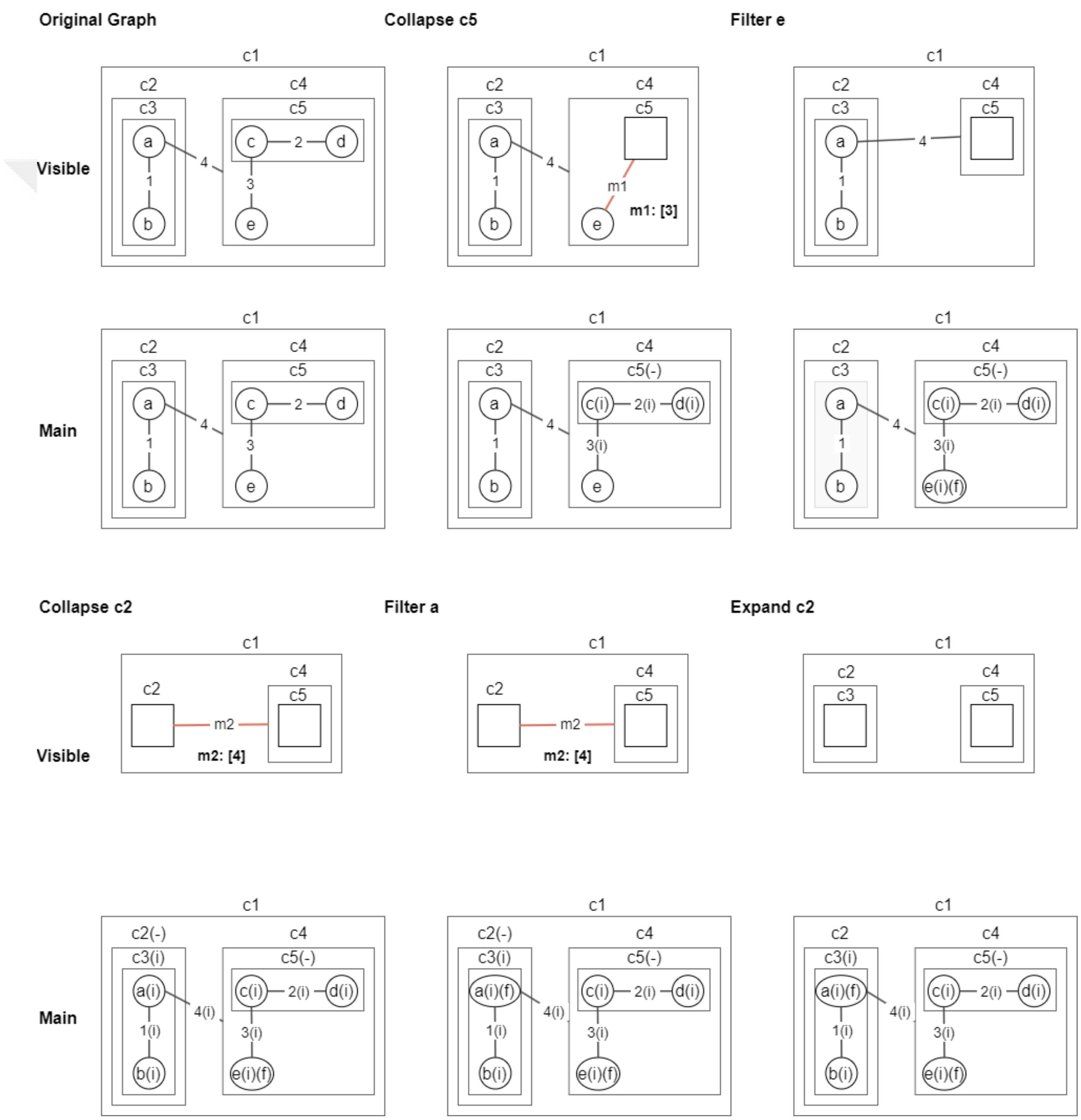



Figure 3.8: A scenario illustrating a sequence of node collapse/expand and filter operations to show how visible and main graphs change.

## 3.4 Adjusting Layout

To facilitate effective analysis, the protection of the user's mental map is essential when performing any complexity management operations. Layout adjustments are performed after each complexity management operation to make sure the overall graph is not disturbed too much and the user's mental map remains intact. Additionally, the layout algorithm employed post complexity management operations should yield results aligned with the *base layout* algorithm used by the rest of the application. When choosing a base algorithm, it is crucial to take into account its capability to handle compound structures and incremental layout, as well as its flexibility to adjust the layout after complexity management tasks. For our study, we opted for fCoSE [11] as our base layout algorithm, as it fulfills the above-mentioned criteria for base layout.

Most complexity management operations may be handled using the base layout algorithm in incremental mode, where the current respective positions of graph elements are maintained (in the case of force-directed layout, this simply corresponds to starting the iterative algorithm from the current node positions). Adjustment of the layout after a collapse operation is exemplified in Figure. 3.9. However, especially the expand node operation requires a specialized incremental layout algorithm to ensure newly introduced, possibly large content, does not overlap and hence tangle with surrounding graph elements, tarnishing the existing nice layout. An expand operation in the middle of the nesting hierarchy of multiply nested compound structures makes things even more difficult.

After expand the operation, the layout needs to be able to fit the expanding compound in its current position without disturbing the entire graph. The main idea of our layout algorithm is to make an approximate estimation of the size of the expanding compound after expansion and then clear that much space around the expanding compound node to make room for expansion. Then the expand operation is executed followed by a final incremental base layout (fCoSE in this case) to make sure the graph is compact and polished. Following is a detailed description of the entire process.

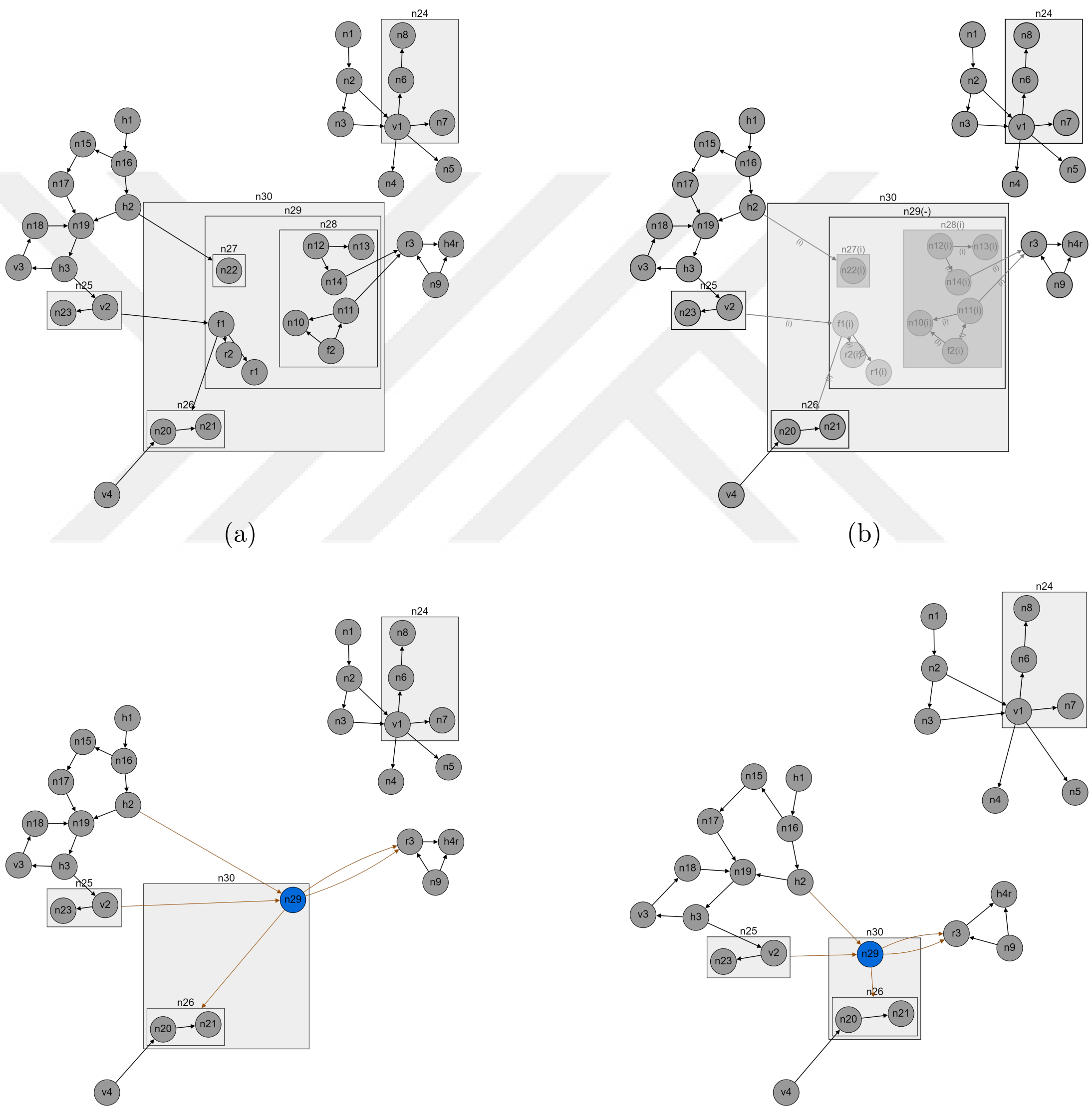



Figure 3.9: An illustration depicting layout adjustment on collapse operation. (a) Initial visible graph (b) Main graph (c) Visible graph after collapsing $n29$ node, and (d) Final visible graph after applying incremental base layout.

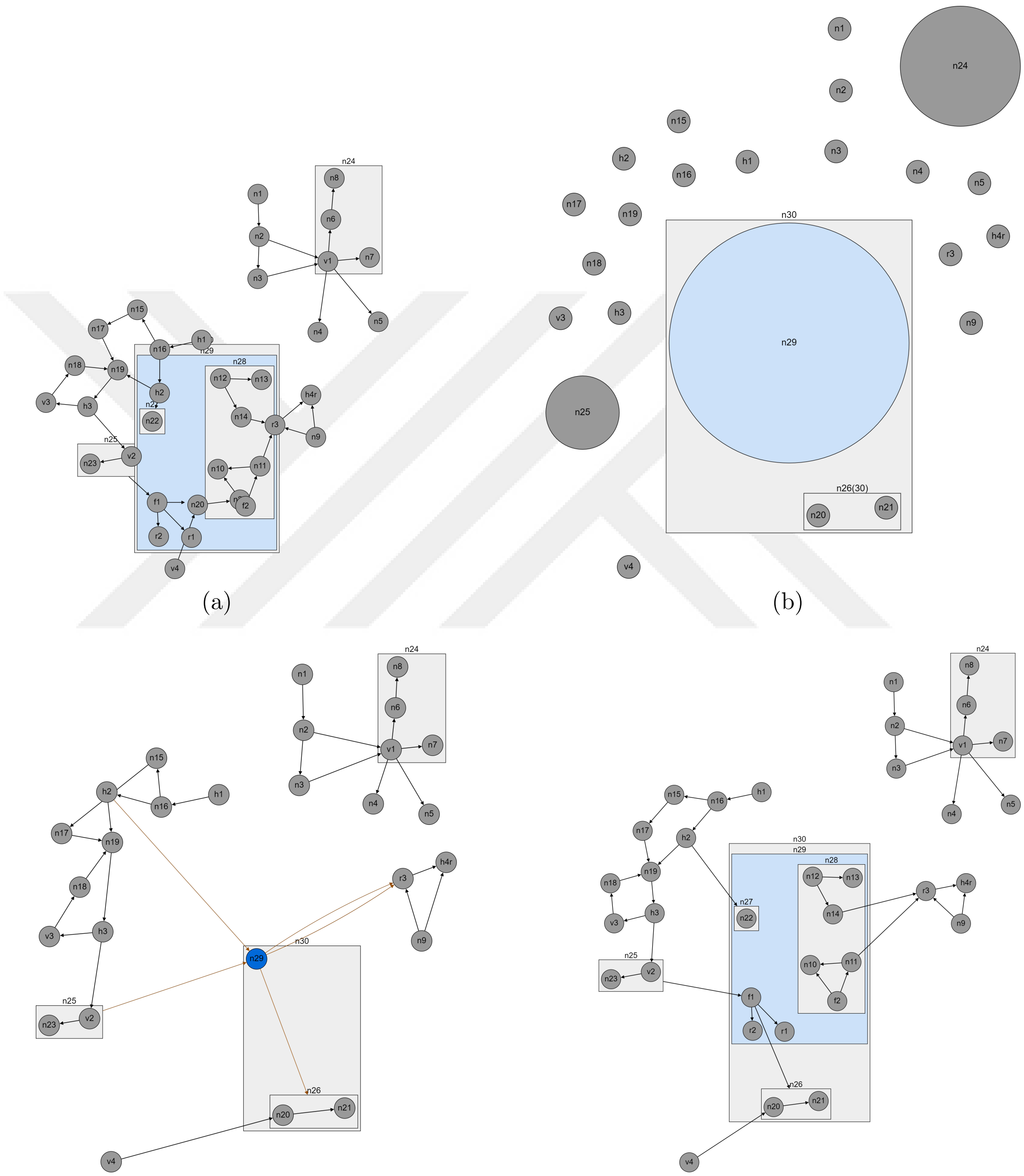



Figure 3.10: An illustration depicting layout adjustment after an expand operation, continuing from Figure 3.9 (d). (a) Visible graph without any layout adjustment after expanding $n29$ (b) Proxy graph used to estimate the size of $n29$ after expand and incremental layout (c) Visible graph after graph components are repositioned based on the layout of the proxy graph, and (d) Final adjusted layout.

First, the algorithm begins by calculating the *expansion factor*, the approximated height or width (whichever is greater) of the square area covered by the expanding compound node after expansion. Simply put, it is an approximation of the space needed to accommodate the expanding compound node. In order to do that, a neighborhood graph is created having the expanding compound node with all its children nodes and edges, including the intergraph edges with their end nodes outside the expanding compound node. Then the base layout is applied to adjust the positions of all the nodes, and a bounding box is drawn around the compound node (Figure 3.12). Larger of height and width of said bounding box is the *expansion factor*.

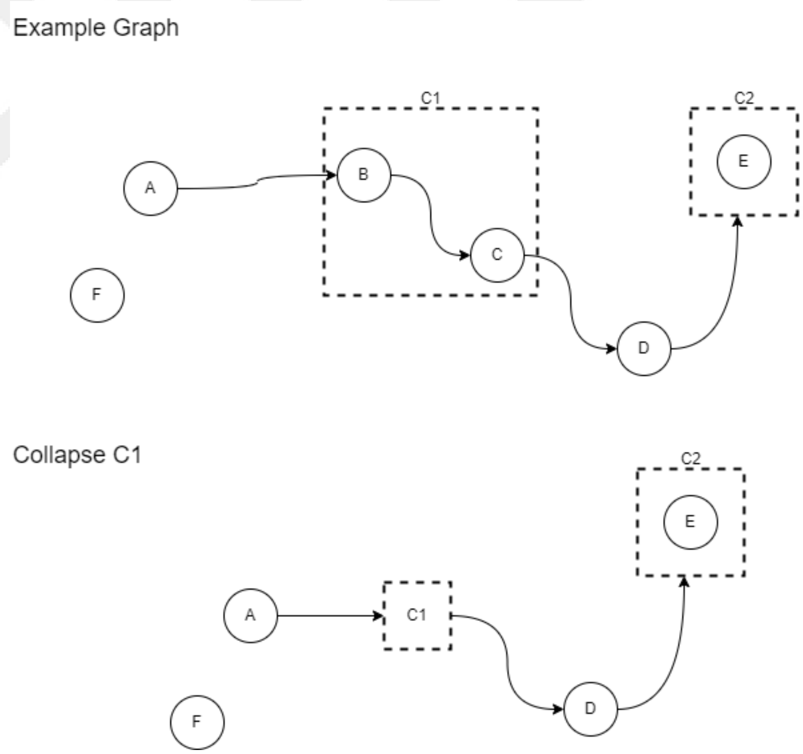



Figure 3.11: A sample compound graph before and after collapse operation.

After the calculation of the expansion factor, space around the expanding compound node needs to be cleared. For that, a proxy graph is created with nodes representing top-level nodes of the visible graph with the same sizes and same positions as in the visible graph (Figure 3.11 and Figure 3.13). The expansion factor is used as the size of expanding a compound node. Then the base layout

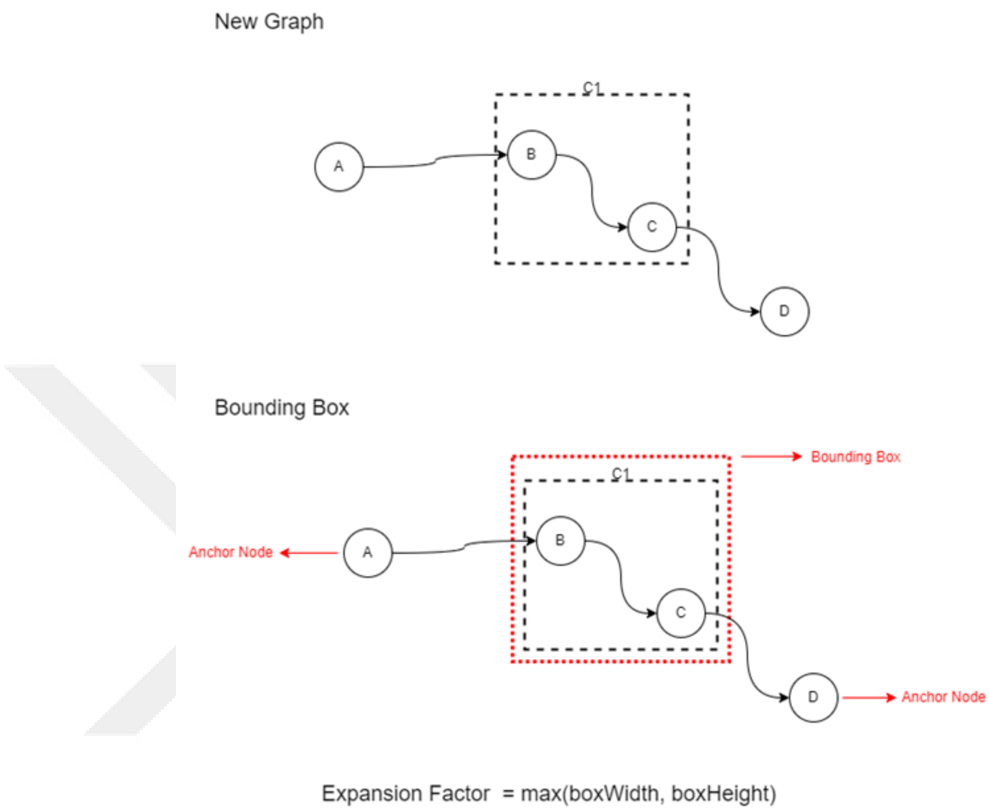



Figure 3.12: A neighborhood graph with focus node, anchor nodes & bounding box (continued in Figure 3.11).

is applied to this proxy graph in incremental mode, adjusting the layout according to the new size of expanding the compound node. Using the proxy graph, new positions for all nodes in the visible graph are calculated. To determine the movement of the compound node's children, a drift increment is calculated for each compound node in both the $x$ and $y$ directions, and then it is added to the original position of children nodes in the visible graph. This drift increment is obtained by measuring how much a compound node has shifted from its initial position to the newly calculated position in both the $x$ and $y$ directions. Subsequently, we apply these drift increments to move the children nodes accordingly (refer to Figure 3.13). Once all the nodes are moved to their new position after applying the drift increment, expand operation is executed and the base layout is applied.

The above-mentioned process assumes that the expanding compound node is

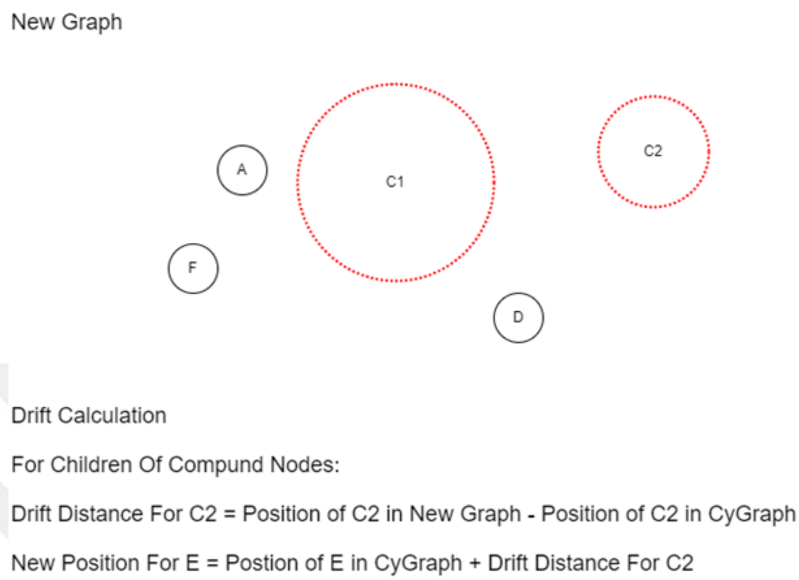



Figure 3.13: A proxy graph representation during expand operation (continued in Figure 3.11).

a top-level node in the visible graph, but in the compound graph with nested hierarchies, it is very much possible that the expanding compound node is not a top-level node. In such cases, the parent, siblings, and the hierarchical structure become a challenge when creating the second proxy graph and calculating the drift increment. If only the top-level node of the hierarchy of expanding compound nodes (let's call it the focus node) is represented in the proxy graph, then the drift increment will have no impact on the nodes surrounding the expanding compound node and there will be a huge node overlap after expansion which will change the entire layout when the final based layout is performed. If both the focus node and the expanding compound node are represented in the proxy graph, the drift increment calculated will place the focus node on one side, and expanding compound node on the other side, thus the drift increment will try to place the expanding compound node outside its hierarchy which will create node overlaps upon expansion and long inter graph edges which will pull the expanding compound node into the hierarchy moving the entire layout too much during the final base layout execution which will end of destroying user's mental map.

To avoid these problems, in such cases, the algorithm first identifies the focus

**Algorithm 5**

**function** $\textsc{ExpandNodeWithLayout}(\text{ Node } v)$
  $exp \leftarrow \textsc{CalculateExpansionFactor}(v)$
  $components \leftarrow \textsc{CreateComponents}(exp, v)$
  $\textsc{ExpandGraph}(componenets, v)$
  $\textsc{ExpandNode}(v)$

**Algorithm 6** Calcualte Expansion Factor
**Expected Runtime Complexity:** O(V + E), where V is the number of nodes and E is the number of edges in the graph.

**function** $\textsc{CalculateExpansionFactor}(\text{Node } v)$
  $GM \leftarrow mainGraphManager$
  $vGM \leftarrow visibleGraphManager$
  $descendants \leftarrow \textsc{getDescendantsInOrder}(v)$
  $cyLayout \leftarrow \textsc{cytoscape}(\{options\})$
    ▷ creating a new cytoscapejs graph
  $cyLayout.\textsc{add}(v)$
  $savedNodes \leftarrow []$
  **for each** $c \in descendants.compoundNodes$ **do**
    **if** $c.owner.parent \notin cyLayout.nodes$ **then**
      $cyLayout.\textsc{add}(c.owner.parent)$
    **else**
      $savedNodes \leftarrow savedNodes \cup c.owner.parent$
  **for each** $c \in savedNodes$ **do**
    $cyLayout.\textsc{add}(c)$
  **for each** $c \in descendants.simpleNodes$ **do**
    $cyLayout.\textsc{add}(c)$
  **for each** $e \in descendants.edges$ **do**
    **if** $e.source \notin cyLayout.nodes$ **then**
      $cyLayout.\textsc{add}(e.source)$
    **else if** $e.target \notin cyLayout.nodes$ **then**
      $cyLayout.\textsc{add}(e.target)$
    $cyLayout.\textsc{add}(e)$
  $cyLayout.\textsc{layout}(layoutOptions).\textsc{run}()$
  $expFactor \leftarrow cylayout.\textsc{get}(v).\textsc{boundingBox}()$
  **return** $expFactor$

**Algorithm 7** Create the proxy graph and uses an instance of ExpansionFactor class that stores the size and position of the expanding node.
**Expected Runtime Complexity:** O(V), where V is the number of nodes in the graph.

**function** CreateComponents(ExpansionFactor $exp$, Node $v$)
  $GM \leftarrow mainGraphManager$
  $vGM \leftarrow visibleGraphManager$
  $topParent \leftarrow GM.$getTopParent($v$)
  $componenets \leftarrow []$
  $cyLayout \leftarrow$ cytoscape($\{options\}$)
  **for each** $c \in cy.nodes$ **do**
    **if** $v \neq topParent \ \wedge \ v.parent = null$ **then**
      **if** $v.child = null$ **then**
        $selected \leftarrow v.$select()
      **else**
        $selected \leftarrow$ selectChildren($v$)
      $b \leftarrow selected.$boundingBox()
      $compoenents \leftarrow components \cup \{id : c.id, data : selected, pos : b.pos\}$
      $n \leftarrow cyLayout.$add($c$)
      $n.$position($b.pos$)
      $n.$size($c.width, c.height$)
      $cy.$unselectAll()
  **if** $v.parent = null$ **then**
    $n \leftarrow cyLayout.$add($v$)
    $n.$style($\{width : v.w, height : v.h, position : v.pos, color : Blue\}$)
  **else**
    $n \leftarrow cyLayout.$add($topParent$)
    $n.$position($topParent.position$)
    $n.$size($topParent.w, topParent.h$)
    $children \leftarrow$ selectChildren($topParent$)
    **for each**$c \in children$ **do**
      $b \leftarrow c.$boundingBox()
      **if** $c \neq v$ **then**
        **if** $c.$isChildless() **then**
          $compoenents \leftarrow components \cup \{id : c.id, data : selected, pos : b.pos\}$
          $n \leftarrow cyLayout.$add($c$)
          $n.$position($b.pos$)
          $n.$size($c.width, c.height$)
      **else**
        $n \leftarrow cyLayout.$add($c$)
        $n.$style($\{width : c.w, height : c.h, position : c.pos, color : Blue\}$)
    $cyLayout.$layout($layoutOptions$).run()

node in the main graph and then creates the proxy graph with all the top-level nodes with their respective sizes and positions except for the focus node. For the focus node, the algorithm adds it as a compound node to the proxy graph with all its children nodes including the expanding compound node with their respective sizes (except the expanding compound node, expansion factor is used as its size) and positions. Adding this hierarchy to the proxy graph affects the positions of siblings of the expanding compound node in the hierarchy according to the drift increment, creating empty space around it to accommodate expansion without disturbing the entire graph.

---

**Algorithm 8** Expand Graph
**Expected Runtime Complexity:** O(V), where V is the number of nodes in the graph.

---

**function** EXPANDGRAPH(List *components*, Node $v$)
  **for each** $c \in components$ **do**
    $newPos \leftarrow cyLayout.\text{GET}(c.ID).position$
    $driftFactor.x \leftarrow newPos.x - c.pos.x$
    $driftFactor.y \leftarrow newPos.y - c.pos.y$
    **for each** $n \in c.data$ **do**
      MOVECHILDREN($n, driftFactor, v$)

---

Figure 3.10 demonstrates an example scenario involving the expansion of a compound node within the hierarchy. The graphs shown illustrate how the algorithm creates the proxy graph and adjusts the layout to accommodate the expansion and create sufficient space for the newly added content. Algorithm 5 is a function to expand a compound node with a well-maintained layout. First, it calculates the estimated size of expanding node using Algorithm 6, then creates components in a proxy graph and estimates the drift increments for all the nodes in the main graph and moves them to make room using Algorithm 7 and Algorithm 8 respectively. Detailed explanations of the inner workings of this function are already given above.

# Chapter 4

# Implementation and Evaluation

CMGV is a standalone framework for graph complexity management with a unified model (CMGM), scripted and developed in a JavaScript framework (see details in Section 3.1 and Section 3.2). It is designed to be used with any graph rendering library via a CMGM API for that rendering library. For the purpose of demonstration and evaluation, we are using Cytoscape.js as a rendering library, and an API is created to communicate between Cytoscape.js rendering library and the CMGM. The structure and functioning of API have been discussed in detail in Section 3.1.2.

The focus of this chapter is mainly on the implementation of demo and test scripts for integrity and runtime testing of the framework. For demonstration purposes, we have implemented a demo interface (see 4.1) with a tool panel on the left side having a button to perform the following operations:

- File Operations.
  - Open (To load a graph from JSON, GRAPHML, or SIF file).
  - Save (To save the graph to JSON, GRAPHML, or SIF file or to save a snapshot of the graph in PNG format).
  - Open Sample (To load sample graph).

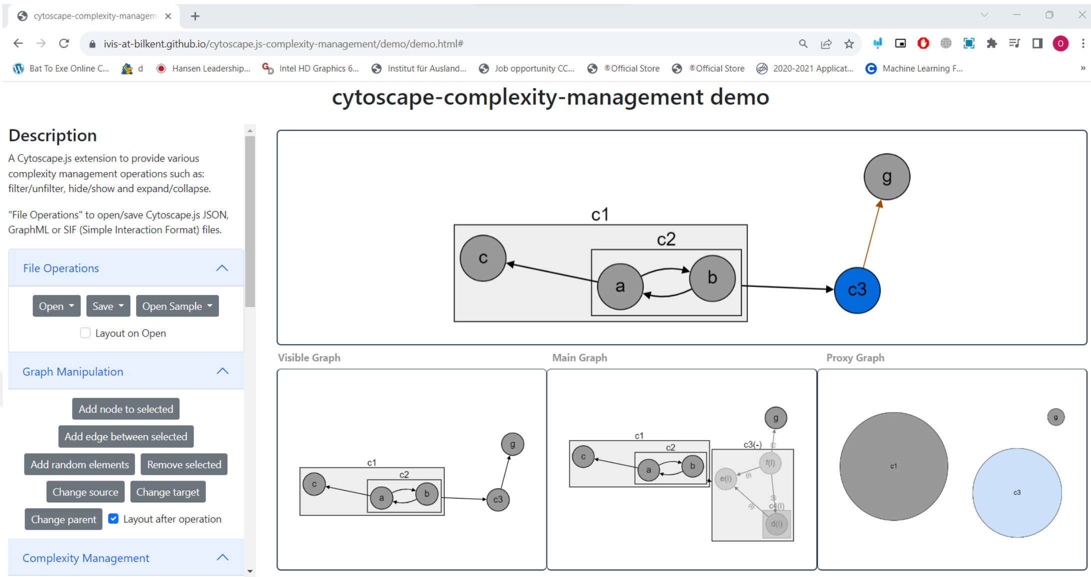



Figure 4.1: Illustrating the demo page to try out the complexity management operations interactively.

- Graph Manipulation Operations.
  - Add node (To add a node to a selected graph level).
  - Add edge (To add an edge between a selected node pair).
  - Add random elements (To add a randomly generated graph to the selected graph level).
  - Remove (To remove selected graph element).
  - Change source (To change the source of selected edge to selected node).
  - Change target (To change the target of selected edge to selected node).
  - Change parent (To change parent of first selected node with second selected node).
- Complexity Management Operations
  - Filter/Unfilter nodes (To filter nodes based on node weight value).
  - Filter/Unfilter edges (To filter edges based on edge weight value).
  - Hide selected (To hide selected graph elements).

- Show hidden neighbors (To show the hidden neighbors of a selected node).
- Show all (To show all hidden graph elements).
- Collapse selected nodes (To collapse selected nodes).
- Expand selected nodes (To expand selected nodes).
- Collapse selected edges (To collapse selected edges).
- Expand selected edges (To expand selected edges).
- Collapse edges between selected nodes (To collapse edges between all selected nodes).
- Expand edges between selected nodes (To expand edges between all selected nodes).
- Collapse all nodes (To collapse all nodes).
- Expand all nodes (To expand all nodes).

• Graph View options.

  - Hide/Show Graph element's label and weight.
  - Change position of node labels.

The demo also has four graph view windows to visualize each graph's current state. The window on the top is for the rendering library side graph, which end users of this framework will see. There are three more windows below that which are called *visible graph*, *main graph*, and *proxy graph*. The visible graph shows the graph with only the visible elements; it will always be the same as the rendering library side graph. The main graph shows the graph with all the elements visible and invisible with clear labeling and styling differences, a filtered element is marked with ($f$), a hidden element is marked with ($h$), an invisible element is marked with ($i$) and a collapsed node is with (-) next it is the original label. The proxy graph, o the other hand, shows the graph created for graph expansion layout during expand node operation (See details in Section 3.4). Each node represents a top-level node in the graph, and the expanding node is marked with the color blue. In case the expanding node is not a top-level node, then the complete hierarchy of the top-level node of the expanding node is plotted.

## 4.1 Overall Performance Evaluation

We conducted a series of experiments to assess performance, during which we measured commonly used performance metrics. Our testing was performed on a standard computer equipped with an Intel i7-11700 2.50GHz CPU and 16 GB of RAM.

For testing, we have implemented an automated script that takes scenarios (i.e., a series of complexity management operations) and executes each operation. For integrity testing of the framework's ability to consecutively execute various complexity management operations, we are using a complete circle scenario, which performs a sequence of various complexity management operations (see Scenario 6.1.1). The script executes the designated scenario and then compares the main graph to the visible graph. This comparison involves checking the nodes, their incoming and outgoing edges, as well as the total degree of each node. Since the topology of the main graph contains all graph elements in their original state, comparing node and edge count along with node degrees between the main graph and visible graph is enough to assess the integrity of the algorithm. To further assess the integrity of each individual complexity management operation and other auxiliary operations, we also manually performed identical operations and cross-referenced the expected graphs. Results showed that the framework is working as expected for all intended operations. To test the limits of the framework, the automated script is programmed to perform the above-mentioned steps multiple times and, each time, adds a random neighborhood at a random node of the graph, which increases the size and complexity of the graph.

For runtime testing, the automated script executes the operations from the selected scenario with $x$ number of operations and records the execution time for the scenario $t$. Then, it reloads the graph and repeats the process $u$ times. Finally, it calculates the average time taken per operation $T$ by following the equation.

$$T = \frac{\sum_u t_u}{x * u} \tag{4.1}$$

We have constructed two scenarios (Scenario 6.1.2 & Scenario 6.1.3) with 10 and 15 operations, respectively for automated testing. Both scenarios include a combination of various complexity management operations offered by the framework. The latter scenario includes *adding a random neighborhood* operation, which increases the size of the graph by randomly adding new nodes and edges. We have recorded runtimes by running the above-mentioned scenarios on six different randomly generated compound graphs of various sizes ranging from 188 to 1020 nodes from [11] compound graph dataset. The compound graph dataset was generated as follows. Initially, 81 graphs from the Rome dataset [19] were randomly chosen. Which included, 60 small-size graphs containing 10 to 200 nodes, 21 medium-sized with 250 to 5000 nodes, and their graph density (d(G)) was no more than 3. Next, they used the Markov Clustering Algorithm [20] to identify clusters within the graph. Subsequently, the nodes within each cluster were grouped together into a single compound node. Through several iterations of this process, variable-depth compound graphs were generated. Figure 4.2 shows a line chart of the average run time per operation vs graph size (number of nodes) as described in the previous paragraph. The average runtime per operation increases linearly with respect to the graph's size. This is a predictable outcome, as all fundamental complexity management actions exhibit linear expected runtimes, as indicated by asymptotic runtime analysis (see Section 3.3). It can be seen that the second scenario has a slightly higher average runtime per operation. As the average runtime per operation is the average run time over multiple iterations, each iteration added multiple random neighborhoods to the graph which increased the size, thus causing a slightly higher average runtime. Overall average runtime relative to the size of graphs is within an acceptable range for interactive applications.

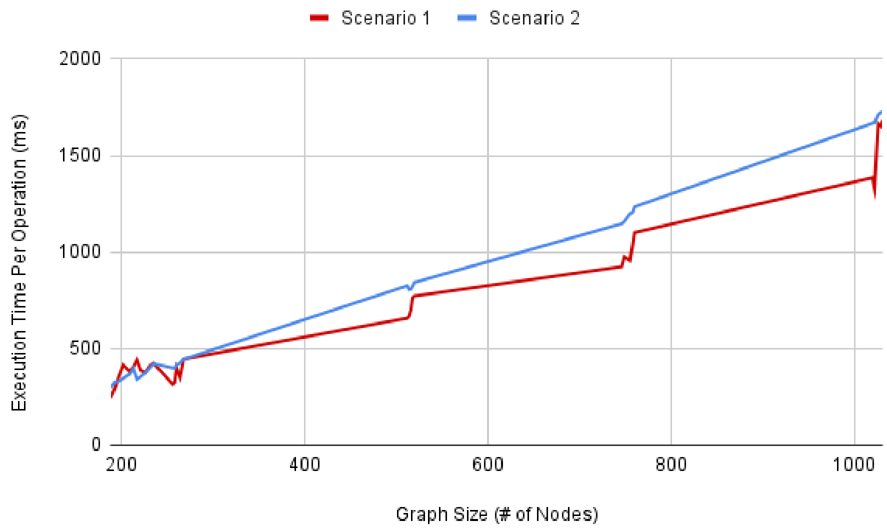



Figure 4.2: Comparing Estimated Run Time per Operation with Graph Size for Two Scenarios.

## 4.2 Expand Layout Algorithm Evaluation

To test our new layout algorithm, we are using various commonly used metrics for performance measurement and efficiency, such as the number of edge crossings, number of node overlaps, average edge length, total edge length, the area covered, and total edge length. We are using *Cytoscape.js-layvo* extension [18] for measurements of values of metrics. We are recording initial values of all above-mentioned metrics for six graphs of different sizes ranging from 188 to 1020 nodes laid out by *fCoSE algorithm* [11]. Then we execute a sequential collapse and expansion operation on a randomly selected compound node. The resulting graph, after expansion, is arranged using our novel layout algorithm, and we collect measurements for all metrics. This process is repeated five times for each graph, and the average value for each metric is calculated. Figure 4.3 compares the initial and average post-layout edge crossings. It is evident that our new layout algorithm achieves a lower number of edge crossings compared to the initial value for each graph. On average, a 23.77% reduction in the number of edge-crossing can be observed. Figure 4.4 compares the initial and average post-layout node overlaps. A slight reduction in node overlaps can be observed compared to the initial value for each graph. On average, a 5.87% reduction in the number of node overlaps can be observed. Since the edges are not involved during proxy graph creation and graph expansion, the nodes surrounding the expanding compound node experience significant displacement to accommodate the expansion, and incremental layout makes improvements in that area. This decrease in edge crossings and node overlaps is expected after successive incremental layouts by the new algorithm, where the results are improved by each execution.

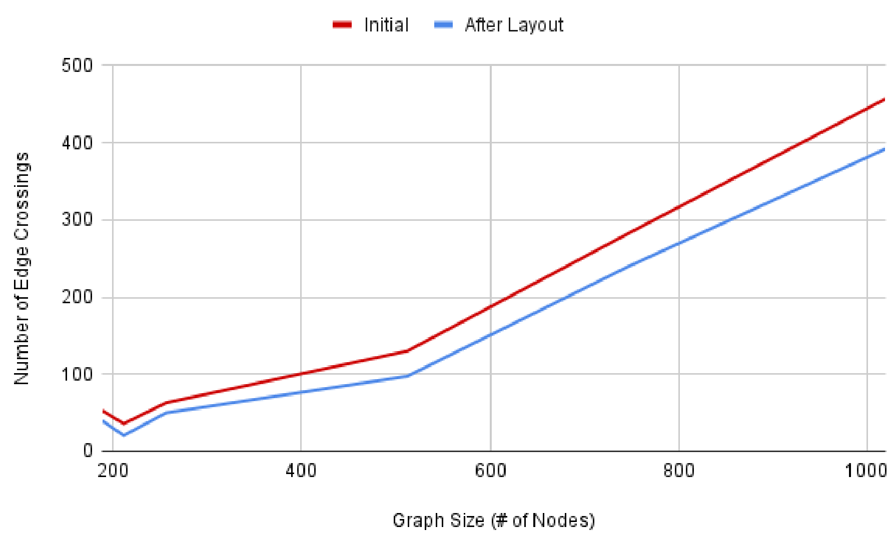



Figure 4.3: Comparing Initial and Post-Layout Edge Crossings with Graph Size.

Figures 4.5, 4.6, and 4.7 present a comparison between the initial and average post-layout values for average edge length, total edge length, and total area covered, respectively. These figures reveal a respective increase of 3.02%, 16.37%, and 3.47% in the post-layout values. Such a slight increment in all three metrics is anticipated. This increase is expected because the successive incremental layouts are reducing the number of edge crossings and node overlaps at the cost of expanding the overall graph and positioning nodes further apart from each other. A certain number of incremental changes and layouts in a graph is going to eventually result in poorer performance. Hence, the user might like to start a static (non-incremental layout) every now and then, to start an unbiased, from scratch layout for best results at the cost of losing their mental map.

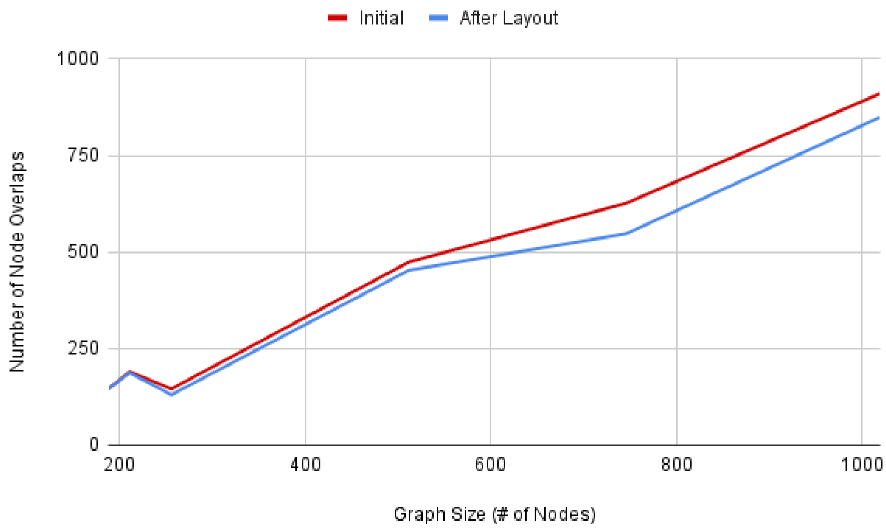



Figure 4.4: Comparing Initial and Post-Layout Node Overlaps with Graph Size.

| Graph Size (# of Nodes) | Neighborhood Proximity | Orthogonal Ordering |
|---|---|---|
| 188 | 0.96 | 0.81 |
| 211 | 0.96 | 0.79 |
| 256 | 0.97 | 0.82 |
| 512 | 0.97 | 0.80 |
| 746 | 0.98 | 0.79 |
| 1020 | 0.97 | 0.77 |

Table 4.1: Comparison of Graph Size (# of Nodes), Neighborhood Proximity, and Orthogonal Ordering for different graphs after consecutive collapse expand operation.

Table 4.1 represents the average values for neighborhood proximity and orthogonal ordering after performing consecutive collapse and expand operations for multiple iterations on different graphs. Neighborhood proximity is a metric that quantifies the distance or closeness between nodes in a graph layout. It measures how far apart or closely positioned the nodes are relative to each other in the visual representation of the graph. Orthogonal ordering measures the relative position of nodes in relation to each other within the layout, calculated as the polar angle $\theta$ (See Figure 2.4). By analyzing the comparison metrics, it can be seen that Neighborhood proximity values are very close to 1, indicating that edge length and node separation are not changing drastically. For orthogonal ordering,

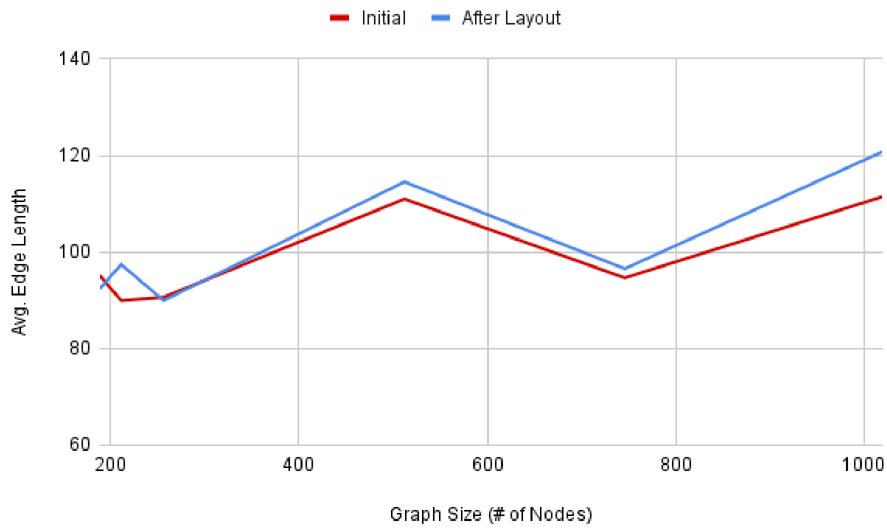



Figure 4.5: Comparing Initial and Post-Layout Average Edge Length with Graph Size.

values are not very close to 1 but not far off. This marginal change interim of orthogonal ordering is expected as expand layout algorithm does not take into account the edges when plotting the proxy graph, which hurts node positioning around the expanding node. Thus, it is likely that end nodes of edges around an expanding compound node do not get placed ideally, producing higher values of a polar angular difference $\Delta\theta$. During testing, it was observed that the bigger the expanding compound having numerous intergraph edges produces an orthogonal ordering value farther from 1.

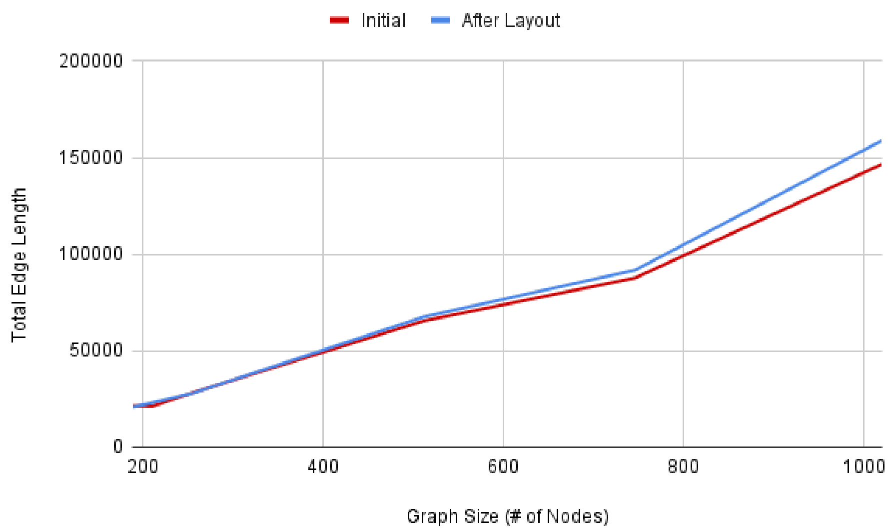



Figure 4.6: Comparing Initial and Post-Layout Total Edge Length with Graph Size.

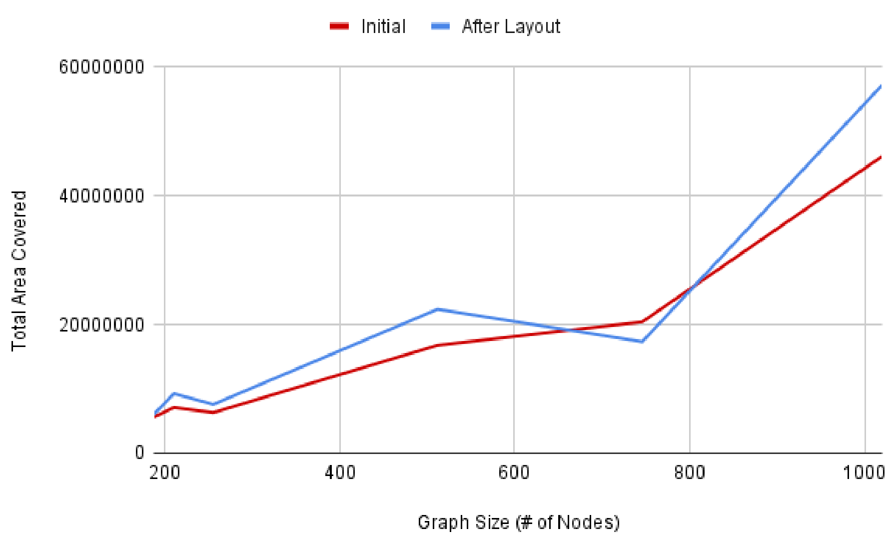



Figure 4.7: Comparing Initial and Post-Layout Total Area Covered with Graph Size.

# Chapter 5

# Conclusion and Future Work

In this thesis, we have presented the Complexity Management Graph Visualization framework (CMGV), a novel approach to addressing the challenges of analyzing and comprehending large-scale relational data presented as graphs. The vast amount of data produced in today's technology-driven world demands innovative solutions to help us analyze and understand it better. CMGV fills a critical gap by introducing a unified framework that integrates complexity management techniques, ensuring a coherent and intuitive representation of complex data.

Our work highlights the importance of managing the complexities associated with graph visualization, mainly when dealing with large-scale relational data. Existing methods often operate in isolation, potentially leading to confusion and inconsistencies. CMGV tackles this issue by harmonizing a range of complexity management operations, enhancing users' ability to interpret and derive meaningful insights from intricate data structures. A key highlight of CMGV is its preservation of the user's mental map throughout complexity management operations. The specialized layout algorithm, coupled with the intuitive graph complexity management model (CMGM), ensures that users can confidently analyze the data without losing their understanding of its relationships and patterns. Through our experiments, we have demonstrated the practical effectiveness of CMGV. By conducting operations on randomly generated graphs, we have shown that CMGV excels in executing consecutive graph complexity management tasks while maintaining the integrity of the underlying data structure. Another notable characteristic is that the framework works independently of specific rendering libraries and is compatible with different graph rendering libraries. This adaptability allows CMGV to cater to different visualization needs, ensuring its relevance in diverse domains.

As future work, additional complexity management and analysis operations can be added, such as semantic zooming (where expand and collapse operations on compound nodes could be performed with increasing and decreasing graph zoom level respectively, and the collapsed status is managed on the framework side). Similarly, the new layout algorithm introduced in this work focuses specifically on the expand operation; there are exciting possibilities for future work. This could involve integrating other element-adding complexity management operations like showing or unfiltering elements. One needs to identify the positions where unfiltered elements will be initially situated. These positions might be close to a visible node connected to an unfiltered graph node, or they could be the same as their previous placement before the filtering process. Subsequently, the current algorithm can expand the graph around these positions to incorporate these unfiltered elements. The current algorithm's success in preserving the user's mental map during the expand operation will serve as a foundation for further studies.

An open-source implementation of CMGV is available as a GitHub repository: https://github.com/iVis-at-Bilkent/cytoscape.js-complexity-management

An open-source implementation of CMGM is available as a GitHub repository: https://github.com/iVis-at-Bilkent/cmgm

A demo page is available to try out the layout interactively: https://ivis-at-bilkent.github.io/cytoscape.js-complexity-management/demo/demo.html.

# Bibliography

[1] H. Balci, M. C. Siper, N. Saleh, I. Safarli, L. Roy, M. Kilicarslan, R. Ozaydin, A. Mazein, C. Auffray, Ö. Babur, *et al.*, "Newt: a comprehensive web-based tool for viewing, constructing and analyzing biological maps.," *Bioinform.*, vol. 37, no. 10, pp. 1475–1477, 2021.

[2] W. J. Turkel, "Niche twitter followers." `https://williamjturkel.files.wordpress.com/2011/08/fig-5-niche-twitter-followers-20110421.jpg`. (accessed: July 8, 2022).

[3] AWS Perspective, Github repository, "Architecture diagram." `https://github.com/awslabs/aws-perspective/blob/main/docs/architecture-diagrams/arch-diagram.png`. (accessed: July 8, 2022).

[4] Cytoscape.js, "Using layouts." `https://blog.js.cytoscape.org/2020/05/11/layouts`. (accessed: July 23, 2022).

[5] U. Brandes, "Drawing on physical analogies," in *Drawing Graphs* (M. Kaufmann and D. Wagner, eds.), pp. 71–86, Springer, 2001.

[6] E. Bobek and B. Tversky, "Creating visual explanations improves learning," *Cognitive research: principles and implications*, vol. 1, pp. 1 – 14, 2016.

[7] A. C. A. C. U. Dogrusoz, E. Giral and E. Demir, "A layout algorithm for undirected compound graphs," *Cognitive research: principles and implications*, vol. 179, pp. 980 – 994, 2009.

[8] L. K. Klauske, *Effizientes Bearbeiten von Simulink Modellen mit Hilfe eines spezifisch angepassten Layoutalgorithmus.* PhD thesis, 2012.

[9] U. Dogrusoz, A. Karacelik, I. Safarli, H. Balci, L. Dervishi, and M. C. Siper, "Efficient methods and readily customizable libraries for managing complexity of large networks," *PLoS ONE*, vol. 13, p. e0197238, 2018.

[10] R. T. G. Di Battista, P. Eades and I. Tollis, "Graph drawing: Algorithms for the visualization of graphs.," *Upper Saddle River, NJ, USA: Prentice Hall*, vol. 1, 1998.

[11] H. Balci and U. Dogrusoz, "fcose: A fast compound graph layout algorithm with constraint support," *IEEE Transactions on Visualization and Computer Graphics*, vol. 28, no. 12, pp. 4582–4593, 2022.

[12] U. Dogrusoz and B. Genc, "A multi-graph approach to complexity management in interactive graph visualization," *Computers & Graphics*, vol. 30, p. 86–97, 2006.

[13] G. Di Battista, P. Eades, R. Tamassia, and I. Tollis, *Graph Drawing: Algorithms for the Visualization of Graphs.* Upper Saddle River, NJ, USA: Prentice Hall, 1 ed., 1998.

[14] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis, "Algorithms for drawing graphs: an annotated bibliography," *Computational Geometry-Theory and Application*, vol. 4, no. 5, pp. 235–282, 1994.

[15] P. Eades, "A heuristic for graph drawing," *Congressus numerantium*, vol. 42, pp. 149–160, 1984.

[16] T. Kamada and S. Kawai, "An algorithm for drawing general undirected graphs," *Inf. Process. Lett.*, vol. 31, no. 1, pp. 7–15, 1989.

[17] T. Fruchterman and E. Reingold, "Graph drawing by force-directed placement," *Software: Practice and Experience*, vol. 21, no. 11, p. 1129–1164, 1991.

[18] I. Safarli, "Methods and tools for synchronized visualization of evolving networks," 2018.

[19] S. S. Bridgeman, G. Di Battista, W. Didimo, G. Liotta, R. Tamassia, and L. Vismara, "Turn-regularity and optimal area drawings of orthogonal representations," *Computational Geometry*, vol. 16, no. 1, pp. 53–93, 2000.

[20] S. M. Van Dongen, *Graph clustering by flow simulation*. PhD thesis, University of Utrecht, May 2000.

# Chapter 6

# Appendix

## 6.1 Scenarios

There are six graphs chosen for testing and each graph has multiple levels of hierarchy, compound, and simple nodes with a balanced amount of edges with respect to the number of nodes. Each graph element has a randomly assigned weight ranging from 0-100. Following are the scenarios with sequences of complexity management operations to evaluate the integrity and performance of our framework. The scenarios can include a wide variety of actions like collapsing random compound nodes or edges between a pair of random nodes, hiding random nodes using the hide function, applying filter operation using weight assigned to node and edges, or showing all hidden elements or unfiltering all filtered elements. All scenarios are designed to be completely circular meaning, once all the operations are executed all graph elements will be visible.

### 6.1.1 Scenario 1

- collapse random compound node
- collapse edge between random nodes

- hide random nodes
- collapse all nodes
- expand all nodes
- filter node with weight less than 30
- filter edges with weight less than 50
- expand all edges.
- unfilter all

### 6.1.2 Scenario 2

- add edges
- hide random nodes
- collapse all nodes
- collapse all edges
- filter node with weight less than 30
- filter edges with weight less than 50
- expand all nodes
- expand all edges.
- Show all
- unfilter all

### 6.1.3 Scenario 3

- add edges
- collapse all edges
- filter node with weight less than 25
- collapse all nodes
- filter edges with weight less than 40
- hide random node
- expand all nodes
- hide random node
- filter edges with weight less than 85
- collapse all nodes
- unfilter all nodes
- unfilter all edges
- show all
- expand all edges
- expand all nodes