# HYSE: A SPRING EMBEDDER APPROACH FOR LAYOUT OF HYBRID GRAPHS

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
Hamza Islam
September 2023

HySE: A spring embedder approach for layout of hybrid graphs
By Hamza Islam
September 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.

Uğur Doğrusöz(Advisor)

İsmail Hakkı Toroslu

Eray Tüzün

Approved for the Graduate School of Engineering and Science:

Orhan Arıkan
Director of the Graduate School

# ABSTRACT

## HYSE: A SPRING EMBEDDER APPROACH FOR LAYOUT OF HYBRID GRAPHS

Hamza Islam
M.S. in Computer Engineering
Advisor: Uğur Doğrusöz
September 2023

In recent times, the growth of data has been exponential, making the visual analysis of relational data progressively complex. Presenting such data in a visually appealing manner can help simplify the analysis process. Hybrid graphs, comprising a central directed or hierarchical part and interconnected undirected components, offer a practical structure for representing relational data with varying levels of abstraction while managing its complexity. To comprehend the relationships in data, discover insights, and get important patterns, a well-optimized graph layout for such graphs is needed. In response, we present HySE (Hybrid Spring Embedder), a novel graph layout algorithm tailored for hybrid graphs. HySE makes use of a holistic approach based on the popular spring embedder to achieve the aesthetics and quality of an optimized force-directed layout, not only on the undirected part of the graph but also on the hierarchy while maintaining the cohesion between both directed and undirected elements of the graph. The layout algorithm assumes the rank information of directed graph elements is already calculated with one of the popular approaches. Then, it finds appropriate initial positions and uses a force-directed layout technique to integrate the undirected parts into the layout, applying spring forces to model the edges, and repulsive electric forces for the nodes. Iteratively, HySE converges to an equilibrium state with minimized energy, resulting in visually pleasing and interpretable layouts for intricate hybrid graphs.

Experiments performed on graphs, generated randomly through a well-designed process, validate that HySE performs as well as the state-of-the-art algorithms in terms of quality. It also matches the speed of well-established algorithms as well in small-to-medium-sized graphs.

*Keywords:* Information visualization, graph visualization, graph layout, graph

drawing, hybrid graphs, visual graph analysis, graph visualization software.

# ÖZET

# HYSE: HİBRİT ÇİZGE YERLEŞİMİ İÇİN YAY BAZLI YAKLAŞIM

Hamza Islam
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Danışmanı: Uğur Doğrusöz
Eylül 2023

Son zamanlarda verilerin büyümesi katlanarak arttı ve ilişkisel verilerin görsel analizini giderek karmaşık hale getirdi. Bu tür verileri görsel olarak çekici bir şekilde sunmak, analiz sürecini basitleştirmeye yardımcı olabilir. Merkezi yönlendirilmiş veya hiyerarşik bir parça ve birbirine bağlı yönlendirilmemiş bileşenlerden oluşan hibrit çizgeler, karmaşıklığını yönetirken değişen soyutlama düzeylerine sahip ilişkisel verileri temsil etmek için pratik bir yapı sunar. Verilerdeki ilişkileri anlamak, içgörüleri keşfetmek ve önemli kalıpları elde etmek için bu tür çizgelere yönelik iyi optimize edilmiş bir çizge düzenine ihtiyaç vardır. Buna yanıt olarak, hibrit çizgeler için özel olarak tasarlanmış yeni bir çizge düzeni algoritması olan HySE'yi (Hybrid Spring Embedder) sunuyoruz. HySE, hem yönlendirilmiş hem de yönlendirilmemiş arasındaki uyumu korurken, yalnızca çizgenin yönlendirilmemiş kısmında değil, aynı zamanda hiyerarşide de optimize edilmiş kuvvet yönlendirmeli düzenin estetiğini ve kalitesini elde etmek için popüler yay yerleştirmeyi temel alan bütünsel bir yaklaşımdan yararlanır. Yerleştirme algoritması, yönlendirilmiş çizge elemanlarının sıralama bilgilerinin, popüler yaklaşımlardan biriyle önceden hesaplandığını varsayar. Daha sonra uygun başlangıç konumlarını bulur ve yönlendirilmemiş parçaları yerleşime entegre etmek için kuvvet yönlendirmeli yerleşim tekniğini kullanır, kenarları modellemek için yay kuvvetlerini ve düğümler için itici elektrik kuvvetlerini uygular. Yinelemeli bir yöntem olan HySE, enerjinin en aza indirildiği bir denge durumuna yakınsar, bu da karmaşık hibrit çizgeler için görsel olarak hoş ve yorumlanabilir düzenler sağlar.

İyi tasarlanmış bir süreçle rastgele oluşturulan bileşik hibrit çizgeler üzerinde gerçekleştirilen deneyler, HySE'nin kalite açısından en son teknoloji algoritmalar kadar iyi performans gösterdiğini doğrulamaktadır. Aynı zamanda küçük ve orta boyutlu çizgelerde de bilinen algoritmaların hızını yakalar.

*Anahtar sözcükler*: Bilgi görselleştirme, grafik görselleştirme, grafik düzeni, grafik çizimi, hibrit grafikler, görsel grafik analizi, grafik görselleştirme yazılımı.

# Acknowledgement

I'm profoundly thankful to Prof. Dr. Uğur Doğrusöz for his invaluable support during my MS studies. This was surely the best experience of my life and I could not have undertaken this journey without his guidance and assistance.

I extend my gratitude to Dr. Eray Tüzün and Prof. Dr. İsmail Hakkı Toroslu for serving on my thesis jury. I also acknowledge TÜBİTAK's financial support for project number 121E230, which made this thesis feasible.

Special thanks to Osama Zafar for being such a support and a really fun buddy during these 2 years. Thank you to all the lab mates Dr. Hasan, Eren, Selbi, and Sina for making it pleasant and enjoyable. It was such a memorable time spent with you.

I'm extremely grateful to have friends in my life who support me irrespective of the circumstances. I want to thank especially Khaliq Dad Lak, Faisal Latif, Shamoon, Ans, Humza Gulzar, Ali, Junaid, Muntazir, Kashif, and the rest of Manto Street from the bottom of my heart for being the source of comfort, love, and inspiration. I definitely would not be managing this well without you guys.

I also want to thank my fellows Fardan, Amna, Ahmed Mushtaq, AR, Humza, Mudassir, Atif, Haris, and Hassan who instilled in me with motivation to do good in my field and take up new challenges. Thanks to Mehran, Moiz, Talha Bin Zahid, Haseeb, Ahmad Salman, Saif, Arfat, Ussama Yousaf, and Armaghan for being such good school and childhood friends and for inspiring me to do good in my studies.

I would like to thank Bilkent and its communities as well for providing a delightful environment. I am extremely grateful to Hilal Khan, Talha, Asfand, Hamid, Tufail, Najeeb, and Badar for looking after me when I got my yearly injuries. A bundle of thanks to the ones who played sports, the same names as above and Asad Shah, Mohsin, Ilyas, Imran, Waqar, Muzaffar, Shakir, Nasheet, Ahsan, Roshaan, Mohid, Rashid, Saad, Usama, and made these 2 years exciting.

Anyone who helped me, taught me, inspired me, motivated me, I am grateful to you.

In the end, I would like to express my deepest gratitude to the most important ones, my parents Muhammad Islam and Safina Islam, my sister Saman, and my brother Rouf for their continued love and support throughout my life.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Motivation

In the recent history of mankind, we have been generating an amount of data that we have never produced before. Especially with the incorporation of electronic devices and gadgets in our daily lives, we have been participating in this exponential growth of data even unknowingly. Researchers and scientists use this humongous amount of data to gain insights, derive patterns, and understand the relations between different entities. Data presented in a visual manner makes it simple and easy to understand the relations as compared to other forms of presentations such as descriptive and tabular form [10]. Here, information visualization takes charge of handling this task. *Information visualization* is the practice of presenting data in a useful and visually appealing manner which helps the user understand it relatively easier than the other forms stated above. Information visualization can help present the data in different forms: charts, plots, diagrams, graphs, etc. A huge amount of our real-world data is relational in nature. Graphs can be used to visualize the data if there are relationships between the data entities, where nodes are used to denote objects and edges denote the relation between them. This eventually gives a broad and extensive overview of the

large-scale data. *Graph visualization* nowadays is extremely helpful in bioinformatics, social network analysis, financial analysis, fraud prevention, architecture diagrams, etc (Figure 1.1 through Figure 1.4).

Data with relational attributes often exhibit hierarchical structures, cyclic patterns, clusters, etc. Hierarchical data, represented with a hierarchical graph is a subset of directed data, which can be used to describe the flow of any entity. For example, an organization's hierarchy might denote the power structure and describe the reporting pathway in that organization. Or in bioinformatics, the hierarchical graph can be used to denote the flow of the signal along a pathway as shown in Figure 1.3. This helps the readers and researchers to obtain a comprehensive understanding of the relations and the underlying complexities.

These graphs can sometimes contain additional information that is related to the entities denoted by nodes. This information is typically not hierarchical. If it were hierarchical in nature, then it would probably have been expressed as part of a hierarchy same as the other nodes. So we consider this data to be relational but not hierarchical. As a result, such additional information is attached to the hierarchical part making the graph hybrid, composed of two parts: hierarchical and non-hierarchical.

We can use these hybrid graphs to represent some complex relational data, but laying it out on screen can be a challenge. Placing the nodes and edges on the screen such that it is visually appealing and simple to comprehend, is a vital requirement of laying out a graph. The layout is an important aspect because a poor layout may confuse the user, make it difficult to understand, hide important patterns, or introduce irrelevant information. On top of this, it may take up 25% of the user's time just to make adjustments manually [11]. The metrics that are generally used to judge the quality of a graph layout and are commonly accepted are the number of edge crossings, average edge length, total area, etc. Graph layout algorithms are free to play around and optimize these metrics but in our case, we have the constraint of a hierarchy in our graph that we need to follow.

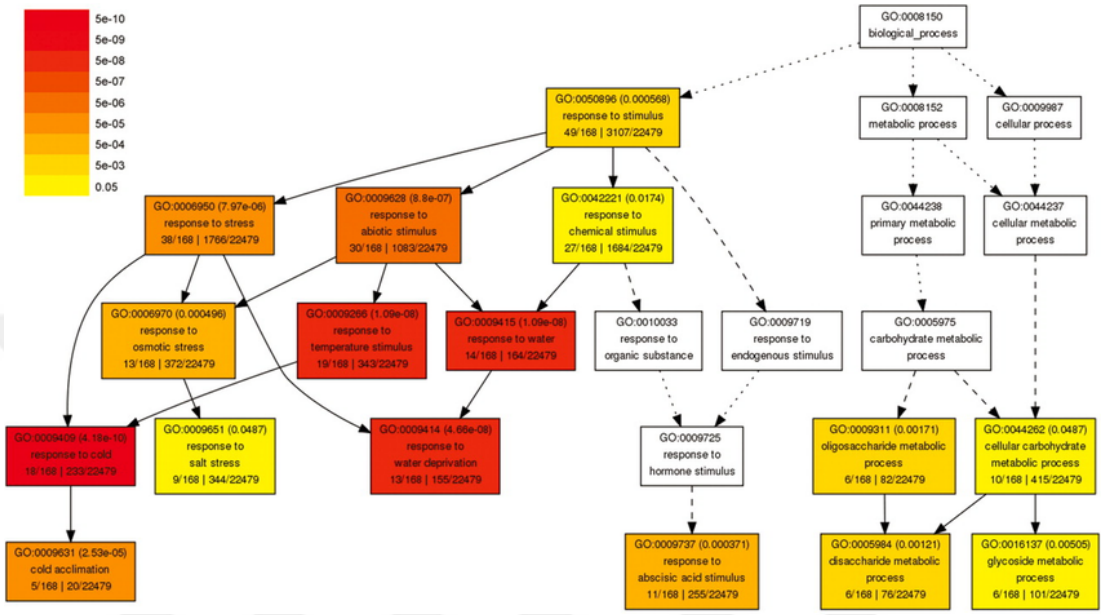



Figure 1.1: Hierarchical tree graph of overrepresented GO terms in biological process category [1]

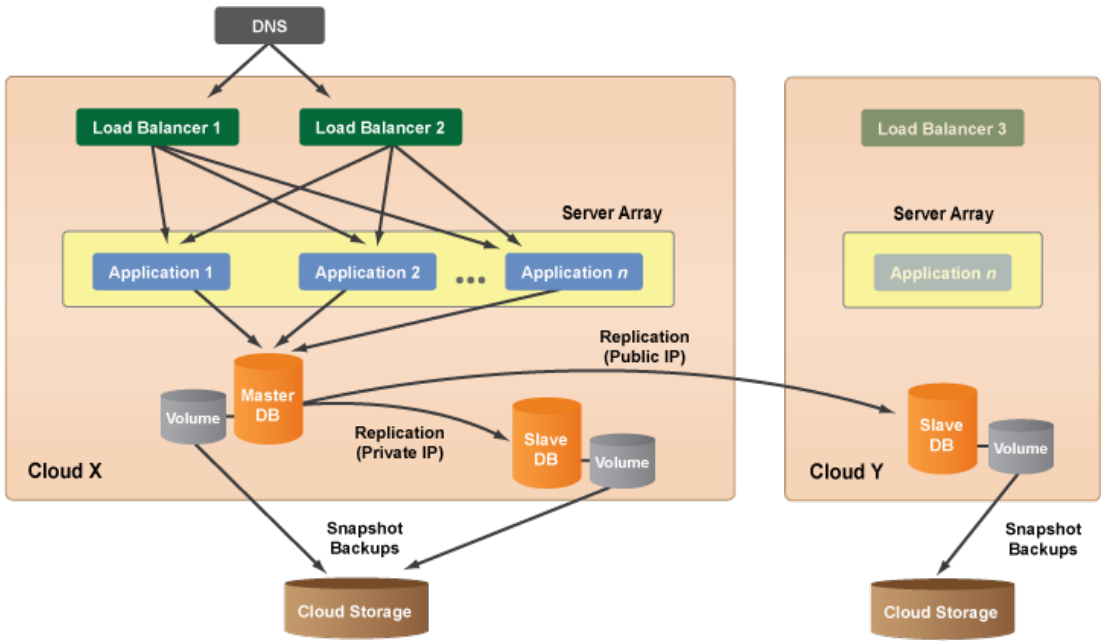



Figure 1.2: Hierarchical layout of cloud system architecture [2]

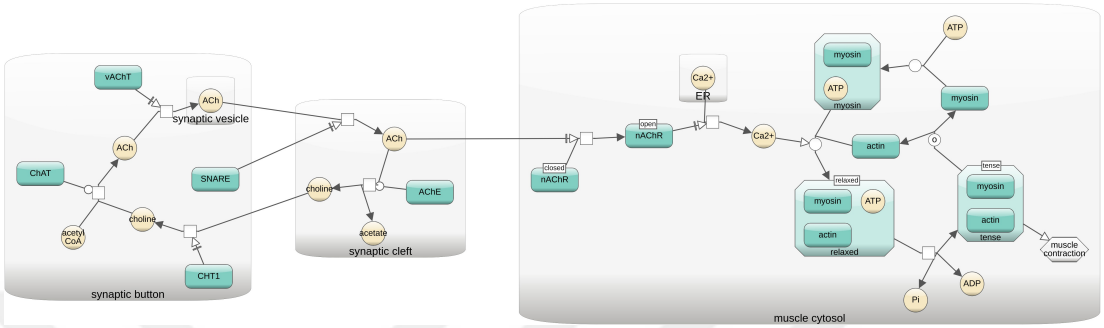



Figure 1.3: A sample biological pathway - neuronal muscle signaling [3]

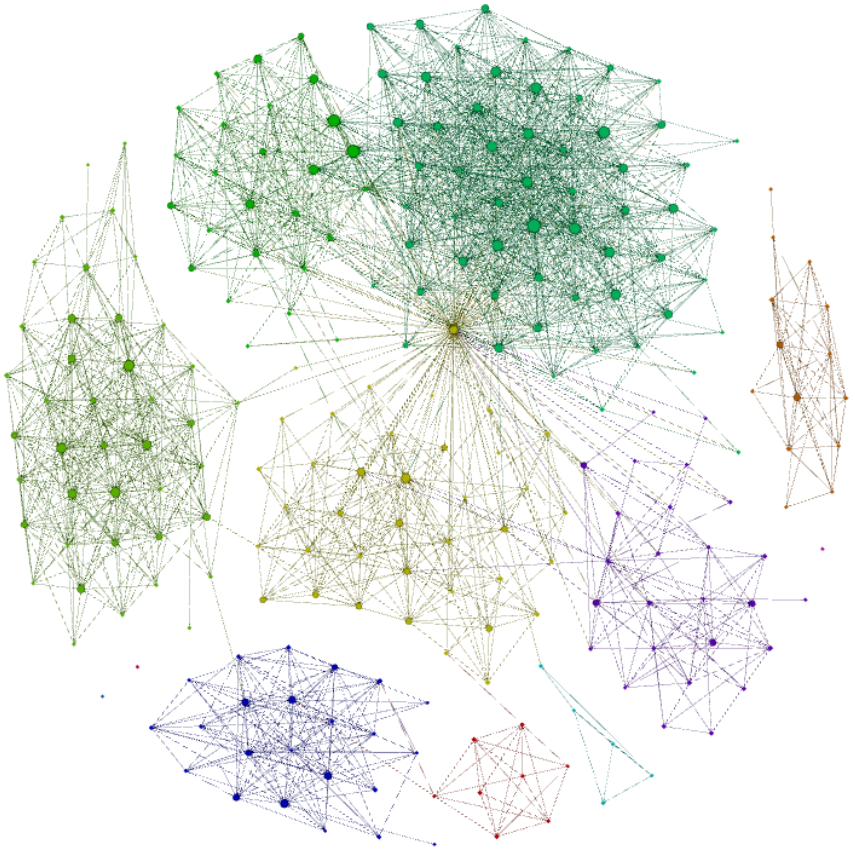


Figure 1.4: A sample social network of Twitter followers [4]

Although there is a considerable amount of work done both on directed/hierarchical and undirected graph layout [12] [13] [14], the work on the combination, hybrid graph layout is almost non-existent. Any existing work typically favors either part of the graph and the other part is somehow integrated into the algorithm in a non-ideal manner, sacrificing the quality of that secondary part. There can be workarounds using the existing algorithms that can handle these hybrid graphs but none of them can be a single solution that gives users the opportunity to layout the graph without any extensive coding or configuration. Besides no such work can support compound structures in the undirected part of the hybrid graph. In this thesis, we present HySE (Hybrid Spring Embedder), a novel graph layout algorithm that uses a "holistic" approach to layout hybrid graphs, which performs well in terms of commonly accepted graph layout quality metrics and execution time for small to medium-sized graphs.

## 1.2 Contribution

HySE (Hybrid Spring Embedder) is a graph layout algorithm that is designed to work with hybrid graphs having a central hierarchical part and typically loosely connected undirected parts. The algorithm makes use of a holistic spring embedder approach and places the nodes of both parts so that the graph is cohesive and good-looking aesthetically. The algorithm also works to minimize the edge crossings in the hierarchical part and correct the final positioning of the graph so that the graph is laid out neatly. To the best of knowledge, HySE is the first algorithm that provides a neatly laid out hierarchical part, which is as good as implementations of the well-known Sugiyama layout [8] in terms of edge crossings and better in terms of average edge length, without sacrificing the quality of the undirected components.

The subsequent sections of the thesis are organized in the following manner: In Chapter 2, essential concepts regarding graphs, layout algorithms, and mathematical concepts applied in this thesis are presented as well as an overview of

the related literature. Detailed information about the Hybrid Spring Embedder (HySE) is provided in Chapter 3, while Chapter 4 provides an assessment of the experimental outcomes. Chapter 5 explores potential avenues for future research.

# Chapter 2

# Basics & Related Work

## 2.1 Graphs

A symbolic structure that is composed of a non-empty set of vertices $V$ and a set of edges $E$ (can be empty), where each edge connects two nodes such that $u, v \in V$, is known as a *graph* $G = (V, E)$. Two nodes are called *adjacent* to each other if an edge exists between them. These nodes can be classified as *source* and *target*, respectively, denoting the direction of the edge. The number of edges that are incident upon a node is known as the *degree* $d(v)$ of that node, meaning it is the number of edges for which this node is either a source or a target. The average degree of the nodes in a graph can be a measurement to describe how dense a graph is, calculated as $d(G) = \frac{1}{|V|} \sum_{v \in V} d(v) = \frac{2|E|}{|V|}$.

A graph is called a *directed graph* or a *digraph* if all the edges in the graph have direction. If the edges do not mean to show any direction and just the connection/relation between two nodes then the graph is called an *undirected graph*. A graph can have both undirected and directed subgraphs/parts. This graph would be called a *hybrid graph*.

In graph theory, a path is a non-repeating order of nodes such that each node

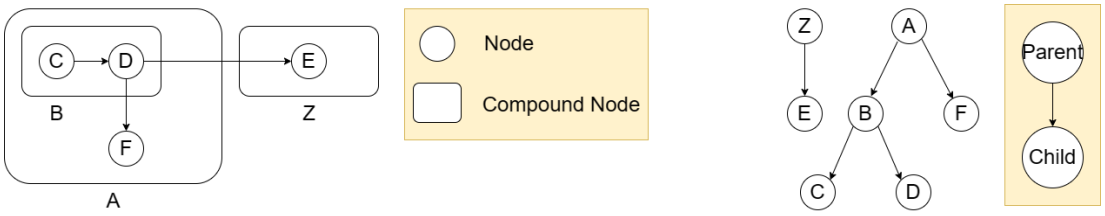



Figure 2.1: A compound graph $G = (V, E, F)$, where $V = \{A, B, C, D, E, F, Z\}$, $E = \{e_1 = \{C, D\}, e_2 = \{D, E\}, e_3 = \{E, F\}\}$ and $F = \{(A, B), (A, F), (Z, E), (B, C), (B, D)\}$, containing three compound nodes $A$, $B$ and $Z$, and two inter-graph edges $\{D, E\}$ and $\{D, F\}$ (top left), its inclusion tree or nesting hierarchy (right) [5]

in it is connected to the node next in the order through an edge. In directed graphs, a path should be along the direction of edges. We call an undirected graph *connected* if there exists at least one path between any of the two nodes in the graph, if there is none, then we say the graph is *disconnected*. If there is only one path between any of the two nodes in the graph, then that graph is called a *tree*, and a tree where one of the nodes is assigned as a root is known as a rooted tree.

Given a digraph $G = (V, E)$, a *path* is a sequence of nodes $a_0, a_1, a_2, ..., a_n$ where edge $(a_i, a_{i+1}) \in E$ for each $i \in \mathbb{N}$. A closed directed path of length greater than 1 from node $u$ to node $v$ such that $u = v$ is called a *directed cycle*. A graph containing at least one directed cycle is known as a *directed cyclic graph* and a graph that does not contain any directed cycle is called a *direct acyclic graph* or more commonly known as *dag*.

A *hierarchical graph* is one where the nodes/vertices are grouped in layers or levels, and usually, the edges connect nodes from one level to the nodes in the next level. These graphs are represented by *dags* as they have directed edges, and their acyclic nature helps in defining clear hierarchical relations and avoids infinite loops while traversing the graph.

There is a difference between laying out a *DAG* and a *layered graph*. The hierarchy in the DAG is maintained strictly and edges have only one direction. A parent and a child should be in different layers of the laid-out graph. This

condition is equivalent to the fact that no two nodes in the same level of the hierarchy should have an edge between them. While this condition is true for the DAG layouts, it is not supported by the layered graphs. Layered graphs can have edges in any direction and they can be between the nodes of the same layer, as long as the nodes are structured or placed in layers.

A *subgraph* $H = (V', E')$ is a graph taken from the original graph $G = (V, E)$ such that $V' \subseteq V$ and $E' = \{(u, v) \mid u, v \in V'\}$

A *compound graph* $G = (V, E, F)$ is composed of several components, a set of vertices $V$, a set of edges $E$ and a set of inclusion edges $F$ [15]. An inclusion tree $T = (V, F)$ is composed of vertices $V$ and inclusion edges $F$, which is used to represent the hierarchical nature of a compound graph. An edge $e = (u, v)$ in $F$ denotes that $u$ is a parent of $v$. A compound node $u$ is a parent of all the nodes that belong to the child graph or all the graphs nested within it. A node cannot have an edge with its parent or child $E \cap F = \phi$ (Figure 2.1). An edge that is between the nodes $x$ and $y$ when $x$ and $y$ are in the same level of the same compound node is called an *inter-graph edge*, and if they are in different levels, then it would be called an *intra-graph edge*. Compound nodes are helpful in representing the hierarchical structure as well as they can be used to represent a subgraph [16] (Figure 2.1).

A *hybrid compound graph* $G = (V = V_h \cup V_o, E = E_h \cup E_o \cup E_i, F)$, on the other hand, is a compound graph with two classes of nodes: $V_h$ and $V_o$, composing nodes in the hierarchy and outside the hierarchy, respectively. Edges of a hybrid compound graph are classified into three: $E_h$, $E_o$ and $E_i$, respectively representing those edges within the hierarchy, those between nodes outside the hierarchy and those across the nodes in the hierarchy, and nodes outside the hierarchy. $F$ again defines an inclusion tree for representing varying levels of abstraction through parent-child relations. The set of nodes in the directed part of the graph, which are connected to the undirected nodes are called *seed nodes* $V_s = \{v \mid v \in V_h \wedge u \in V_o \wedge [(u, v) \in E_i \vee (v, u) \in E_i]\}$. Each undirected component can have more than one seed node. An example hybrid graph with varying components defined above can be found in Figure 2.2.

Within the scope of this work, we will restrict the inclusion relations to the nodes and edges outside the hierarchy. In other words, nodes within the hierarchy are not allowed to be nested.

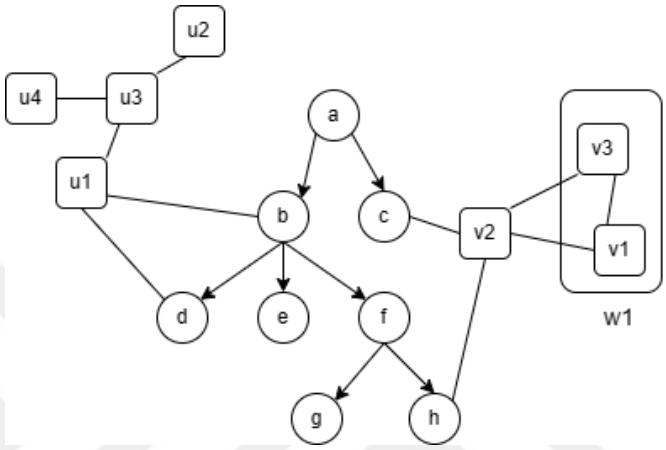



Figure 2.2: An example hybrid compound graph $G = (V = V_h \cup V_o, E = E_h \cup E_o \cup E_i, F)$, where $V_h = \{a, b, \cdots, h\}$, $V_o = \{u1, u2, u3, u4, v1, v2, v3, w1\}$, $V_s = \{b, c, d\}$, $E_i = \{\{b, u1\}, \{d, u1\}, \{c, v2\}\}$ and $F = \{(w1, v1), (w1, v3)\}$

## 2.2 Graph Layout

*Graph layout* refers to the organization of nodes and edges within a graph to optimize its usability, comprehension, and visual appeal. This process involves employing a mathematical function that maps each node to a unique position in either 2D or 3D space and each edge to a Jordan curve in the same space, with the endpoints aligning with the corresponding node positions [13]. The quality of a graph layout is very subjective and may depend upon several factors, but there are specific metrics such as the number of edge crossings, average edge length, and area of the graph that are broadly accepted to be the measure of a high-quality aesthetic graph layout [17]. In the case of compound nodes, the child nodes are supposed to be within the boundary of their parent, making the graph layout compact and organized, providing a comprehensive view of the relation between nodes.

Generally, while laying out a graph, nodes are given initial random positions

on a plane. An ideal arrangement is then achieved by repeatedly refining these positions. However, an *incremental layout* method can be useful when a user is content with the current node positions and wants to just improve the arrangement without disturbing their relative positions much. By starting the layout process from the current node positions and making small, gradual changes, this method preserves the graph's existing structural map while improving the overall aesthetic and coherence of the graph in the presence of relatively minor changes in the topology and/or geometry of the graph.

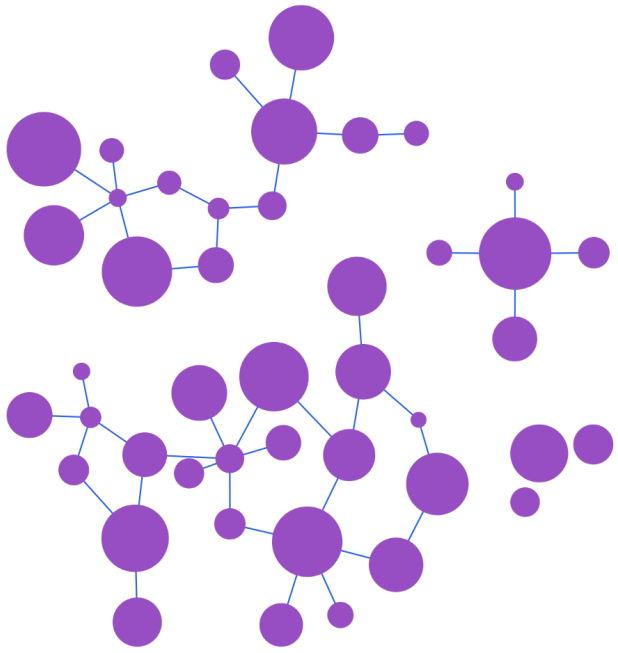


Figure 2.3: A forced directed layout sample [6], produced using fCoSE [5]

A variety of graph layout algorithms exist, including force-directed, hierarchical, and orthogonal, each one having its own uses and benefits. Each algorithm is used for a specific need, with some being versatile for various graph types and others optimized for other types. Force-directed, especially the spring embedder layout algorithm is really good at refining the layout of graphs and can prove to be important for the kind of graphs we are trying to layout. Force-directed algorithms yield visually appealing layouts for both simple and complex graphs.

### 2.2.1 Hierarchical Layout

A *hierarchical layout* is used to display the graph elements/nodes in a hierarchic fashion. The nodes of the graph are placed in layers on top of each other, denoting some type of order of precedence. They are assigned to the layers or ranks such that the overall direction of most edges if not all, is the same. Then for the aesthetics, within the layers, the nodes are ordered such that there are minimal edge crossings and the graph is not cluttered much. These layouts can be used to easily represent the organizational structures, file directories, family trees, biological pathways, or any scenario where items have a clear "parent" or "superior" relationship with others.

In 1981, Sugiyama et. al. presented a method to draw layered graphs [18]. The algorithm was named after its creator Kozo Sugiyama. The algorithm starts first by detecting and removing cycles from the graph and making it a directed acyclic graph. In the next step, the nodes are assigned layers or ranks. Then the edges that span more than one layer are modified, by adding dummy nodes in the layers between of source and target node of that edge. This makes sure that all the edges are between adjacent layers and that no edge spans more than one layer. Then all the nodes including the dummy ones are moved horizontally such that edge crossings are minimal. The horizontal placement of nodes within each layer is calculated and dummy nodes are removed. The steps are shown in Figrue 2.4.

Dagre is a graph layout extension, basically for DAGs, which are slightly different than layered graphs as explained earlier. It is a collection of different algorithms, each designed for a specific problem that is faced while creating a quality graph layout, an example shown in Figure 2.5. As it is used to lay out DAGs, it can be compatible with hierarchical graphs as well. The basics of this algorithm are taken from ideas described by Gansner for drawing directed graphs [13] and Sugiyama [18]. It places the nodes in ranks, according to the hierarchy defined by the edges of the input graph. The basic idea for node assignment is that the nodes with fewer incoming edges will tend to be placed on top of the hierarchy

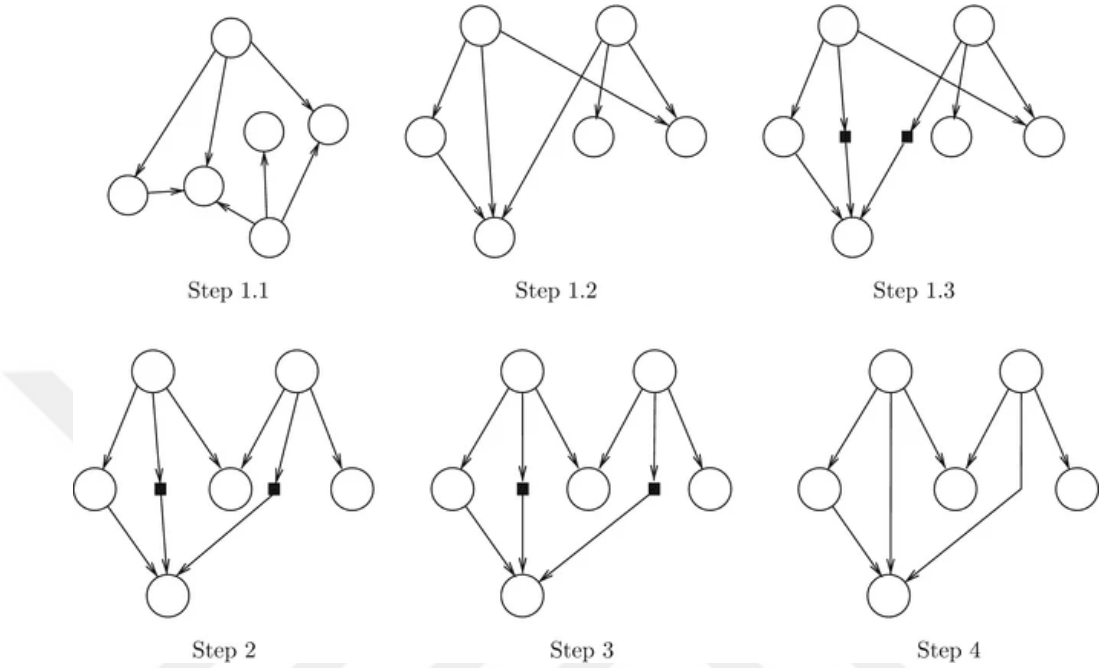



Figure 2.4: Steps following Sugiyama algorithm [7]

and nodes with fewer outgoing edges will be placed at the bottom. Then for the edge crossing minimization, it uses various techniques described by Michael Junger and Petra Mutzel [19] such as median ranking and barycenter ranking. Then in the end when it's time to assign final positions to nodes on the canvas or a plane, it makes use of the Fast and Simple Horizontal Coordinate Assignment method [20].

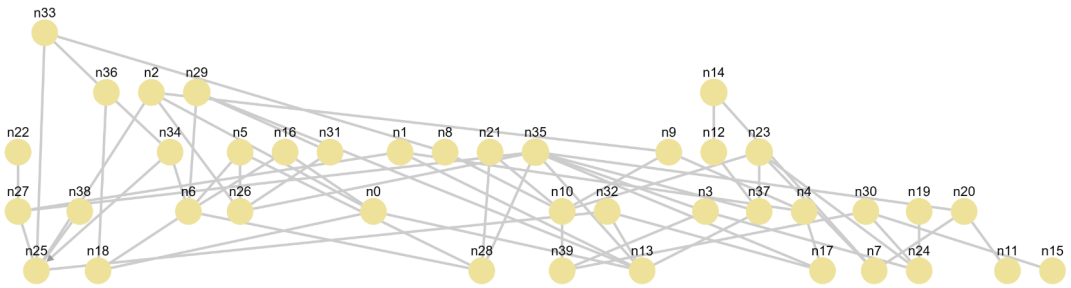



Figure 2.5: A hierarchical graph laid out using Dagre [8]

### 2.2.2 Force-Directed Layout

One of the most widely adopted techniques in automating the layout of graphs is the force-directed layout. It is known for its simplicity in its implementation and produces very aesthetic layouts that are pleasing to the eyes. One force-directed layout is known as spring embedders [9]. This method takes inspiration from forces of nature and tries to simulate graph representation by applying those forces to the elements of the graph. Eades [21] did a study where they considered nodes as metal rings or spheres that would repel each other, and considered edges as springs, which would try to achieve an ideal edge length by pushing or pulling the attached nodes, depending on their deviation from the ideal edge length.

The layout starts with placing the nodes in random positions. Then let the nodes move freely, by applying the above-mentioned forces on them over many iterations. After each iteration, the nodes are moved to their new positions depending on the forces that are exerted on them. This process is repeated iteratively until the overall forces or energy of the system is down to the threshold defined (Figure 2.6). The algorithmic run-time complexity of this approach is $\mathcal{O}(|V|^2 + |E|)$ in each iteration. The repulsion forces calculated between the node pairs account for the $\mathcal{O}(|V|^2)$ and spring forces are calculated for every edge, and for that $\mathcal{O}(|E|)$. Here it is worth noting that Eades did not apply forces on nodes that were adjacent, and also used a logarithmic version of spring forces that is not what Hooke's law states.

First, Eades [21] in 1984, presented a heuristic to draw graphs and it paved the way for Kamada and Kawai [22] and Fruchterman and Reingold [23]. These next versions improved the quality and performance of the spring embedder and also modified it for specific use cases.

So after this study, Kamada and Kawai [22], only used spring forces and assumed that all the nodes are attached to each other with a spring. Then after the initial placement of the nodes, they tried to bring the total energy of the system down by letting the nodes move freely to their new positions, by solving partial differential equations.

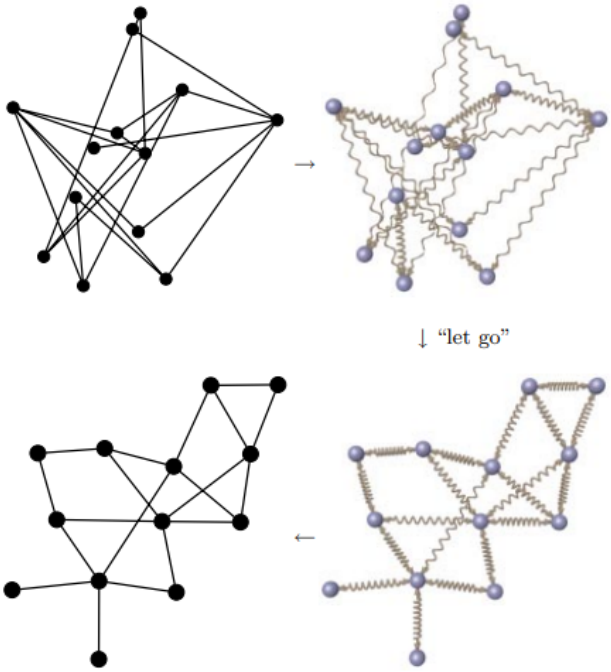



Figure 2.6: An illustration of the force-directed approach [9]

After Kamada and Kawai, Fruchterman and Reingold [23] used Eades' approach and modified it by applying repulsion forces between every node in the graph. Plus, they used a spring force formula similar to Hooke's law. This would also try to distribute the nodes evenly on the plane. Below are the formulas that they used in their algorithm:

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

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

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.

They also made use of an approach called simulated annealing, in which the

system is forced to lower its energy after every number of iterations. Initially, the energy or the forces of the system are high, but as we move forward with the iterations, the energy is forced to be lower by lowering the force constants. Eventually, after some iterations, the system reaches a stable state.

Fruchterman and Reingold also improved this algorithm by introducing a grid variant of it, making the overall run-time complexity $\mathcal{O}(|V| + |E|)$. They made use of the heuristic that the nodes that are far away from each other have negligible repulsion forces, so they should ignore the repulsion calculation for them. To implement this, they divided the plane into squares, by making a grid and placing each node in a square. When calculating the repulsion forces, they would only calculate the force for a node with nodes in its neighboring square. As the algorithm tries to place the nodes at a uniform distance from each other, it is safe to say that there would only be a small number of nodes surrounding a specific node at a time. So as the number of squares around a node is constant, the number of nodes to calculate the repulsion forces will also be similar. Eventually, the time complexity for the repulsion forces comes down to $\mathcal{O}(|V|)$ making the overall run-time complexity $\mathcal{O}(|V| + |E|)$.

The number of publications or research is relatively low for the automatic layout of compound graphs, owing to their complex structure, as it can be tedious to handle several factors such as inter-graph edges, and a flexible number of nesting levels. The studies by Sugiyama and Misue [24], Sander [25] and Eades et al. [26] were conducted on the compound graphs having directed edges. Similar to previous work, these algorithms, and methods were good for graphs having hierarchies defined by directed edges.

The CoSE algorithm [14], makes use of Fruchterman and Reingold's force-directed model [23]. The basics are the same here, nodes being considered as rings/spheres that would repel each other and edges acting as springs that would try to maintain an ideal edge length. But to handle the compound nodes, CoSE uses some other techniques as well.

Like previous models, CoSE makes use of physical analogies to model upon.

Here, the compound node is considered an "elastic cart", which is free to move along both axes. Also, it can stretch or contract its boundary depending on the placement of nodes in its child graph. Because the compound nodes can have different sizes, as it depends on its child graph nodes, CoSE uses the minimal Euclidean distance between the boundaries of two nodes, unlike the previous models where the nodes were considered to be of uniform shape and size.

For the edges in the same graph level, known as intra-graph edges, CoSE considers their ideal edge length to be the same. For inter-graph edges, it also takes into consideration the depth of the edge's source and target nodes from their common parent in the inclusion tree.

CoSE makes use of Fruchterman and Reingold's grid variant model to calculate the repulsion forces between the nodes. This helps in lowering the time complexity of the algorithm. An important point is that it only calculates the repulsion forces for the nodes that are in the same depth level and graph. Plus, the force exerted on a compound node is propagated down to its children, which then move and update the boundary of that compound node. In addition to repulsion and spring forces, CoSE uses gravitational force as well. Its magnitude is very small as compared to the other forces mentioned before, but it helps the disconnected components of the graph, stay close with the rest of the components. The time complexity for this algorithm is $\mathcal{O}(|V| + |E|)$. The overall runtime is the same as the grid-variant with an additional factor of $k$, making it $\mathcal{O}(k(|V| + |E|))$ where $k$ is the number of iterations. The number of iterations required for medium and large graphs would be high.

Some methods added constraint support in this force-directed layouts [27] [28], which can be made to use for directed hierarchical graphs. CoLa (Constraint Layout) is a famous constraint-based layout algorithm, which is devised from a number of other studies [29]. It also improves the layout over a number of iterations and makes use of a gradient-projection algorithm to calculate the position of the node in each iteration. For constraint support, such positions are calculated which satisfy the constraint requirements. This constraint feature can be used to work with compound graphs, but it further increases the already high run-time

complexity.

fCoSE is another spring embedder that uses a number of techniques to add constraint support, which it demonstrates can be used for laying out directed graphs [5]. fCoSE is faster than its counterparts and it can produce aesthetically pleasing graphs, but the required information from the user can be a lot.

Sugiyama and Misue also presented a modified version of the spring embedder model called Magnetic Spring Algorithm [30]. This method tries to lay out graphs with both unidirectional and bidirectional edges and calls them *mixed graphs*. Instead of using the traditional spring forces for the edges, this model introduces a new force, *magnetic spring force*. The edges are assigned spring force or magnetic spring force based on the property of direction, as directed edges are assigned magnetic edges and undirected edges are assigned non-magnetic edges. The non-magnetic edges are dealt with as before using the spring force, but the magnetic edges also receive the magnetic force. The magnetic force is a rotational force applied on the edge, depending on the magnetic field, and the magnetic field can be parallel, polar, concentric, orthogonal, or polar-concentric depending on the layout we want to achieve. This algorithm can be used to lay out both directed and undirected graphs. Another algorithm that tries to lay out mixed graphs, where there can be directed and undirected edges, is Hierarchical Edge Bundling [31]. It presents the adjacency relation between the entities in the hierarchy, and as the visualization of such a graph can become very chaotic and dense, it bundles the similar edges and tries to present a visually appealing graph. The visual clutter is reduced by bundling the edges together and then presenting the directed edges with curves and undirected edges with a straight line.

Most of the methods described above are used for either directed or undirected graphs. The two methods, Magnetic Spring Embedder and Hierarchical Edge Bunding, can be considered close but they have their own limitations. Hierarchical Edge Bundling does not always produce hierarchical drawings. Magnetic Spring Embedder can produce directed graph layouts, but those are not strict hierarchies, as it only can produce layouts that show the flow of the graph. In addition to these shortcomings, neither of these methods can handle compound

nodes. The ones with constraint support can be utilized to work with both, but the users would have to add constraints by themselves and the quality of the layouts is not very good due to artificial constraining. So given the kind of graphs that we are dealing with, we could not find any algorithm based on a holistic approach that produces an aesthetically pleasing layout for hybrid graphs. So, in this context, we introduce an algorithm named HySE to address this gap.

# Chapter 3

# Algorithm

This chapter covers the details of HySE, how it processes the input graph and divides different components, how it places them initially, and then refines the overall layout. The details of each step are also presented in this chapter.

As defined in Section 2.1, the graphs we are dealing with will have both directed and undirected parts, called hybrid graphs. The directed part will be allowed to have only simple nodes, while the undirected components may contain compound nodes as well.

To handle both directed and undirected nodes in a specified hybrid graph, we divide the problem of initial positioning into two and try to solve them one by one, and then proceed with a holistic approach, refining the overall layout of the graph. This starts with getting the layering information of the directed part in the given input. Then we need to find the position of the undirected components of our graph. After the initial placement of the nodes, the forces need to be applied, and along with this, the edge crossings in the directed part need to be minimized. At last, a post-process phase needs to be run in order to even out the space between nodes in our graph. Hence the steps followed are (also shown in Figure 3.1):

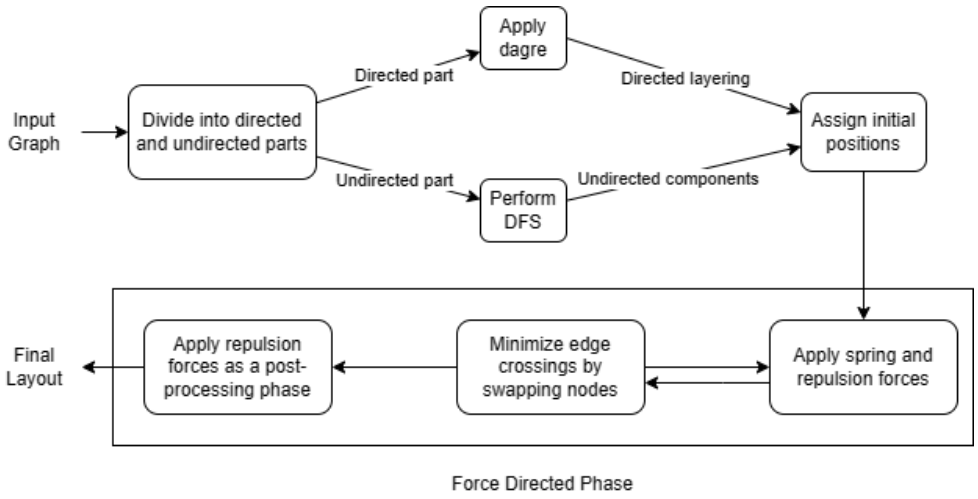
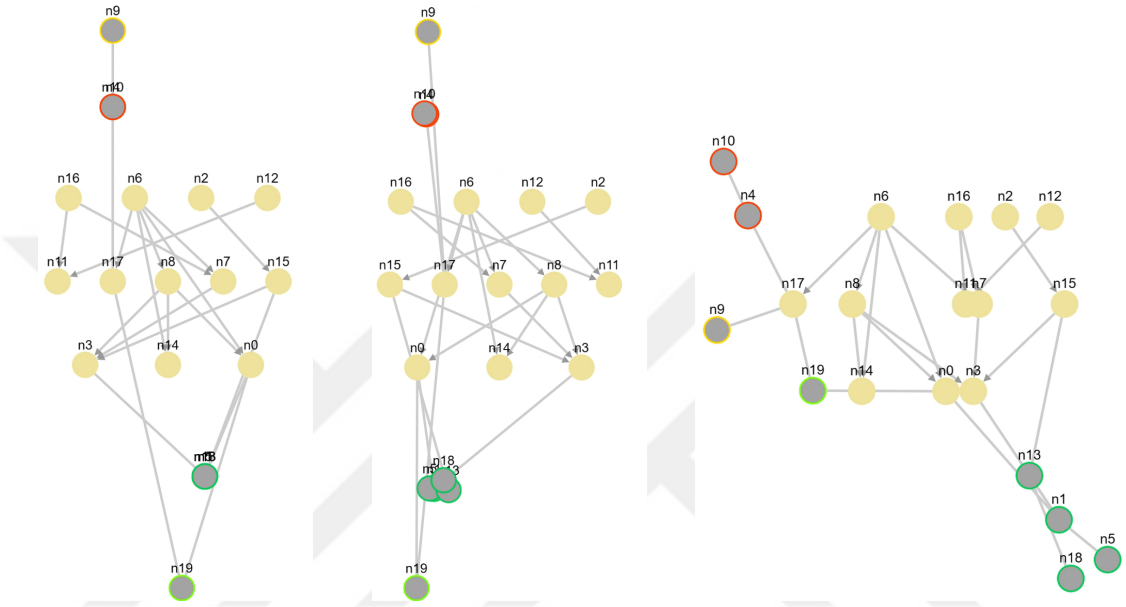


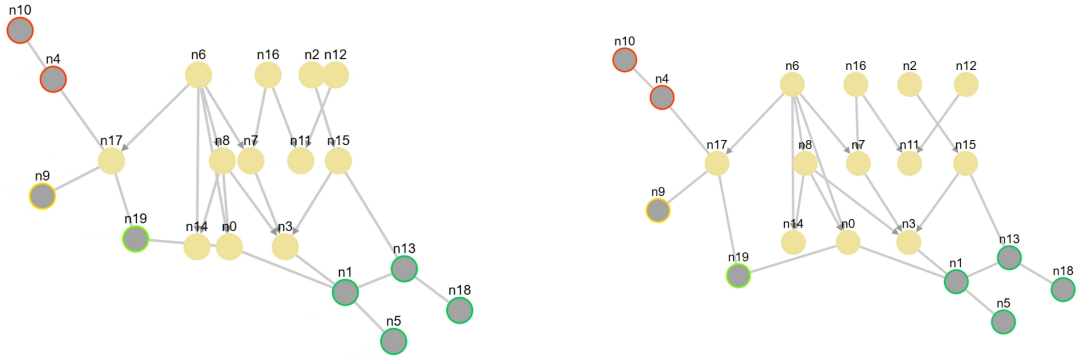

Figure 3.1: An overview of HySE algorithm. In the initial placement phase, the graph elements are positioned, directed part in layering, and undirected components around them. The second phase is the force-directed part where spring and repulsive forces are applied to refine the layout.

- Initial Placement Phase
  - Layering and Initial Placement of Directed Nodes
  - Initial Placement of Undirected Nodes/Components
- Force-Directed Phase
  - Main Force Simulation Phase
    - Repulsion and Spring Forces
    - Directed Node Swaps/Edge Crossing Minimization
  - Post-Process Repulsion Phase

Figure 3.2 shows the visualization of a sample hybrid graph during different steps in the algorithm.



(a) Step 1.1 (b) Step 1.2 (c) Step 2.1.1



(d) Step 2.1.2 (e) Step 2.2

Figure 3.2: (a) The First step is to place the directed nodes initially in layers using the layering information from Dagre and randomizing the order of nodes in each layer. (b) Then place the undirected nodes using the virtual compound nodes around the directed part and randomize the position of nodes in the undirected components. (c) The force-directed phase starts and the graph starts to take shape. (d) The nodes in the directed part overlap and have no distance between them after the force-directed phase, so (e) the post-processing phase tries to keep an even distance.

## 3.1 Directed Subgraph Initial Placement

The first part of the algorithm is placing the directed nodes in layers. For this purpose, we make use of some traditional layout algorithms used for directed and layered graphs [8] [18] [12]. These algorithms are used partially. Partial use here means that only the processes or steps until the layering stage are utilized. We first produce an acyclic graph to be ranked and placed in layers.

Making the graph acyclic is driven by the inherent nature of hierarchical layout as it is supposed to be for the DAGs in the first place. This is also important because cycles introduce dependencies that prevent a proper hierarchy and the layering process. Making the graph acyclic makes the ranking process feasible.

*Feedback Arc Set* (*FAS*), introduced by Eades et al. [32], is used to remove the cycles from the input graph. FAS is a set of edges that can be removed to make the graph acyclic. This problem is actually an NP-complete problem. There are heuristics that are extended by traditional sorting algorithms to solve this problem [33]. Page Rank algorithm can also be used to compute FAS and reduces the size of FAS by 50%, when compared to the ones produced by previous techniques [34]. Other techniques include edge reversal and node removal. A cycle is detected by traversing the graph and identifying the nodes involved in it.

After the graph is made acyclic, the Network Simplex Algorithm is used to rank the nodes in layers. The Network Simplex Algorithm assigns the ranks to the nodes and iteratively improves the ranking to reduce the edge length. The nodes are assigned initial rank values using the longest path algorithm, which assigns the rank values to the lowest position possible. Generally, this causes the bottom ranks to be wide and the edges to be extra long. But after the initial rank assignment, it creates a compact tree, having almost no difference between the actual length of the edge and the expected length of the edge, and iteratively improves the rank values by finding the edges that are unnecessarily long.

After the ranking is finalized, the actual Dagre implementation performs other functions to optimize the layout further and minimize the edge crossings in the

graph. Along with the rank, it sets the order of the node in each rank to minimize the crossings. We, however, only require the ranking information for the directed part of our hybrid graph. It is very important to note that in order to show that our proposed model works as well as Dagre, we randomize the order of nodes in each layer so that we can use our proposed model to minimize the number of edge crossings.

After the nodes are assigned layers, their positioning on the plane is relatively easier. The first node in the first layer is assigned the position $(0, 0)$ on the canvas, and the next nodes are placed using the *order gap* and *rank gap* provided by the user. The $x$ and $y$ coordinates of the position of a node are calculated by the formula 3.1.

$$\begin{aligned} x &= nodeIndex \cdot orderGap \\ y &= layerIndex \cdot rankGap \end{aligned} \tag{3.1}$$

where $nodeIndex$ is the index of the node in its layer or rank and $layerIndex$ is the index of the rank in which that node resides.

## 3.2 Undirected Subgraph Initial Placement

For the initial placement of the undirected subgraph, we use the concept of the seed nodes as defined in Section 2.1. Before we place the undirected nodes close to their seed nodes, we need to find the connected undirected components and their respective seed nodes. For this purpose, we run a DFS on undirected nodes. Notice that this is a DFS that takes the compound nodes into account similar to that used in [35]. To be specific, when the traversal reaches a compound node, it is also assumed to have reached any of its children and vice versa.

Undirected nodes in the graph are placed inside virtual compound nodes such that each compound node contains one connected undirected component. These

virtual or dummy compound nodes are used so that it is easier to manage the forces within and across each undirected component. It also eases the implementation and keeps all nodes in a component together while keeping each undirected component separate from the others.

**Algorithm 1**

```
function PrepareCompoundNodes( G = (V, E, F) )
    groups ← {}
    seeds ← {}
    visited ← set()
    groupNum ← 0
    undirectedNodes ← GetUndirectedNodes(V)
    for each v ∈ V do
        groupNum ← groupNum + 1
        CompoundNodeDFS(v, groupNum)
    end for
end function
function CompoundNodeDFS( node, groupNum )
    if visited.Contains(node) then
        return
    end if
    visited.Add(node)
    groups[groupNum].Add(node)
    directedNeigbors ← GetDirectedNeighbors(node)
    if directedNeighbors then
        seeds[groupNum].Add(directedNeighbors)
    end if
    for each child ∈ node.Children() do
        CompoundNodeDFS(child, groupNum)
    end for
    if node.Parent() then
        parent ← node.Parent()
        CompoundNodeDFS(parent, groupNum)
    end if
    for each neighbor ∈ node.GetUndirectedNeighbors() do
        CompoundNodeDFS(neighbor, groupNum)
    end for
end function
```

Undirected nodes are grouped together based on their connectivity. This is described in Algorithm 1, which finds the seed nodes of each component. It also

keeps track of the compound nodes in the undirected part. DFS would run with the same *groupNum* on children and the parent of an undirected node. So all compound nodes, even in different levels would be part of the same group which would eventually be one undirected component and one virtual compound node.

After the undirected nodes are grouped and divided into components, it is time to place them around the hierarchy and close to their seed nodes. First, for each group a virtual compound node is created and all the undirected nodes are placed inside it as its children. The dimensions of the virtual compound node are calculated by taking into account the number of children it is going to have. Then, the bounds of the hierarchical part are calculated, which is done by finding the leftmost node, rightmost node, topmost node, and bottommost node. This gives us the boundary of the directed part of our graph. So we will place the virtual compound nodes surrounding these points.

The virtual/dummy compound nodes can be placed close to one of the four sides of the directed part, which is bounded by the rectangle calculated. This can be seen in Figure 3.3. We get the seed nodes for an undirected component and take the average of their position on both axes x and y. For $k$ seed nodes having the coordinates $(x_i, y_i)$, the seed center would be calculated as given in 3.2

$$seedCenter = (\frac{1}{k}\sum x_i, \frac{1}{k}\sum y_i) \tag{3.2}$$

This *seedCenter* is then used to find the closest side of the boundary of the directed part. The shortest geometric distance between the seed center and each side is calculated and the closest side is chosen. The compound node is placed by using the extremely placed directed node of that side and adding half of its own width plus some buffer value to maintain a visible distance between the directed node and the compound node. During this process, the compounds already placed can overlap with the new compound node being placed. This is avoided by keeping lists of each side containing the already placed compound nodes on that side, and checking if the new compound node being placed overlaps with any of them. If it does, then it is placed further using the boundary of that

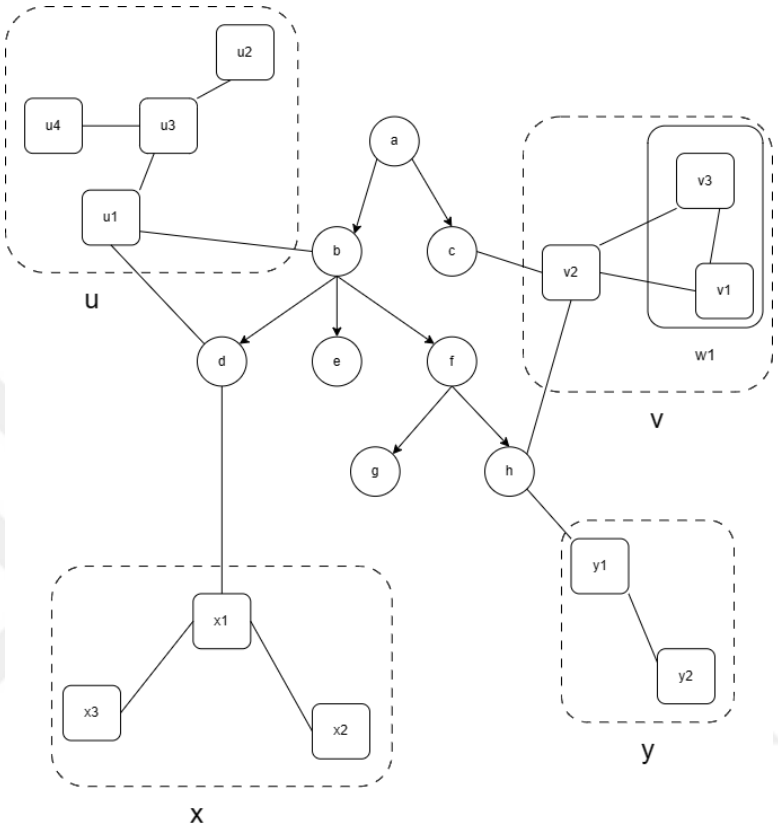



Figure 3.3: Various cases of seed node connectivity. The nodes $b$ and $c$ are just connected to one undirected component, while the nodes $d$ and $h$ can be seen connected to two undirected components. Similarly, the undirected components $x$ and $y$ have just one seed node of their own, while the undirected components $u$ and $v$ have two seed nodes.

already placed node. The main goal here is to make sure the initial positions of undirected components are nonoverlapping. This can be seen in the Figure 3.4.

After an appropriate position for the compound node is found, its children, the actual undirected nodes, need to be placed within its boundary. For this purpose, one could simply place them all in the center of the compound node. This way all the nodes would be at the same position and would overlap with each other. Even though this is rather convenient and simple, in succeeding steps, when repulsion and spring forces are applied, there would be an absurd amount of forces due to fully aligned nodes. So, we opt to place these nodes within the boundary of the compound node at random positions near the center of the compound such that

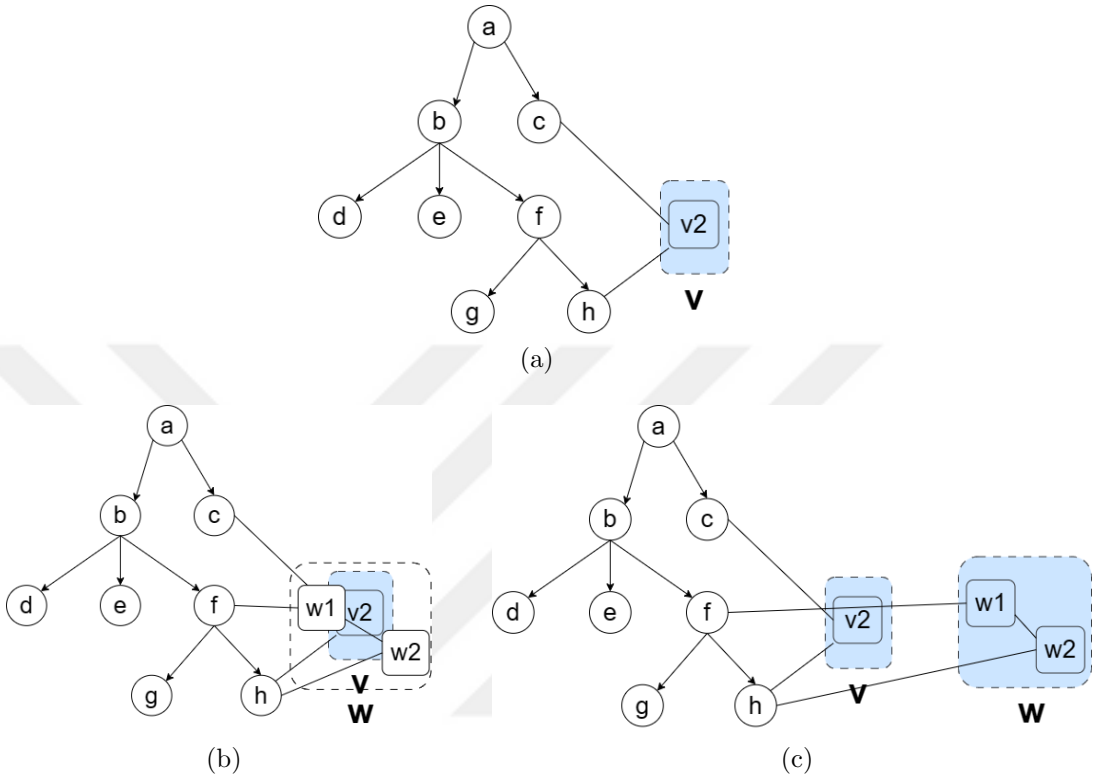



Figure 3.4: Let's assume a scenario where (a) an undirected component $V$ is placed to the right of the directed part due to its seed nodes $c$ and $h$. (b) Another undirected component $W$ has seed nodes $f$ and $h$, so it needs to be placed to the right of the directed part as well. (c) But because $V$ is already placed at the supposed position of $W$, $w$ is moved further in the same direction and some gap or buffer is also added in between $w$ and $v$.

they may still overlap with each other but will not be concentric, as shown in Figure 3.5.

This whole process makes sure that the initial positions for the undirected components are close to their seed nodes. The Algorithm 2 explains the process. Eventually, this helps the force-directed part to be over in much fewer iterations as the graph layout converges much faster because the undirected nodes are already close to their ideal positions in a quality graph layout.

**Algorithm 2**

```
function PlaceCompounds( compoundNode, side, nodesToCheck )
    if side = UP then
        xVal ← seedCenter.x − compoundNode.width/2
        yVal ← topMostNode.y − compoundNode.height − buffer/2
        compoundNode.position ← {x : xVal, y : yVal}
        CheckCollision(compoundNode, side, nodesToCheck)
        nodesToCheck.Add(compoundNode)
    end if
end function
function CheckCollision( compoundNode, side, nodesToCheck )
    collidingNode ← NULL
    for each node ∈ nodesToCheck do
        if Intersects(compoundNode, node) then
            collidingNode ← node
        end if
    end for
    if not collingNode = NULL then
        if side = UP then
            xVal ← seedCenter.x − compoundNode.width/2
            yVal ← collidingNode.y − compoundNode.height − buffer
            compoundNode.position ← {x : xVal, y : yVal}
            CheckCollision(compoundNode, side, nodesToCheck)
        end if
    end if
end function
```

## 3.3 Force-Directed Phase

In this phase, HySE iteratively improves the draft layout, which was generated by placing the graph elements as defined earlier. The inspiration for this process is drawn from the spring embedder model explained in Section 2.2. However, since the type of graph spring embedder targets is different than the hybrid graphs we are focusing on, it needs to be modified to fit our needs. The overall structure of the algorithm is given in Algorithm 3.

The nodes in the directed part of our graph should not move in the $y$-axis (fixed $y$-coordinate), because that is a hard and inherent constraint of being in the hierarchy. For this reason, only the forces in the x-axis are calculated for these nodes. For the undirected parts, however, all forces are calculated and used for movement in both axes.

### 3.3.1 Repulsion Forces

As explained previously in Section 2.2, the nodes in the graph are considered metallic spheres or rings that repel each other. This force spreads out the nodes, preventing overlaps and creating a more evenly-spaced layout. This repulsion force is modeled as an electrostatic force that is exerted by charged particles. Using this analogy, the nodes are considered to be the charged particles that push each other away and because they follow the inverse-square law, the force decreases as the distance between them increases. The force of repulsion that nodes exert on each other depends on the *nodeRepulsionForce* constant provided by the user and the square of the distance between them.

For the compound nodes, it makes sense to include the weight, which would be the number of children, while calculating the repulsion force. As in the physical world, the more charged a particle is, the more force it would generate in the system. The more the number of nodes involved in a repulsion calculation, the

**Algorithm 3**

```
function LAYOUT( G = (V, E) )
    PREPARECOMPOUNDNODES()
    layoutEnded ← false
    layoutEnded ← false
    while layoutEnded = false do
        layoutEnded ← TICK()
    end while
    POSTREPULSIONPHASE()
end function
function TICK
    totalIterations ← totalIterations + 1
    if totalIterations/ConvergenceCheckPeriod = 0 then
        if ISCONVERGED(|)|totalIter > maxIter then
            return true
        end if
        if isFastCooling then
            currentIter ← maxIter − totalIter
            coolFac ← initCoolFac ∗ (currentIter ∗ coolCoeff)/maxIter
        else
            coolFac ← initCoolFac ∗ (currentIter)/maxIterations
        end if
    end if
    RESETTRACKINGVARIABLES()
    UPATEBOUNDS()
    SPRINGFORCES()
    REPULSIONFORCES()
    SWAP()
    MOVE()
end function
```

more the exerted force would be. Similarly, when a force is exerted on the compound node, it would be dispersed to its children. This would make the larger compound nodes "heavier" and difficult to be moved by the smaller components. This will help in keeping the graph layout dictated by the bigger parts of the graph.

The force for the directed nodes would be calculated without the involvement of children as we do not allow the directed part of the graph to have compound nodes. Hence, the force calculation for two nodes $A$ and $B$ in the directed part is given as:

$$F_{rep} = \frac{repulsion_A + repulsion_B}{distance^2} \tag{3.3}$$

It is the same for the undirected part but because it might contain compound nodes, we have to take into account their children as well. For simple nodes, the ones not having any children, the number of children is considered to be 1, so that the equation remains valid for them and correctly calculates the force. So when involving the children, Equation 3.3 becomes:

$$F_{rep} = \frac{(repulsion_A + repulsion_B) \cdot (cA \cdot cB)}{distance^2} \tag{3.4}$$

where $cA$ and $cB$ are the number of children of node $A$ and $B$ respectively.

The distance used in these equations is Euclidian distance, which is simply the difference between $x$ coordinates and $y$ coordinates of both nodes. After the repulsion force is calculated, it needs to be split into the $x$-component and $y$-component of the force. This is done by utilizing the distance in each axis between them as given below:

$$\begin{aligned} F_{rep_x} &= F_{rep} \cdot \frac{distance_x}{distance} \\ F_{rep_y} &= F_{rep} \cdot \frac{distance_y}{distance} \end{aligned} \tag{3.5}$$

These repulsion forces in each axis are directly used to calculate the displacement of the node in that axis.

To make use of the grid-variant properties of the spring embedder, a minimum distance is defined. This acts as a threshold for the distance between the nodes, for the repulsion force to be calculated. Similarly, for the directed part, repulsion force is calculated between the nodes of the same layer. This is because the movement is supposed to happen only along one axis, and there is already a factor of rank gap, which maintains the distance between different layers. Even within the layers, only for some initial iterations the repulsion force is calculated between all the nodes. This phase is controlled by a user-provided parameter, which is the percent of maximum iterations where the repulsion force is calculated between every node within a layer. After this phase, the repulsion force is calculated between the node and a certain number of neighbors to the left and right of the node. This number of neighbors can also be given user-defined.

For the compound nodes, the repulsion force is calculated between two nodes if they have the same parent similar to that in CoSE. There can be no force being exerted by the parent to its children and vice versa. Also, there can be no force exerted directly by a node of a different level on another node. The force would be calculated between the child nodes of the same parent and then if they have children, that force would be trickled down to the children. This explains that the compound nodes do not move on their own. It is their leaf nodes, having no children, that move and define the dimension and position of the compound node. After every iteration, the boundary of the compound nodes is updated so that the calculations in the next iteration are accurate and as expected.

As explained previously, repulsion forces would then be calculated between the directed nodes and the virtual compound nodes, because they would be in the same graph (both are children of the root graph). The force exerted by the directed node would be applied to the undirected component as a whole and not the individual undirected nodes.

### 3.3.2 Spring Forces

As in conventional spring embedders, the edges in the spring embedder model are seen as springs when laying out the graph. These springs will exert attractive forces on the nodes attached to their ends. The intuition behind such a concept is that the connected nodes would be kept close to each other. This would lead to a layout where we can better see the clusters of the connected components as the geometric distance of the nodes should be aligned with the graph-theoretic distance between them. In addition, this will help in keeping the edge lengths close to the given ideal edge length, making the overall layout compact. Edge spring forces are inspired by Hooke's Law which multiplies the spring constant with the change in the length of the spring. Hence, the spring force is calculated as:

$$F_{spring} = edgeElasticity \cdot (L_{edge} - L_{ideal}) \tag{3.6}$$

The *edge elasticity* here is a constant that is used to determine how much a spring will attract or repel for some change in its length. This parameter is similar to *node repulsion* of nodes which is used in repulsive force calculations, requiring a tuning during implementation.

For the spring forces, there can be three types of edges that we will face in the given hybrid graphs. One that is between directed nodes, the second that is between undirected nodes, and the third one that is between an undirected and a directed node. The edges that are between the same type of nodes are handled similarly to each other, the only difference is that the edge between directed nodes will only be used to calculate the spring forces in the $x$-axis.

Similar to that of repulsive forces, the spring forces will be translated into the $x$ and $y$ axis as given in Equation 3.7. The length of the edge in each axis is calculated by simply subtracting the positions of the attached nodes in that axis.

$$F_{spring_x} = F_{spring} \cdot \frac{L_{edge_x}}{L_{edge}}$$
$$F_{spring_y} = F_{spring} \cdot \frac{L_{edge_y}}{L_{edge}} \tag{3.7}$$

Edges across the compound nodes will take into account the depth of its connected nodes while calculating the edge length. This will help in keeping the end nodes outside the compound and the ones inside it at an adequate distance, preventing overlaps. In addition, the end node in the compound node will be affected individually, irrespective of the other end node's depth level or parent. This is a bit different than what we did in the repulsive force calculation. This is done because it will keep the connected node in the compound node close to its parent's boundary and will keep the edge length smaller. This can be seen in Figure 3.6.

The spring forces play a vital role in the swapping mechanism, explained later in Section 3.3.3, which is used to minimize the edge crossings in the hierarchical part. The attractive forces of the springs complement the repulsion forces of the nodes and bring the system to a stable position.

Both the forces described above follow the superposition principle, meaning the net force or the total force being applied on a node is the sum of all the forces being applied at any moment. The movement of the node in each axis is based on (or proportional to) the total force being applied to the node in that axis.

### 3.3.3 Minimizing Edge Crossings

As the hybrid graphs contain a directed part, HySE needs to make sure that the number of crossings in that hierarchical part is minimal. This is important for a quality graph layout as a high number of edge crossings can be very distracting, and can make finding and locating important pieces of information

time-consuming and challenging. For this purpose, HySE makes use of a swapping method, where nodes in each layer are swapped with each other based on the forces being applied to them.

The basic idea is simple following a heuristic approach detailed in Algorithm 4. If two adjacent nodes in a layer are trying to move in opposite directions with forces of magnitude above a defined threshold, they will be swapped, as shown in Figure 3.8. Let $A$ and $B$ be two nodes that are trying to move in positive $x$ and negative $x$ ($-x$) direction respectively, where $A$ is currently to the left of $B$. Because the movement of the nodes is a direct consequence of the sum of all the forces being applied to them, it would make sense to let them move freely. But as $A$ and $B$ come close to each other, they will exert repulsive forces on each other and force each other to move back to their original place. Here, if their desire to move to their new place, left for $B$ and right for $A$, given by their forces in the $x$-direction, is greater than a threshold, HySE will swap their positions.

The swaps are performed in a controlled fashion, however. First, in each layer, adjacent nodes that are trying to move in the opposite direction are identified and kept as node pairs. Then, it is made sure that a node pair was not swapped recently; for this purpose, a *minPairSwapPeriod* threshold is used, which defines the minimum period or the number of iterations after which a node pair is allowed to be swapped again. This is done to prevent the oscillatory motion of the nodes, as they can move ahead of their ideal position and might want to go back in the direction of their original position. If they do not fall victim to the minimum swap period, then they are swapped with each with priority given to the most willing nodes. This means that node pairs are sorted by the swap force in decreasing order and then swapped one by one. This is done as we also keep check of the connected pairs, so that only the first one, which is more willing, is swapped. For example, there are three nodes $A$, $B$, and $C$, present in a layer, with node pairs $(A, B)$ and $(B, C)$ willing to be swapped. Let's assume that the pair $(A, B)$ has more swap force than the other one. So HySE will swap the nodes $A$ and $B$ and keep track of adjacent nodes of these both. So when swapping the pair $(B, C)$, it will be checked if any of the nodes were already involved in any swapping in this iteration. If yes, then that node pair will not be swapped; in this case, $B$ and $C$

will not be swapped.

**Algorithm 4**

```
function SWAP( nodePairs = [(u, v)]  )
    swapPairs ← [()]
    if totalIterations % swapPeriod = 0 then
        for each pair(u, v) ∈ nodePairs do
            pair.swapForce ← u.TotalForce_x − v.TotalForce_x
            if totalIterations − swapIter[(u, v)] >
                    minSwapPeriod and swapForce > swapForceLimit then
                swapPairs.ADD((pair(u, v), swapForce))
            end if
        end for
        SORTBYSWAPFORCE(swapPairs)
        swappedNodes ← {}
        for each pair(u, v) ∈ swapPairs do
            if not u in swappedNodes and not v in swappedNodes then
                SWAPNODES((u, v))
                swappedNodes.ADD(u)
                swappedNodes.ADD(v)
                swapIter[(u, v)] ← totalIterations
            end if
        end for
    end if
end function
```

When swapping two nodes, the dimensions of the nodes are also taken into account to avoid overlapping upon swap. For the nodes with uniform node dimensions, the centers of the nodes are swapped, but for the non-uniform nodes, this is not the case. The nodes are swapped such that they remain within the boundary defined by them both. For this purpose, the difference in the widths of both nodes is taken and added to the current $x$-position of the opposite node, as shown in Figure 3.7.

Let *nodeA* and *nodeB* be two nodes having different dimensions and *nodeA* is placed to the left of *nodeB*. Then the new $x$ positions for both nodes would be calculated as given in Equation 3.8.

$$nodeA_x = nodeB_x + (\frac{nodeA.width}{2} - \frac{nodeB.width}{2})$$
$$nodeB_x = nodeA_x + (\frac{nodeA.width}{2} - \frac{nodeB.width}{2}) \tag{3.8}$$

The concept described above can sometimes cause the nodes to oscillate and swing back and forth around the same positions, causing the overall system to go into an oscillation mode and not letting the energy or forces of the system go down. This will cause the layout algorithm to run unnecessarily for a maximum number of iterations and increase the layout execution time. To handle this problem, we make use of a cooling factor, which lowers the energy of the system as we progress over the iterations, as described in [21], and also add a fast cooling method to further lower the cooling factor value. The cooling factor is used when calculating the displacement of the nodes. The displacement of the nodes is calculated using the forces and the cooling factor as described in Algorithm 5.

### 3.3.4 Post Processing

HySE converges and stops the processing when the *totalDisplacement* in the system is below a threshold or the nodes in the graph are oscillating. We find oscillating behavior by taking the difference between total displacement in the current iteration and total displacement in the previous iteration. If the difference is less than a threshold value then we recognize the graph to be oscillating and stop the layout process. During the layout, as mentioned earlier, we make use of a cooling factor to lower the energy of the system, which would make the total displacement lower and eventually bring the graph layout to a point of convergence. This cooling factor and fast cooling coefficient work really well in the graphs that have the freedom to move on both axes. But in the graphs where movement is restricted to one axis, in our case the directed part of the graph, the behavior is not ideal. As the repelling force decreases over the number of iterations, the nodes in the directed part come really close to each other. Because

**Algorithm 5**

```
function CalculateDisplacement( node )
    Δx ← coolFac * (repulsion_x + spring_x)/noOfChildren(node)
    if not node.IsDirected then
        Δy ← coolFac * (repulsion_y + spring_y)/noOfChildren(node)
    end if
    if Math.Abs(Δx) > coolFac * MaxNodeDisplacement then
        Δx ← coolFac * MaxNodeDisplacement
    end if
    if Math.Abs(Δy) > coolFac * MaxNodeDisplacement then
        Δy ← coolFac * MaxNodeDisplacement
    end if
    if node.Children( )then
        propagateDisplacementDown(Δx, Δy)
    end if
end function
function propagateDisplacementDown(displacement_x, displacement_y)
    for each child ∈ node.Children() do
        if child.Children( )then
            propagateDisplacementDown(displacement_x, displacement_y)
        else
            child.displacement_x ← displacement_x
            child.displacement_y ← displacement_y
        end if
    end for
end function
```

of the ideal edge length a node in the hierarchy would like to be placed directly under its predecessor. And if there are a considerable number of nodes in a layer, they start overlapping, as exemplified in Figure 3.9.

To handle this overlapping issue, we thought of different techniques. For example, placing the directed nodes like Dagre by using the order index of the node in the layer, stretching or expanding the graph by a certain factor, adding a constant to the current positions of the nodes, and moving them away from each other. But we wanted to continue with our holistic approach for this algorithm so we decided to do a post-process repulsion phase.

In this phase, we simply make use of already defined repulsion forces and apply those to the graph for a number of iterations, depending on the graph size. In these iterations, we ignore the cooling factor as it might lead to the same problem as before. We do not consider any convergence or oscillatory behavior, as there will only be a repulsion force being applied to the graph. This makes sure that the graph expands consistently according to the defined forces.

## 3.4 Time Complexity

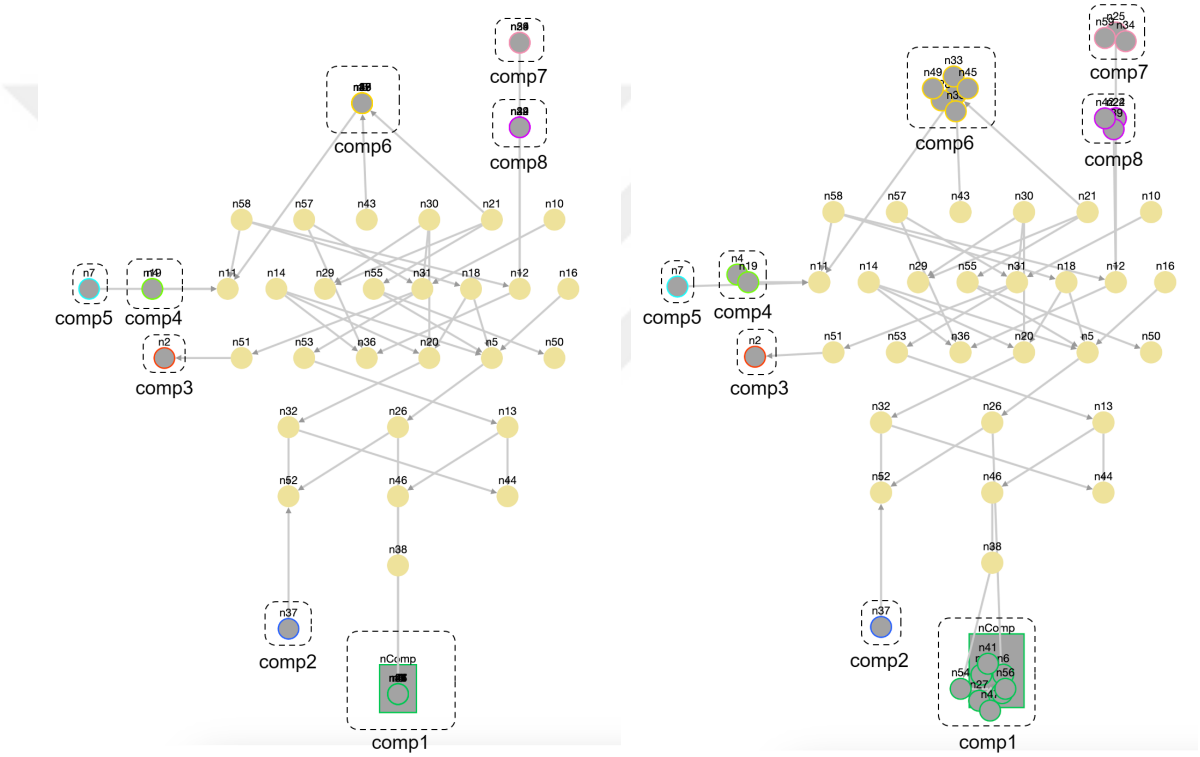
The first step of the algorithm is the layering of the directed subgraph and one of the most important parts of this process is the generation of FAS to make the input graph acyclic. This can be achieved in $\mathcal{O}(|E|)$ using the method defined in [36]. Then placing the nodes in the ranks is done using the Network Simplex Algorithm, which can be covered in $\mathcal{O}(|V| \cdot |E|)$ runtime. As mentioned in the next chapter, our graphs have an average degree between 2.5 and 3.5, so this makes the runtime for the Directed Subgraph Initial Placement phase $\mathcal{O}(|V|)$.

The second step is the generation of virtual compound nodes and their placement around the directed part. To obtain virtual compound nodes, we run DFS on the input graph. This DFS has the run time of $\mathcal{O}(|V| + |E|)$. Then placing

them around the directed part is actually trivial, as we are only looking at the defined four sides of the directed subgraph. So this part has the runtime complexity of $\mathcal{O}(|V| + |E|)$.

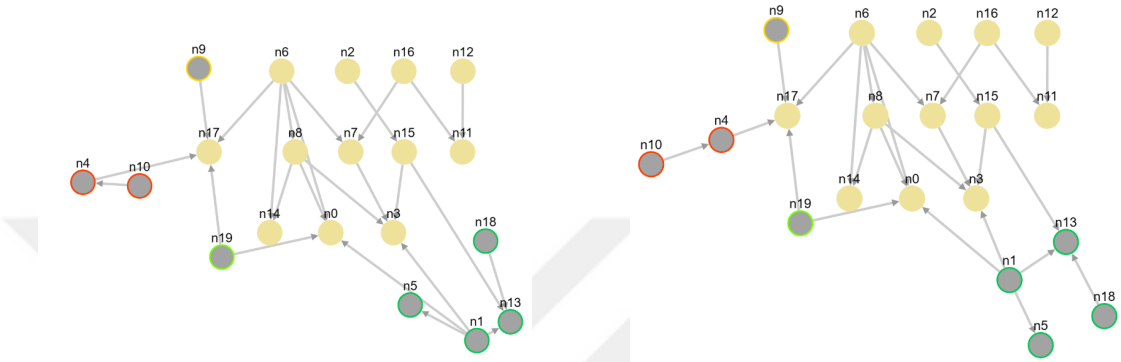
The next steps include iterations where forces are applied to the graph. The two forces applied have different runtimes. For the repulsion force, we make use of a grid to calculate the surrounding nodes of the target node to keep the force calculation linear, $\mathcal{O}(|V|)$. Normally, the repulsion force calculation is done in $\mathcal{O}(|V|^2)$ run time, because the repulsion force is calculated between all the node pairs. However, using a grid [23] helps to calculate the force of a node against only an expected constant number of neighbors. The same force calculation part can be used for the post-processing phase as the same repulsion forces are applied for a number of iterations. For the spring forces, we iterate over all the edges, which simply makes it runtime $\mathcal{O}(|E|)$. So, the force calculation part in each iteration is $\mathcal{O}(|V| + |E|)$. The node swaps in the iterative part also take $\mathcal{O}(|V|)$ as we consider node pairs for the adjacent nodes only to find their swap force and actually swap them. The movement of the nodes simply takes $\mathcal{O}(|V|)$ time. So, for this whole iterative part including the post-processing phase, the time complexity becomes $\mathcal{O}(k \cdot (|V| + |E|))$ where $k$ is the number of iterations, and can be highly dependant on the number of nodes.

Going over each step we calculated its overall run time complexity. When we join them together to get an overall runtime, HySE runs in $\mathcal{O}(|V|) + \mathcal{O}(k \cdot (|V| + |E|))$ time. Assuming that the real-life or benchmark examples of the graphs have a linear relation between the number of nodes and edges, we have $|E| = \mathcal{O}(|V|)$. Hence, the overall run time complexity of the algorithm would be $\mathcal{O}(k \cdot (|V|))$ where $k > 0$. As the number of iterations to bring the energy of the system below the defined threshold increases, the time required to calculate the forces would increase quadratically in the worst case. This is because, as mentioned earlier, $k$ is expected to be $\mathcal{O}(|V|)$ in the worst-case scenario.




(a) (b)

Figure 3.5: (a) Undirected nodes being placed in virtual compound nodes (dashed boundary) close to their seed nodes. Here, the $x$ coordinate of *comp*6 is the average $x$ coordinate of its seed nodes $n11$, $n43$ and $n21$. Similarly, for *comp*7, the $x$ coordinate is given by $n12$. The $y$ coordinate of *comp*7 is decided by checking if it overlaps with any virtual compound nodes already placed, which in this case it did with the *comp*8. So its $y$ axis value was updated and it got placed above *comp*8 (b) nodes being placed randomly inside the boundary of virtual compound node




(a) (b)

Figure 3.6: (a) The case where edges are not considered to be connected to child nodes but with their parent virtual compound node (b) Undirected nodes having edges with directed nodes are pulled automatically close to their seed nodes if spring force is only applied to the node directly connected with an edge. In this case, nodes $n4, n1$ and $n13$ get pulled to the side close to seed nodes.


(a) (b)

Figure 3.7: (a) Two nodes having non-uniform dimensions being swapped around their centers (b) Two nodes having non-uniform dimensions being swapped in the same region; instead of directly swapping the centers, the width difference is taken into account before swapping

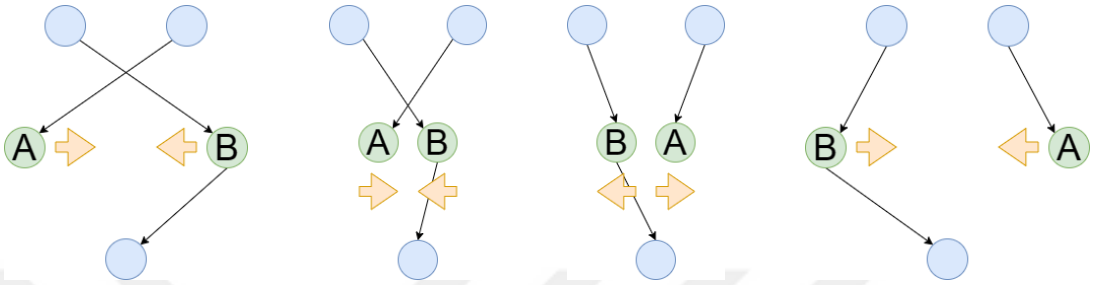



Figure 3.8: A sample swap process for edge cross minimization (a) consider two nodes A and B trying to move in the opposite direction (b) swap check for both nodes (c) nodes get swapped after the checks (d) repel each other to maintain distance between them

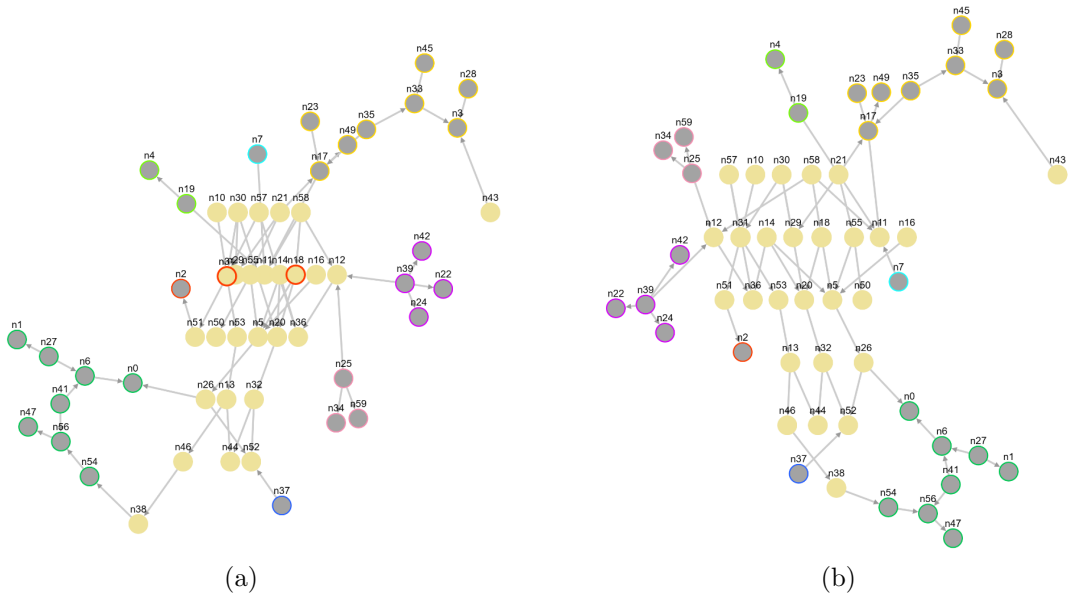



Figure 3.9: (a) HySE ran on a sample graph with low repulsion forces without the post-processing phase to imitate the behavior on large graphs, overlapping nodes are visible in the directed part (b) HySE ran on the same graph with same parameters but with post-processing phase in this case

# Chapter 4

# Implementation & Evaluation

To implement the proposed algorithm, we made use of Typescript and developed it as an extension for the Cytoscape.js library [37]. For the layering and initial placement of directed nodes as described in Section 3.1, we use Dagre [8]. Details of Dagre are mentioned in Section 2.2.1. The source code of Dagre is publicly available as it is an open-source implementation of a collection of algorithms. We modified Dagre's code, written in Javascript, so that we could obtain the layering information of the directed part.

We also utilized the already implemented CoSE [38] to reuse and customize the behavior for hybrid compound graphs by overriding the functionalities already implemented. The classes in our project were inherited from CoSE's corresponding classes, making CoSE the base for the HySE's implementation. Especially, the force calculation and movement methods were overridden in HySE's implementation as we needed to add constraints on the movement of nodes in the directed part. Other methods such as convergence, swapping nodes, and preparing compound nodes were also overridden as they were adapted for HySE. A demo application, which can also be found in the associated GitHub repository, was also implemented to showcase our algorithm along with some sample graphs to play with. These graphs laid out by HySE are in Figures 4.1 through 4.4.

## 4.1 Experiment Setup

We used the aforementioned implementation to perform experimental tests. For this purpose, we needed a directed graph dataset that also contained undirected components. After searching online on different platforms, no such dataset could be found. So we decided to build our own dataset using the ones that are publicly available. The dataset used can be found online here [39] that was used by Bridgeman et al. [40].

The dataset only contains directed graphs with the number of nodes ranging from 10 to 3000, having the average degree $d(G)$ between 2.5 and 3.5, small to medium-sized graphs. We used these directed graphs to build test graphs that would contain undirected components attached to the directed part. For this purpose, we ran BFS on the graph that is going to provide each undirected component, as explained in Algorithm 1 and Algorithm 7. We defined some parameters to manage the shape and structure of the undirected component. *Depth* parameter was used to control the depth of the undirected component, meaning the test graph generation method would not continue adding nodes to the undirected component if the depth limit is reached. *Ratio* maintained the balance between the number of nodes in directed and undirected parts. The seed nodes from the directed graph were chosen at random. For the nodes in the undirected part that are connected to the seed nodes, we chose them from the last level traversed by a BFS traversal. This was done so that the edges between directed and undirected nodes do not create much disturbance in the undirected part and it would feel natural and like the ones in real-life examples.

The experiments were conducted on a standard computer having Windows 10 as the operating system and equipped with an Intel i7-7700 CPU and 16GB of RAM with a max clock speed of 3.6GHz.

The parameters for HySE used for experiments are as follows:

- Ideal Edge Length = 50

**Algorithm 6**

```
function PrepareTestGraph(G_h = (V_h, E_h), G_o = (V_o, E_o))
    N ← Length(V_h)
    M ← N * (1 − R)
    components ← []
    visitedNodesLength ← 0
    while visitedNodesLength < M do
        n_o ← TestBFS()
        n_h ← GetRandomNode(V_h)
        CreateEdge(n_h, n_o)
        visitedNodesLength ← visitedNodesLength
            +Length(components[−1])
    end while
    for each v ∈ V_o do
        flag ← false
        for each comp ∈ components do
            if comp[v] then
                flag ← true
            end if
        end for
        if flag not true then
            RemoveNode(v)
        end if
    end for
    for each e ∈ E_o do
        flag ← false
        for each comp ∈ components do
            if comp[e.Source] and comp[e.Target] then
                flag ← true
            end if
        end for
        if flag not true then
            RemoveEdge(e)
        end if
    end for
end function
```

**Algorithm 7**

```
function TestBFS( G_h = (V_h, E_h), G_o = (V_o, E_o) )
    queue ← []
    levelNodes ← []
    lastLevelNodes ← []
    visited ← {}
    level ← 0
    n_o ← GetRandomNode(V_o)
    queue.Push(n_o)
    queue.Push(NULL)
    while queue and level < Depth and visited.Length < M do
        lastLevelNodes ← []
        while queue[0] not NULL do
            node ← queue.Pop()
            if node in visited then
                continue
            end if
            visited.Add(node)
            lastLevelNodes.Add(node)
            undirectedNeighbors ← node.UndirectedNeighbors()
            queue.Add(undirectedNeighbors)
        end while
        levelNodes.Add(lastLevelNodes)
        queue.Pop()
        if queue then
            queue.Push(NULL)
        end if
        level ← level + 1
    end while
    components.Add(Visited)
    return GetRandomNode(LastLevelNodes)
end function
```

- Rank Gap = 20
- Repulsion Force = 55000
- Ratio (Directed : Undirected) = 0.4
- Depth = 2
- Swap Force Limit = 15000
- Swap Period = 50
- Minimum Pair Swap Period = 10

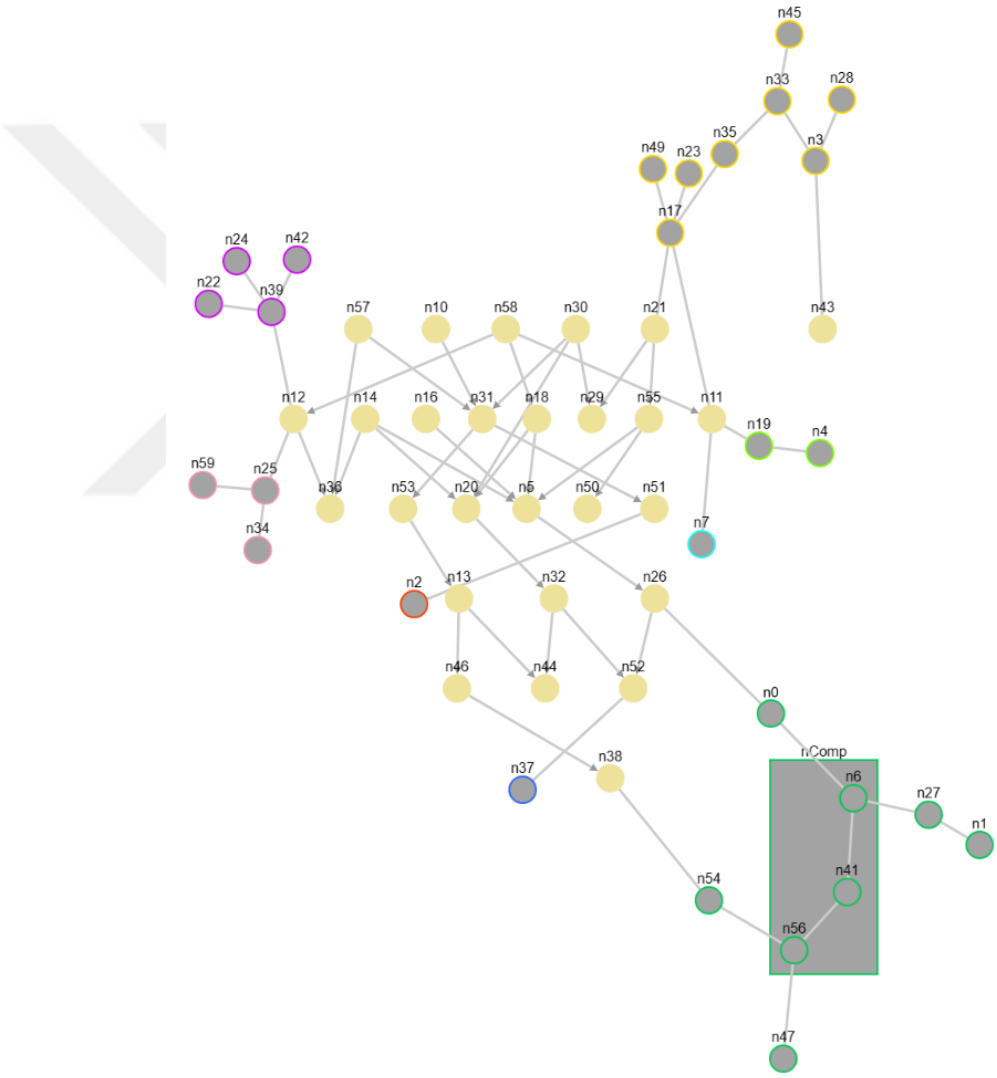



Figure 4.1: A sample of 27 directed nodes in the hierarchy and 29 undirected nodes in 8 components, including a compound node with 3 child nodes

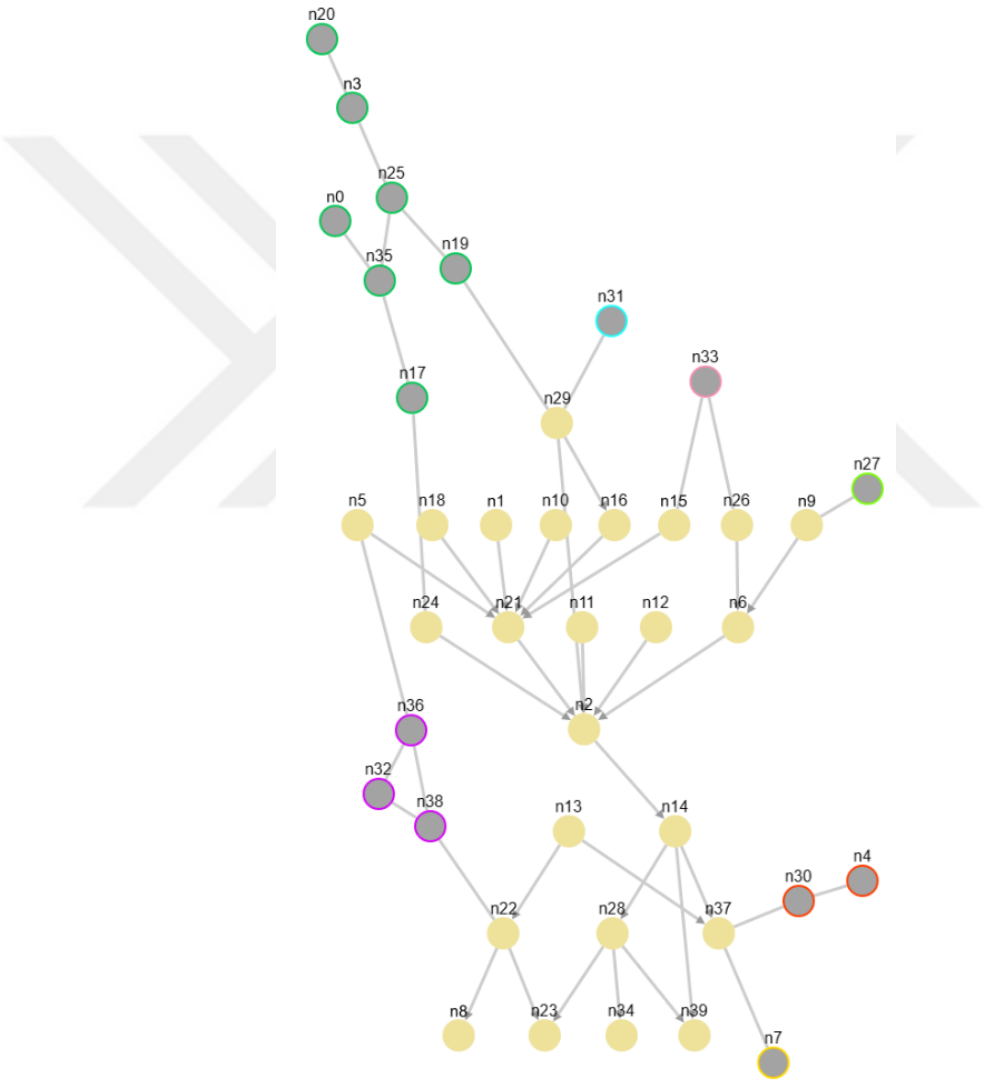



Figure 4.2: 16 undirected nodes divided into 7 components placed around the hierarchical graph of 24 directed nodes.

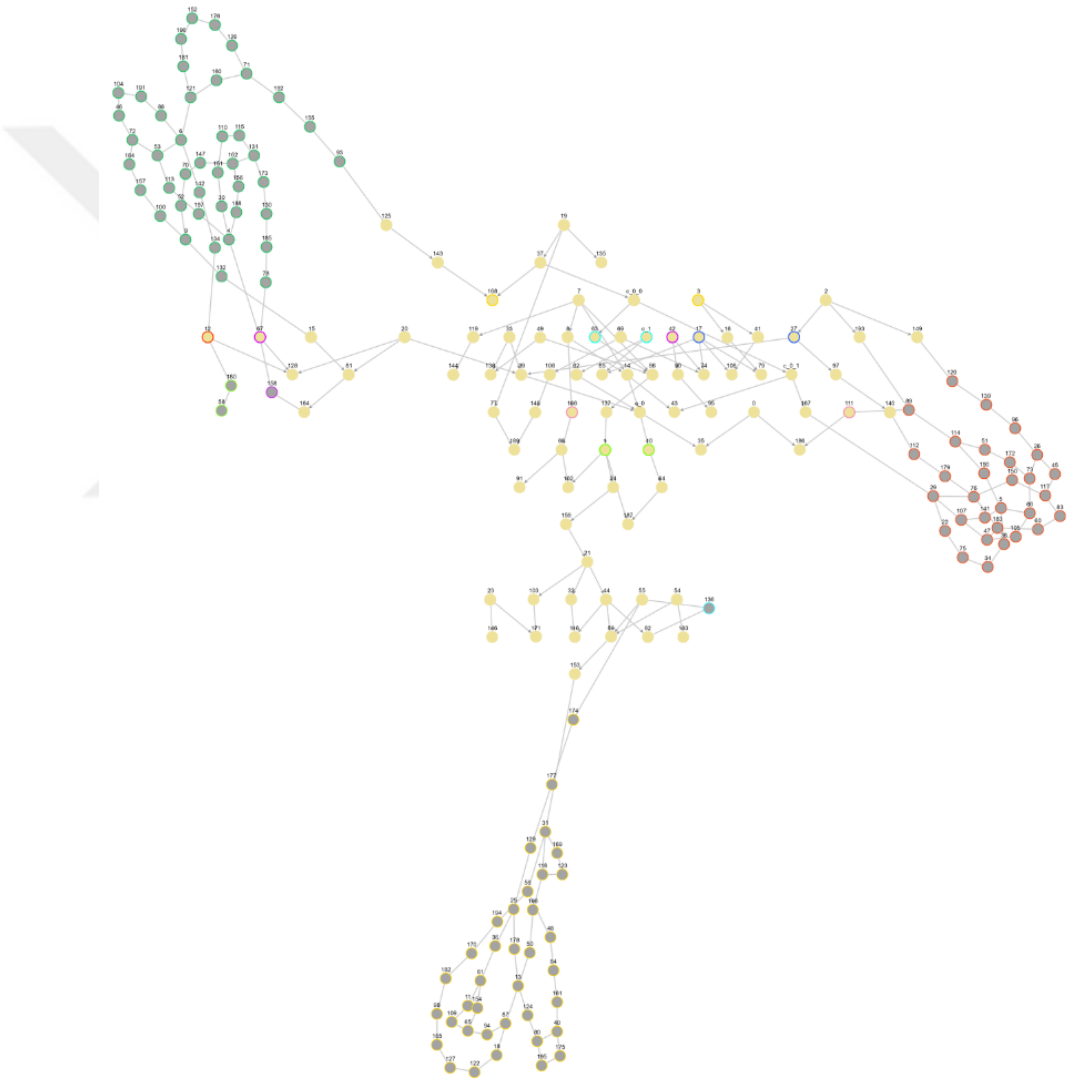


Figure 4.3: A sample graph with 85 directed nodes and around 100 undirected nodes in 6 components.

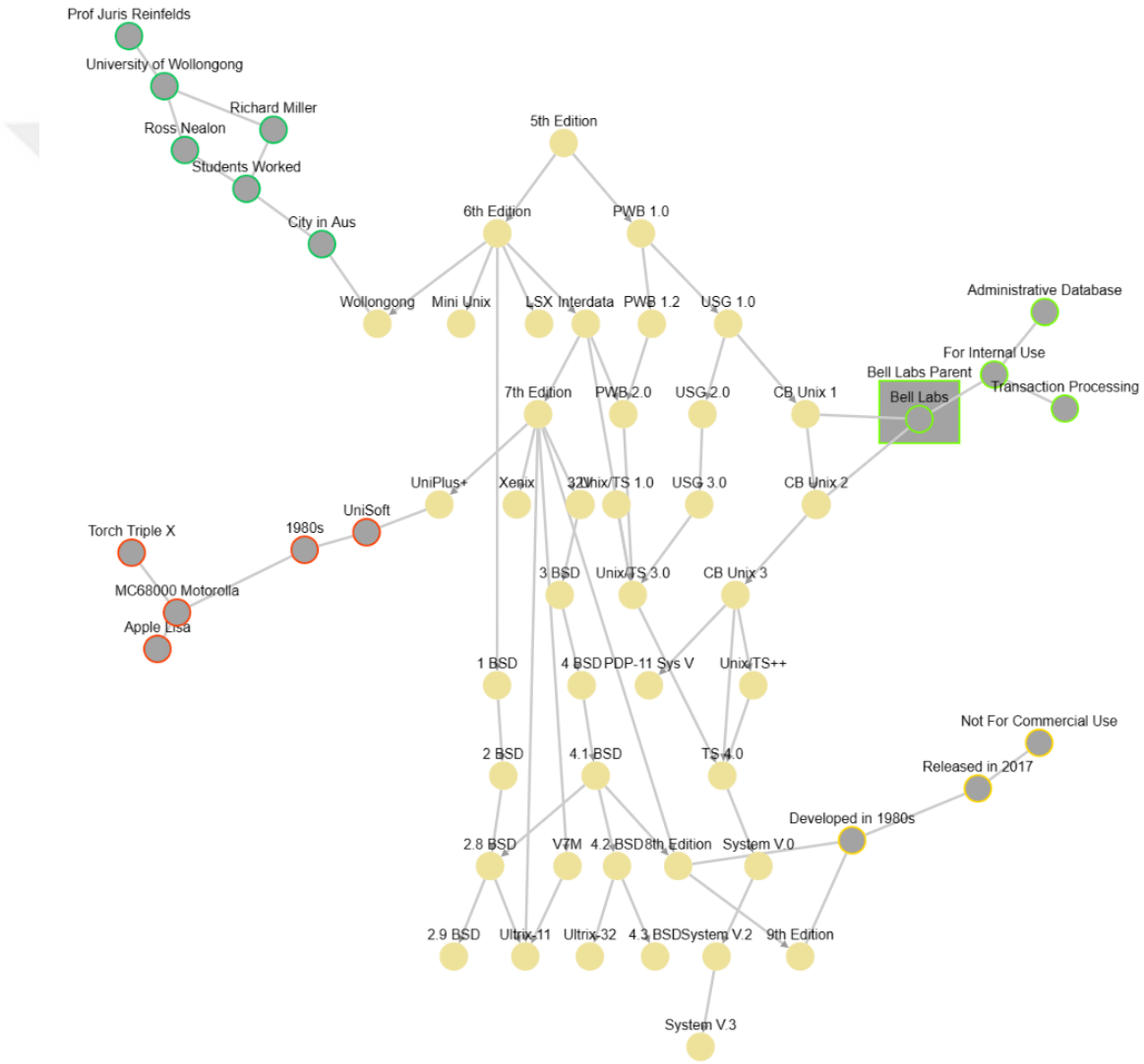



Figure 4.4: Hierarchical graph of different versions of Unix, along with the details of some versions given in the undirected part.

## 4.2 Results

These experiments were done by keeping the parameters of Dagre and HySE at levels where they would perform well because they both target different kinds of graphs. However, Dagre was chosen for the baseline model because of the directed part of the graph. The metrics were checked for different parts of the hybrid graph separately. This made it easier to compare only the directed part of the hybrid graph with the graph laid out by Dagre. Following are the charts of experimental results in Figure 4.5 through 4.9. These graphs had undirected nodes, 60% the number of the original directed graph nodes. For example, if the original directed graph had 100 nodes, 60 undirected nodes would be added in different undirected components, depending on the depth parameter, in the directed graph to make it a hybrid graph. The new test graph would have a total of 160 nodes, 100 directed ones, and 60 undirected ones.

The number of edge crossings in the directed part of HySE is very similar to those in the Dagre, as shown in Figure 4.5, demonstrating that the spring embedder approach works to minimize the edge crossings. Remember that the order of the nodes was shuffled after fetching the layering information from Dagre so that Dagre has no impact on the number of edge crossings being minimized by HySE.

Figure 4.6 compares the number of node overlaps in HySE and Dagre. The number of node overlaps in the directed part of the HySE graph layout is zero. This is because we use a post-processing phase where only the repulsion forces are exerted on the graph elements. This process makes sure that the graph elements are well-spaced. The node overlaps seen in the chart are in the undirected part of the graph, which are due to the modifications in the convergence part taken from the CoSE algorithm.

The comparison for the total area of the graph is given in Figure 4.7. The total area of HySE's graph is larger than that of Dagre's graph because the graph size is almost 1.5 times bigger in terms of the number of nodes. In addition, our

hybrid graphs had an undirected part around the hierarchical part which also added to the increase in the total area. But the total area for the directed part is about the same as that of Dagre.

The average edge length of HySE's graph, despite being larger in terms of number of nodes and area, is less than that of Dagre, shown in Figure 4.8. This is because of the force-directed nature of HySE. The largest edges are between the directed and undirected parts which are due to the restricted movement of nodes in the directed part and our method to treat undirected components as a single virtual compound node.

The only downside to force-directed algorithms is their run time. It can be seen clearly in Figure 4.9. Because they improve the graph layout iteratively, it takes time to find positions for the graph elements where the energy of the system would be below the threshold. To improve the run-time and quality of HySE, we place the undirected components close to their ideal positions, so that it takes fewer iterations to get to the optimal location. But still, the time complexity of the iterative phase is $\mathcal{O}(k \cdot (|V| + |E|))$, where $k$ is very closely related to the number of nodes, making the overall runtime quadratic. As the graphs became larger in terms of the number of nodes, it linearly increased the time for the initial placement phases.

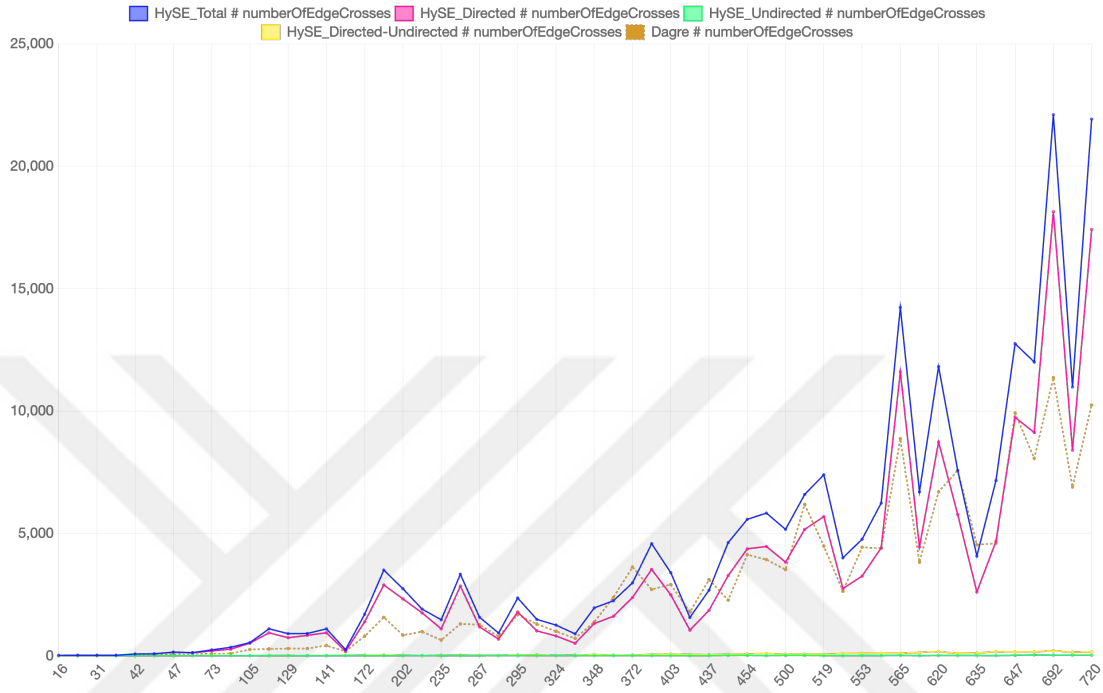



Figure 4.5: Comparison chart of "Number of Edge Crossings" between Dagre and different parts of HySE : 60% undirected nodes of original graph

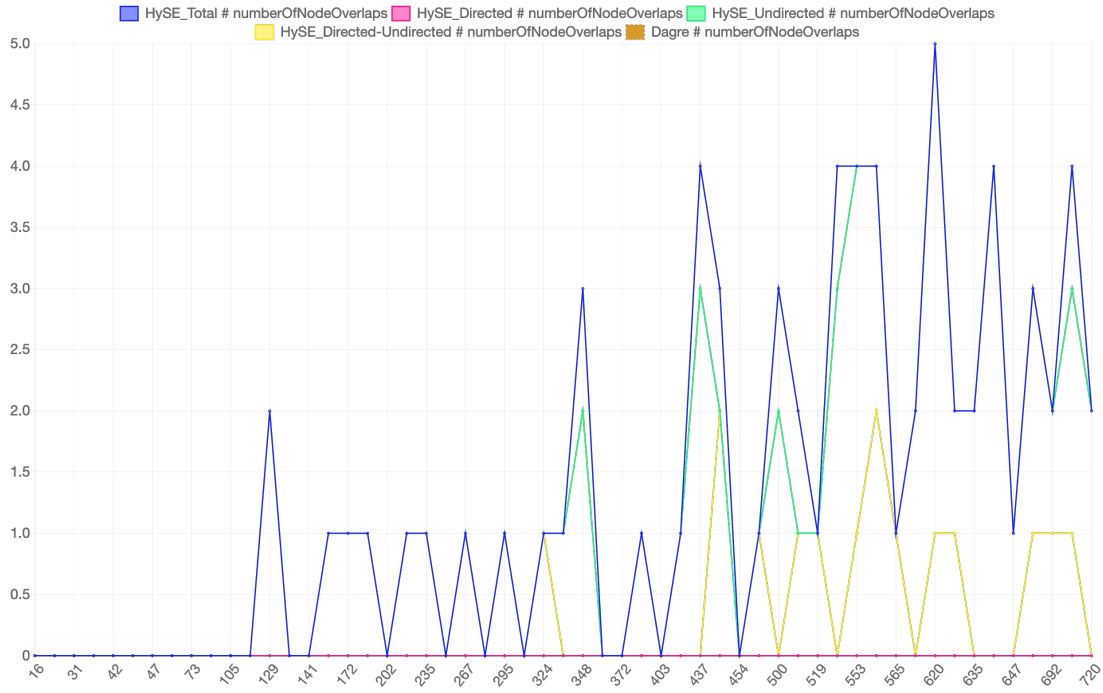



Figure 4.6: Comparison chart of "Number of Node Overlaps" between Dagre and different parts of HySE : 60% undirected nodes of original graph

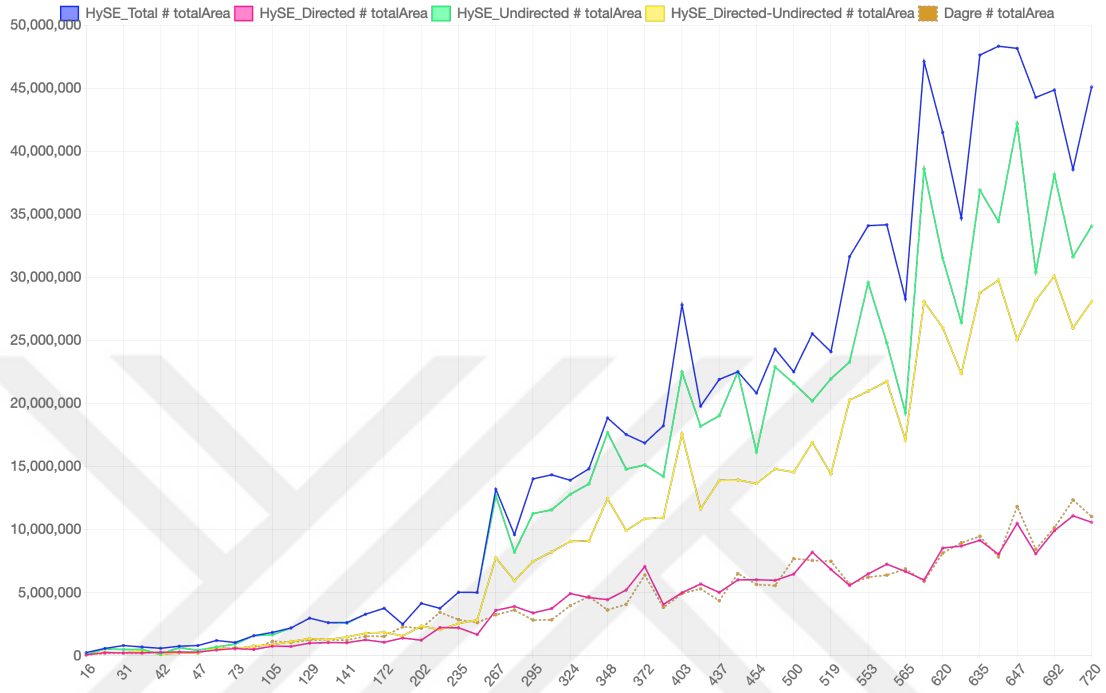



Figure 4.7: Comparison chart of "Total Area" between Dagre and different parts of HySE : 60% undirected nodes of original graph

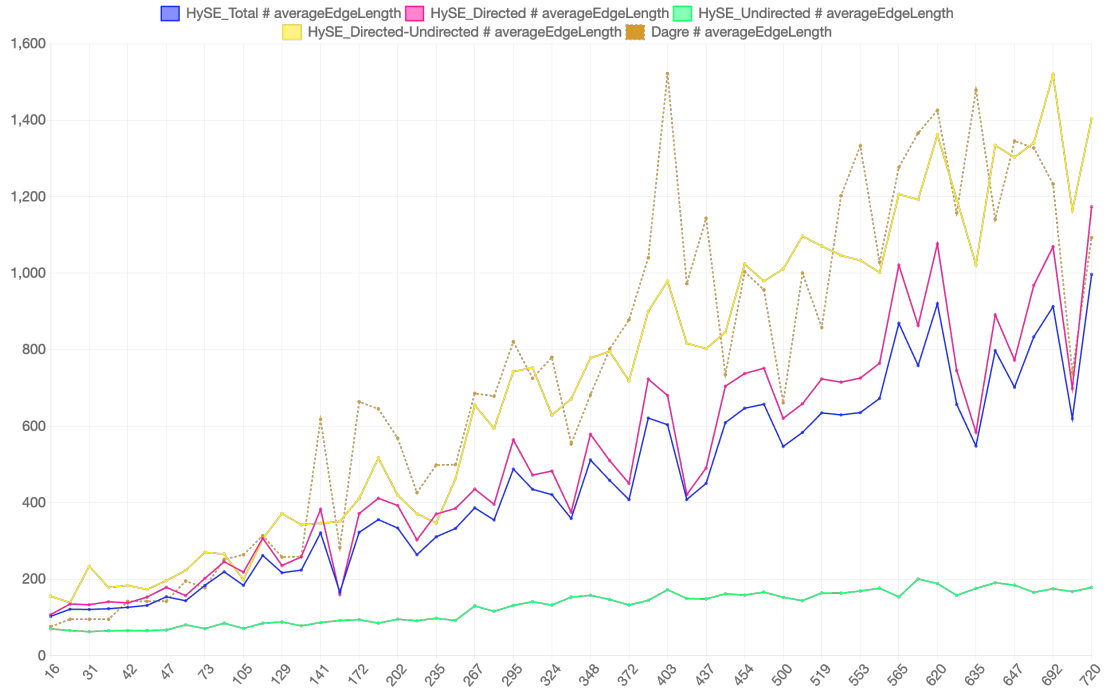



Figure 4.8: Comparison chart of "Average Edge Length" between Dagre and different parts of HySE : 60% undirected nodes of original graph

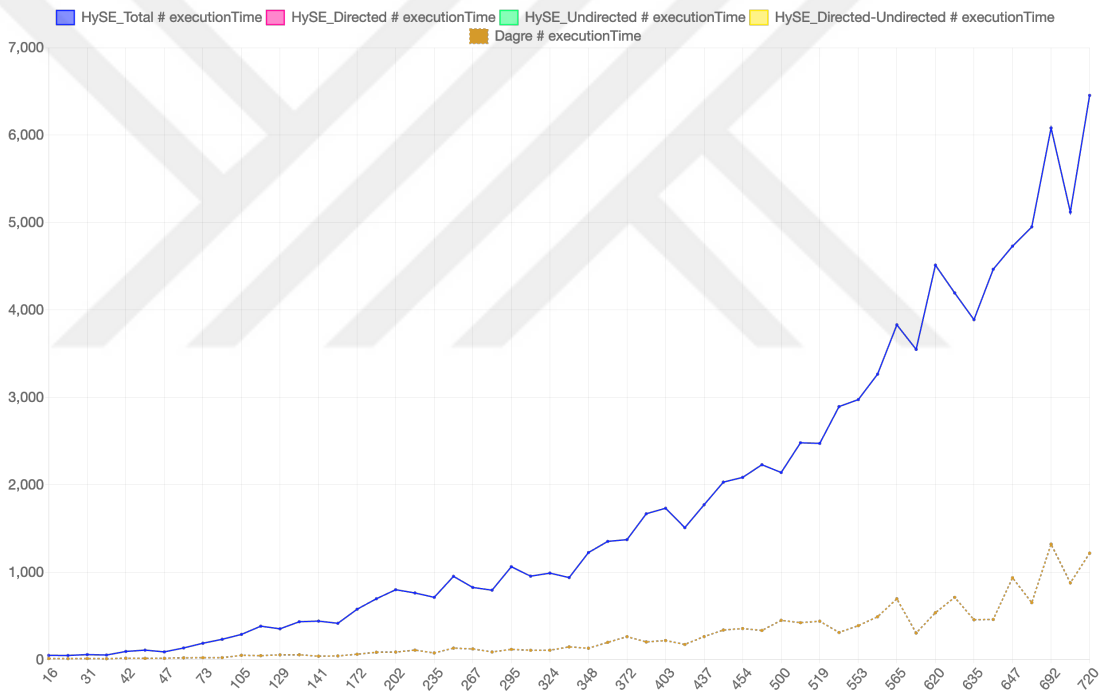



Figure 4.9: Comparison chart of "Execution Time (ms)" between Dagre and HySE : 60% undirected nodes of original graph. Execution time cannot be divided due to the holistic nature of HySE and the results are comparing Dagre on just the directed part of the graph vs HySE on the entire graph.

The above-given experiments were done with undirected nodes that were 60 percent in number compared to the directed part. This means that HySE graphs had 60 percent more nodes than those of Dagre for comparison against these metrics. We collected results for each part of the graph so that we can compare those accordingly. We can see in the results given below, that the number of nodes highly affects the execution time, which is the same for every part of the graph. The runtime complexity of the algorithm is $\mathcal{O}(k \cdot (|V|))$, where $k$ is the number of iterations and highly dependant on $V$, as a graph with a large number of nodes takes more iterations. It can be clearly seen in the "execution time" figures, that the line is neither completely linear nor completely quadratic, so we can assume the quadratic run time to be the worst-case scenario. We performed tests and collected results for different percentages of undirected nodes in our test graphs so that we could evaluate it better against the baseline. It can also be observed that with the high number of undirected nodes, the line denoting the execution time of HySE becomes close to quadratic, whereas, with a smaller number of undirected nodes, it gets close to being linear.

The charts from Figure 4.10 till Figure 4.14 give us the results for the experiments where the number of undirected nodes was 40 percent of the size of the original directed graph.

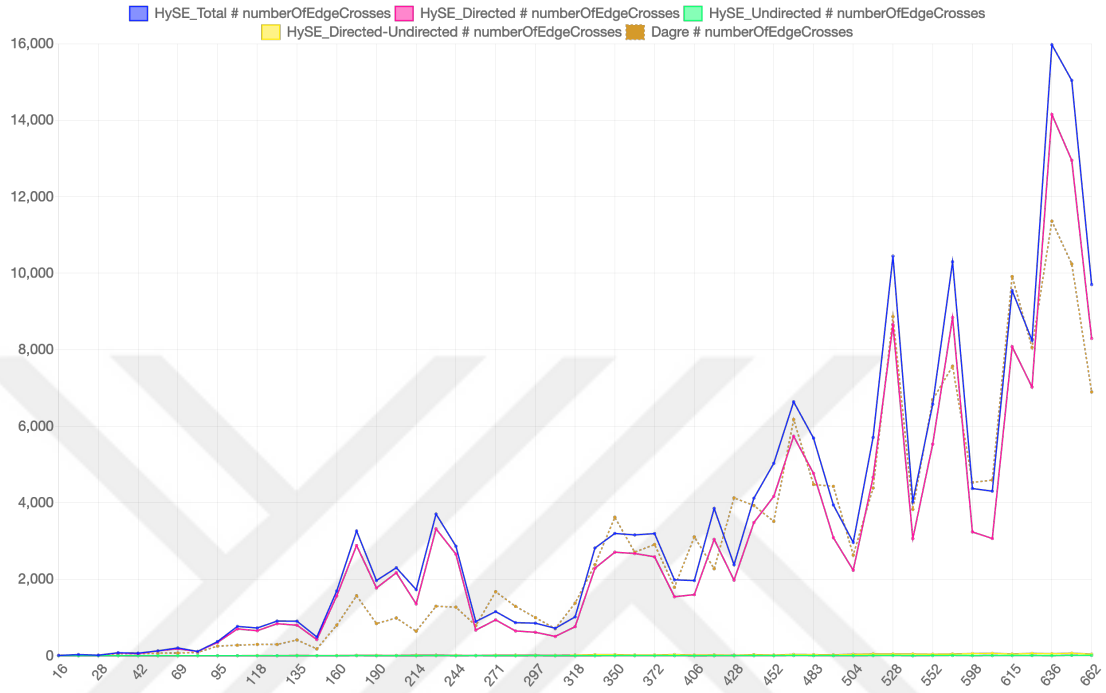



Figure 4.10: Comparison chart of "Number of Edge Crossings" between Dagre and different parts of HySE : 40% undirected nodes of original graph

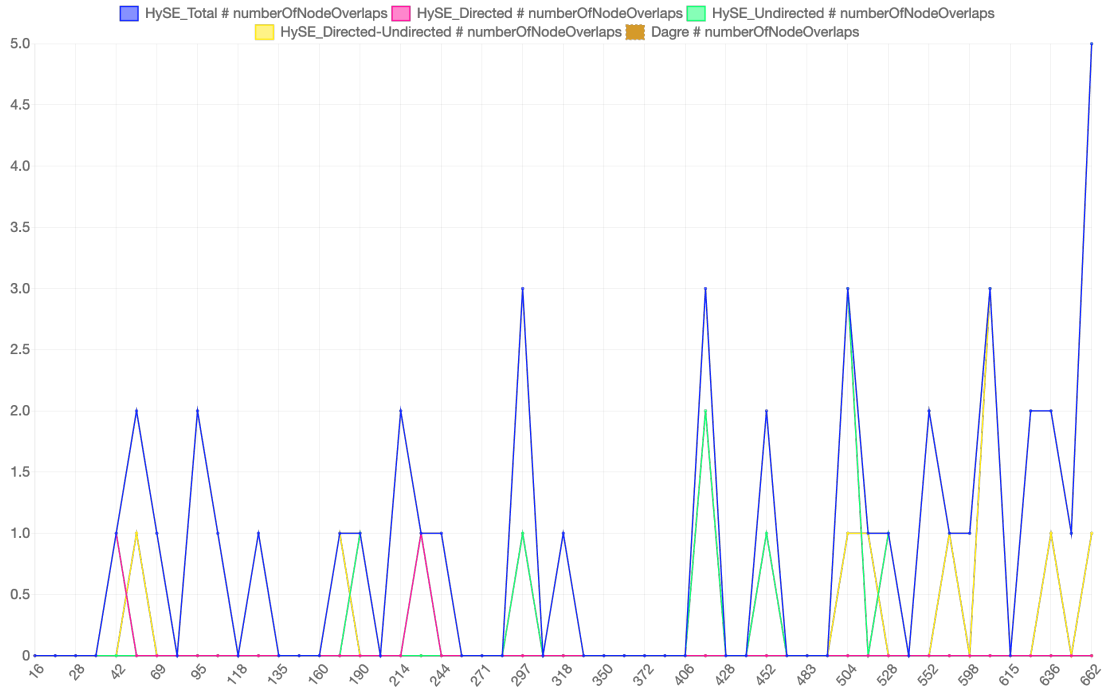



Figure 4.11: Comparison chart of "Number of Node Overlaps" between Dagre and different parts of HySE : 40% undirected nodes of original graph

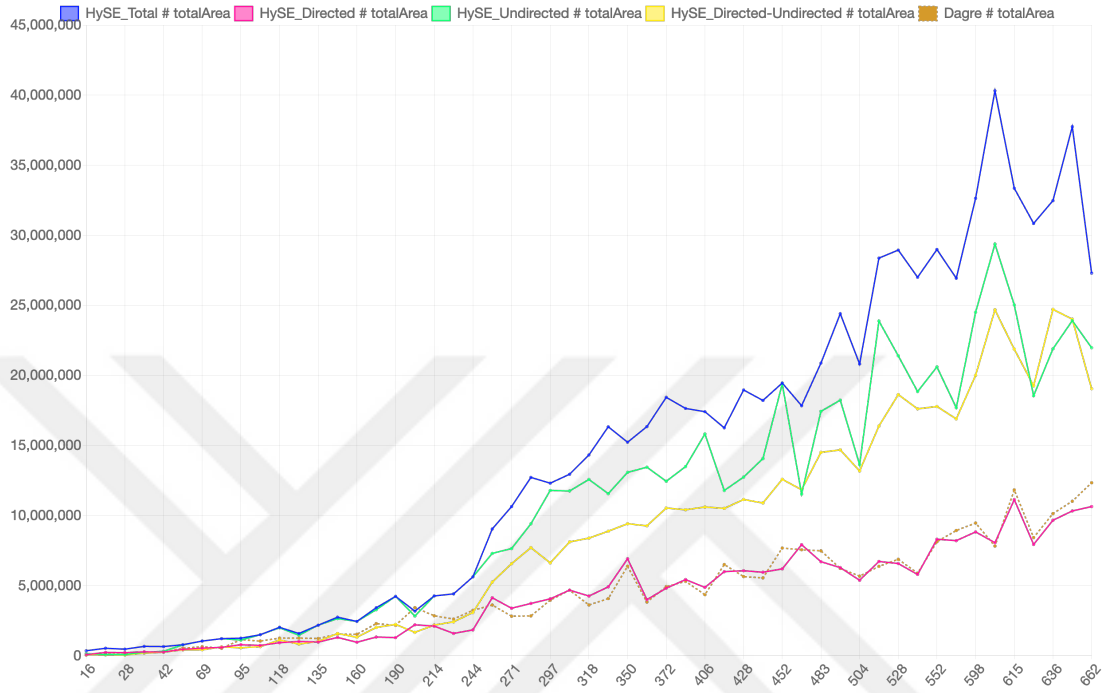



Figure 4.12: Comparison chart of "Total Area" between Dagre and different parts of HySE : 40% undirected nodes of original graph

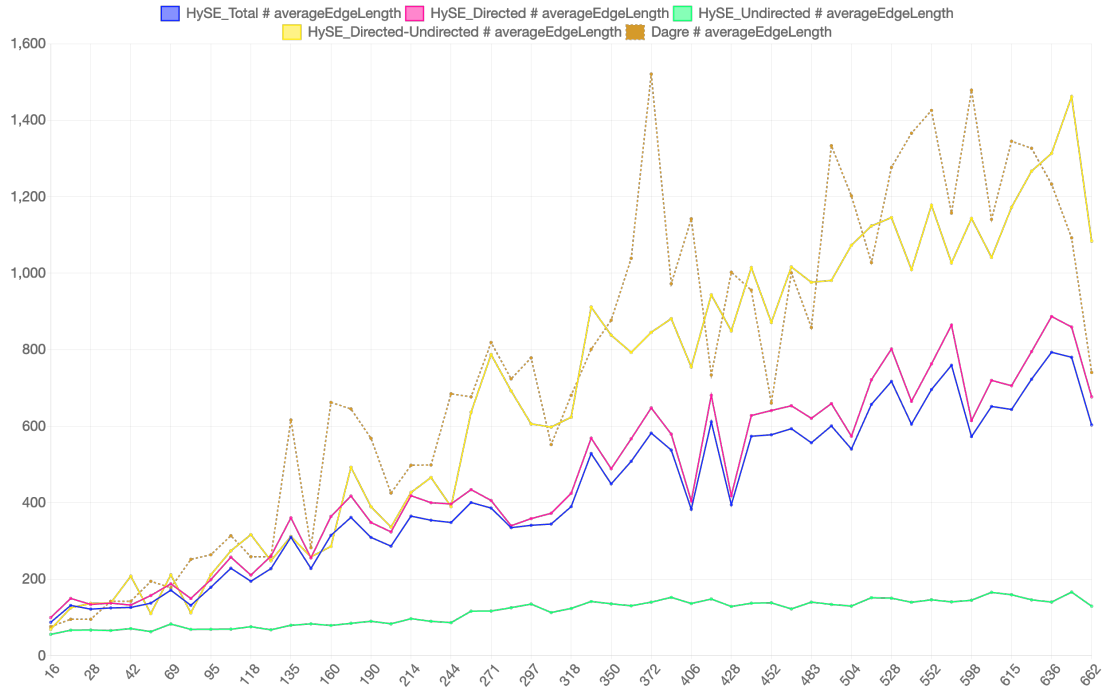



Figure 4.13: Comparison chart of "Average Edge Length" between Dagre and different parts of HySE : 40% undirected nodes of original graph

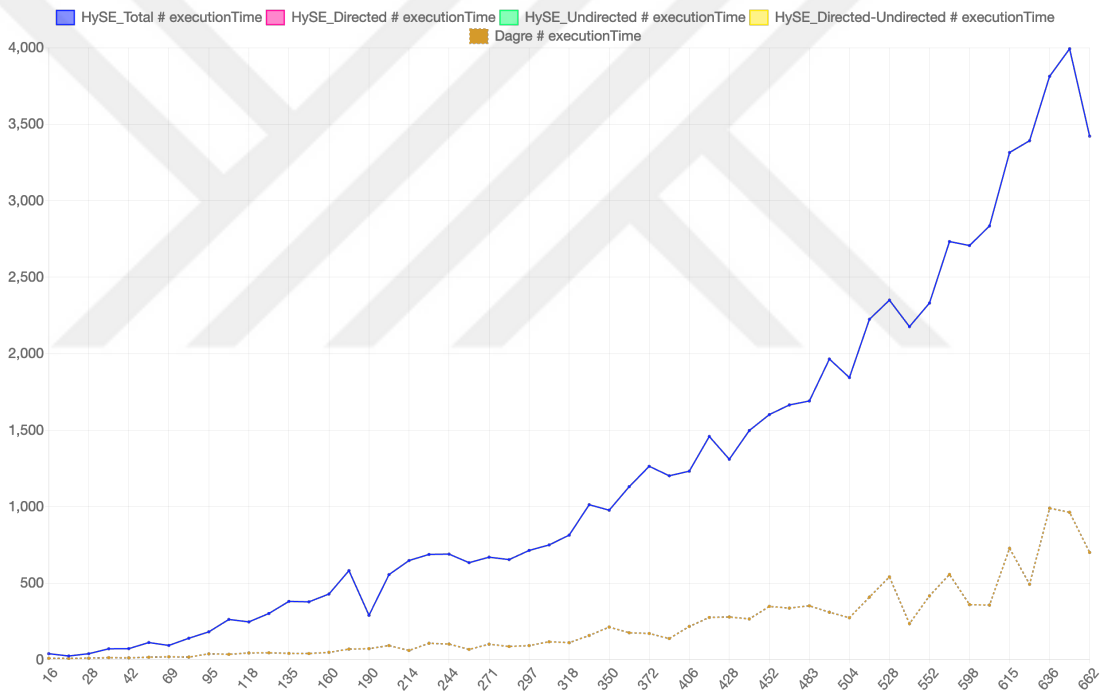



Figure 4.14: Comparison chart of "Execution Time (ms)" between Dagre and HySE : 40% undirected nodes of original graph. Execution time cannot be divided due to the holistic nature of HySE and the results are comparing Dagre on just the directed part of the graph vs HySE on the entire graph.

Figure 4.15 till Figure 4.19 give us the results for the experiments where the number of undirected nodes was 20 percent of the size of the original directed graph. We can clearly observe the execution time getting closer to that of Dagre.

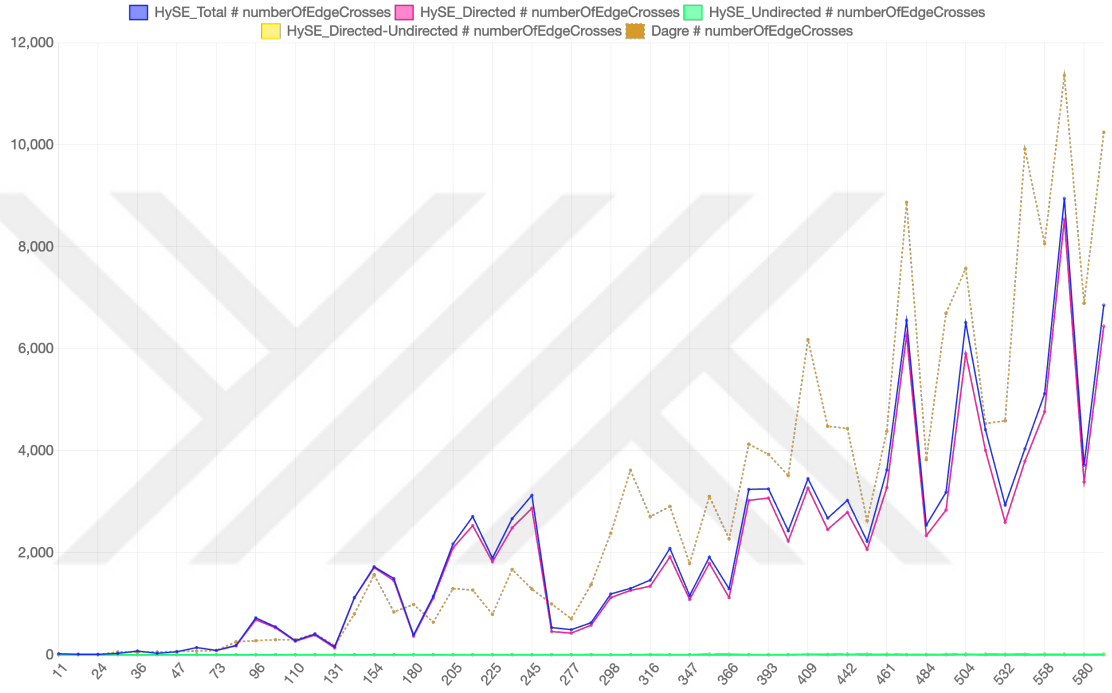



Figure 4.15: Comparison chart of "Number of Edge Crossings" between Dagre and different parts of HySE : 20% undirected nodes of original graph

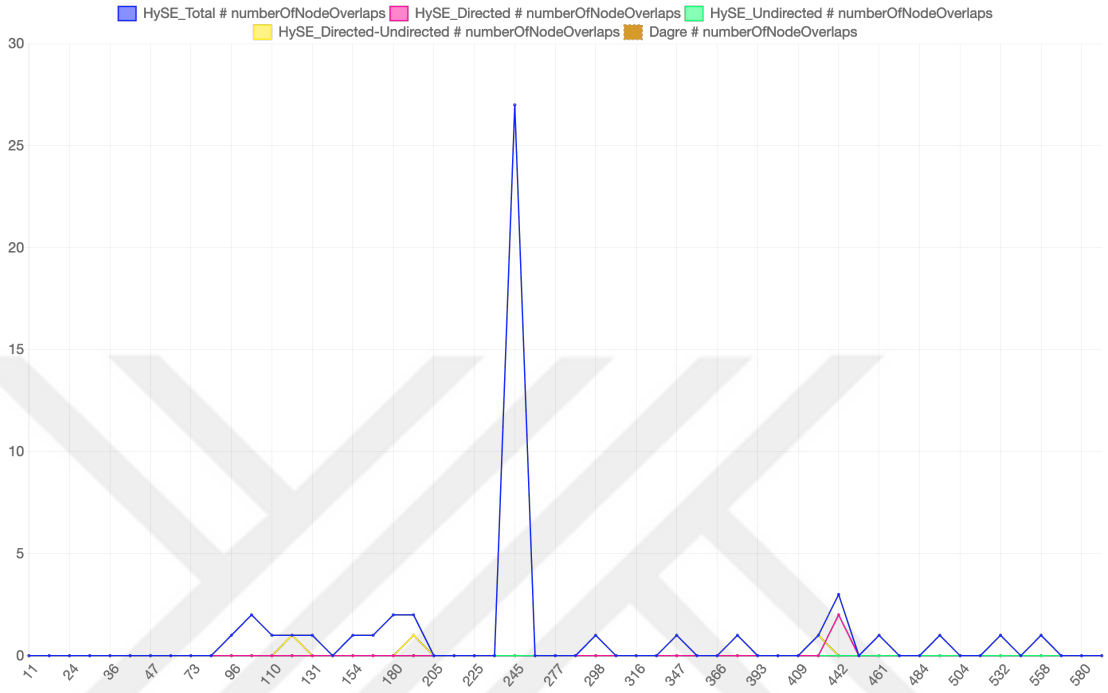



Figure 4.16: Comparison chart of "Number of Node Overlaps" between Dagre and different parts of HySE : 20% undirected nodes of original graph

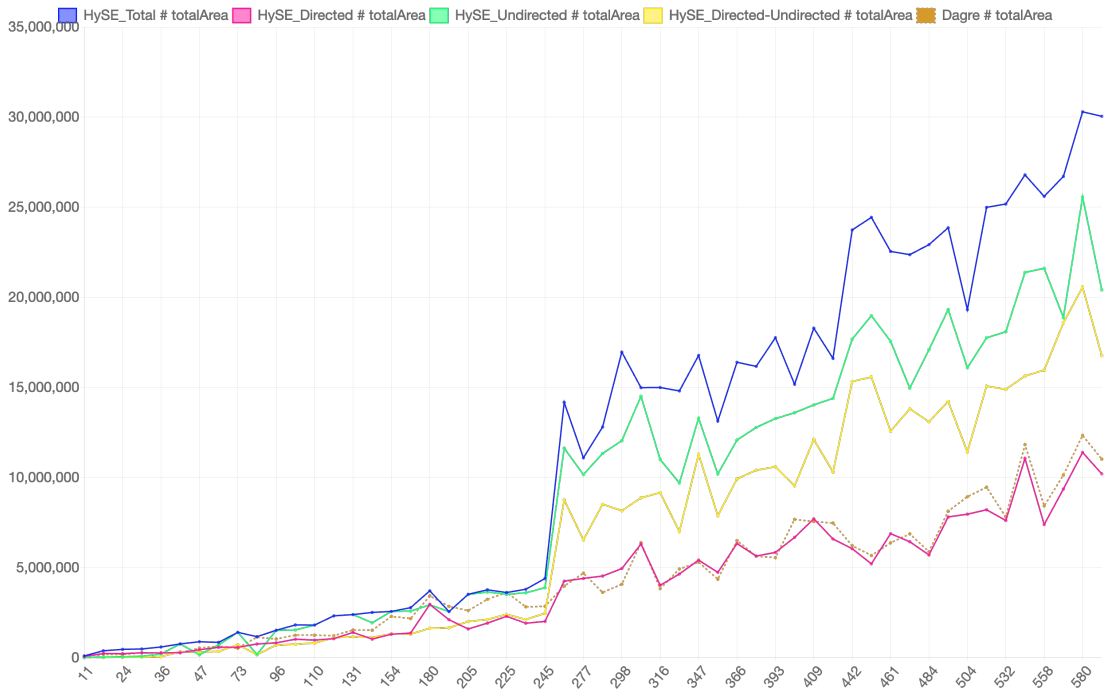



Figure 4.17: Comparison chart of "Total Area" between Dagre and different parts of HySE : 20% undirected nodes of original graph

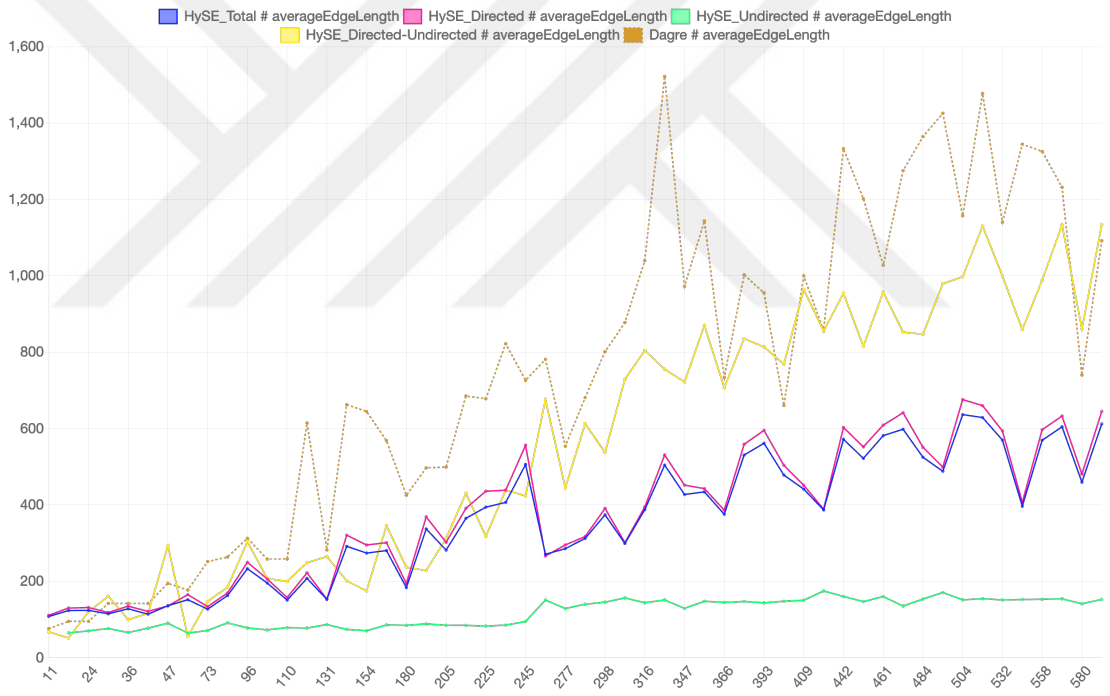



Figure 4.18: Comparison chart of "Average Edge Length" between Dagre and different parts of HySE : 20% undirected nodes of original graph

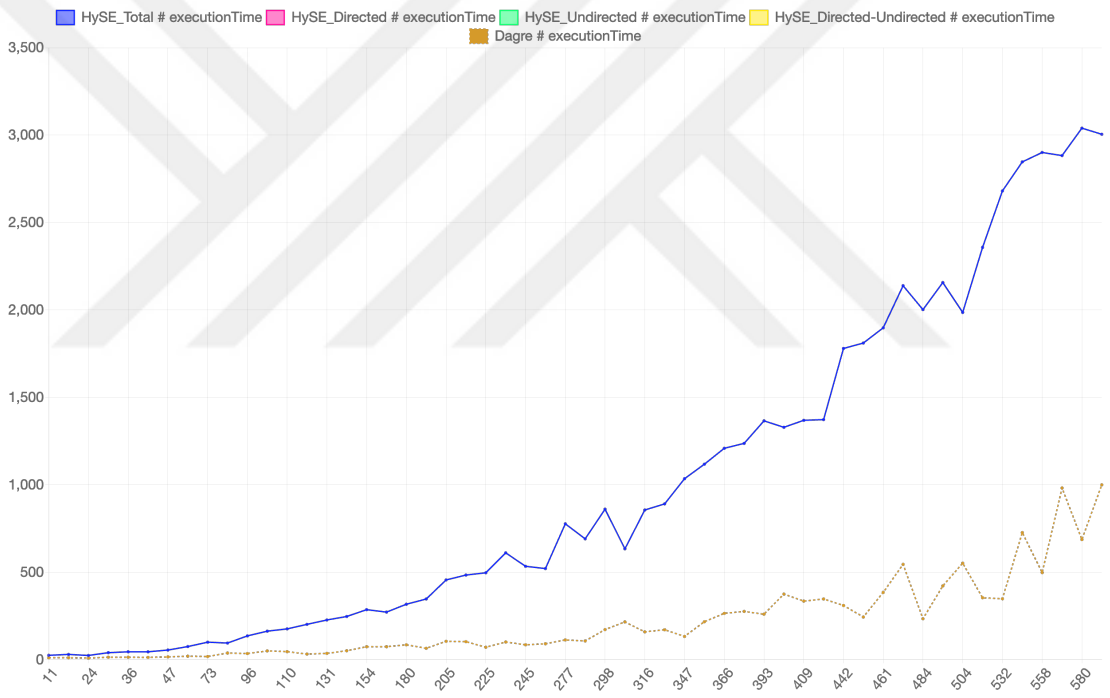



Figure 4.19: Comparison chart of "Execution Time (ms)" between Dagre and HySE : 20% undirected nodes of original graph. Execution time cannot be divided due to the holistic nature of HySE and the results are comparing Dagre on just the directed part of the graph vs HySE on the entire graph.

Figure 4.20 till Figure 4.24 give us the results for the experiments where there were no undirected nodes involved. We can clearly see that the performance improves, but it's the execution time that gets really low in Figure 4.24, almost the same as Dagre's, as there are no undirected components to process. This makes the calculations that are used to check the owner of the graph or the parent of the two nodes irrelevant, as all the nodes belong to the same graph. Furthermore, the grid calculations are only done at regular intervals or only if there are changes in the relative node positions such as node swaps.

So applying these heuristics works for HySE and the execution time decreases drastically and comes close to Dagre's execution time. So, even if we consider HySE just for the directed part, it produces better aesthetics while working in a comparable time as Dagre.

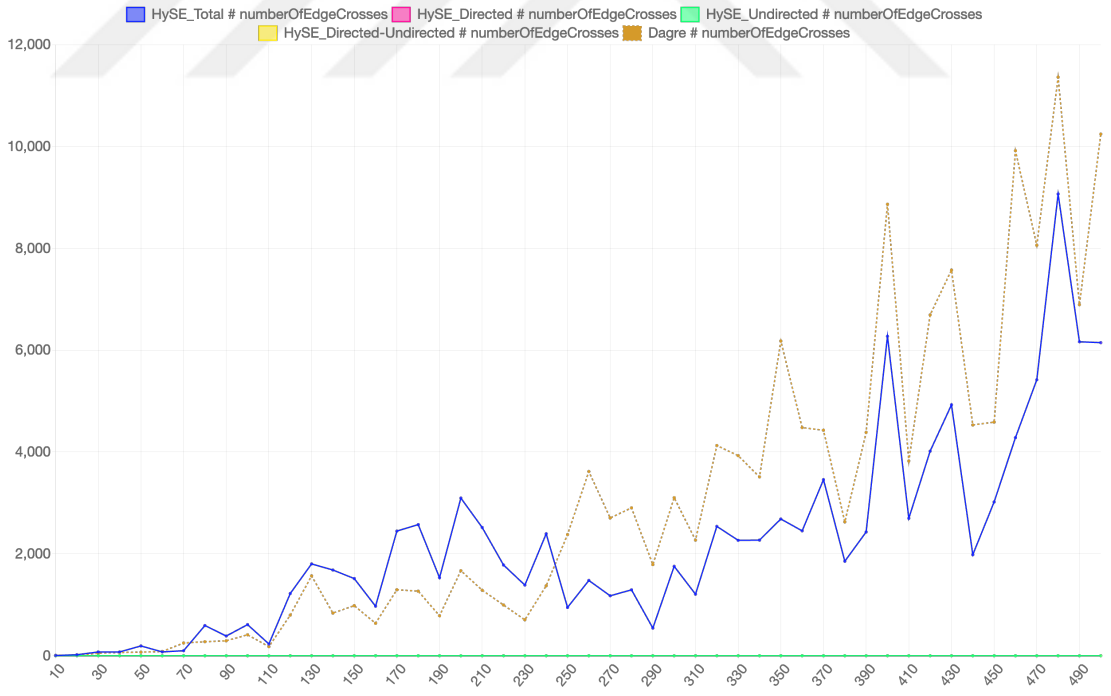



Figure 4.20: Comparison chart of "Number of Edge Crossings" between Dagre and HySE: No undirected nodes in test graph

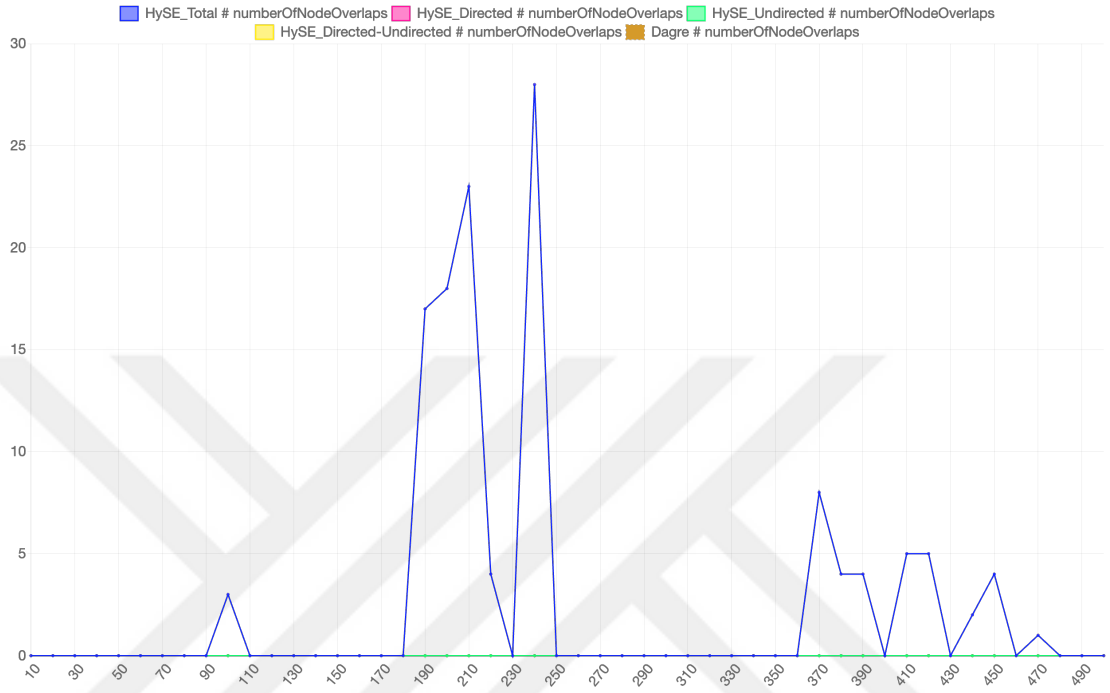



Figure 4.21: Comparison chart of "Number of Node Overlaps" between Dagre and HySE: No undirected nodes in test graph

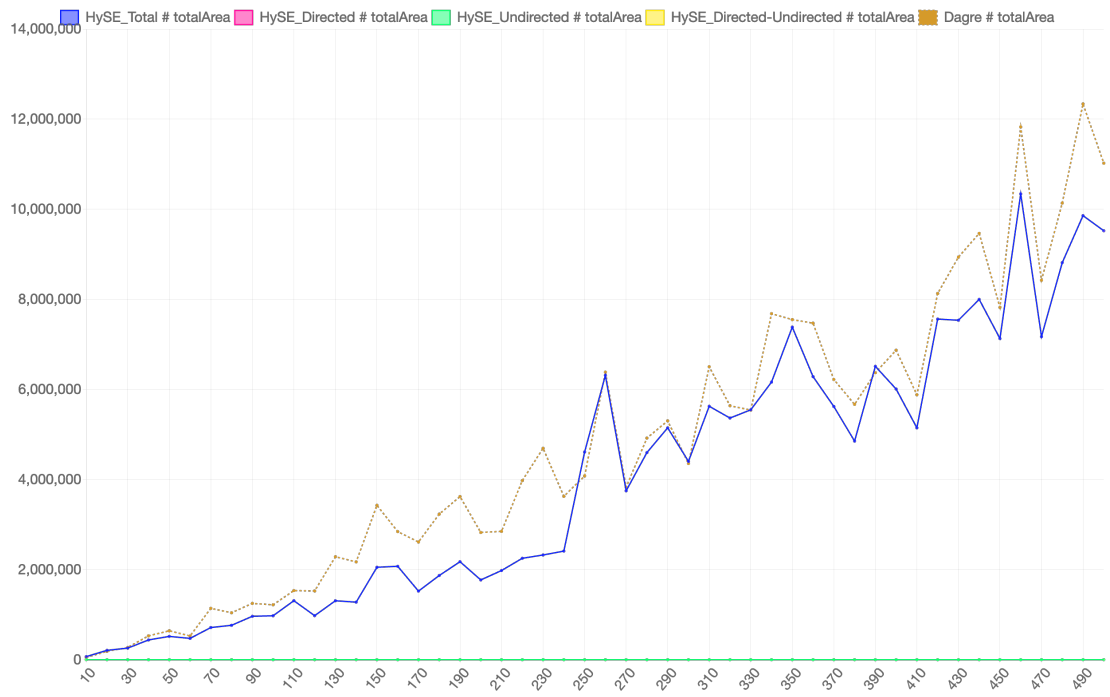



Figure 4.22: Comparison chart of "Total Area" between Dagre and HySE: No undirected nodes in test graph

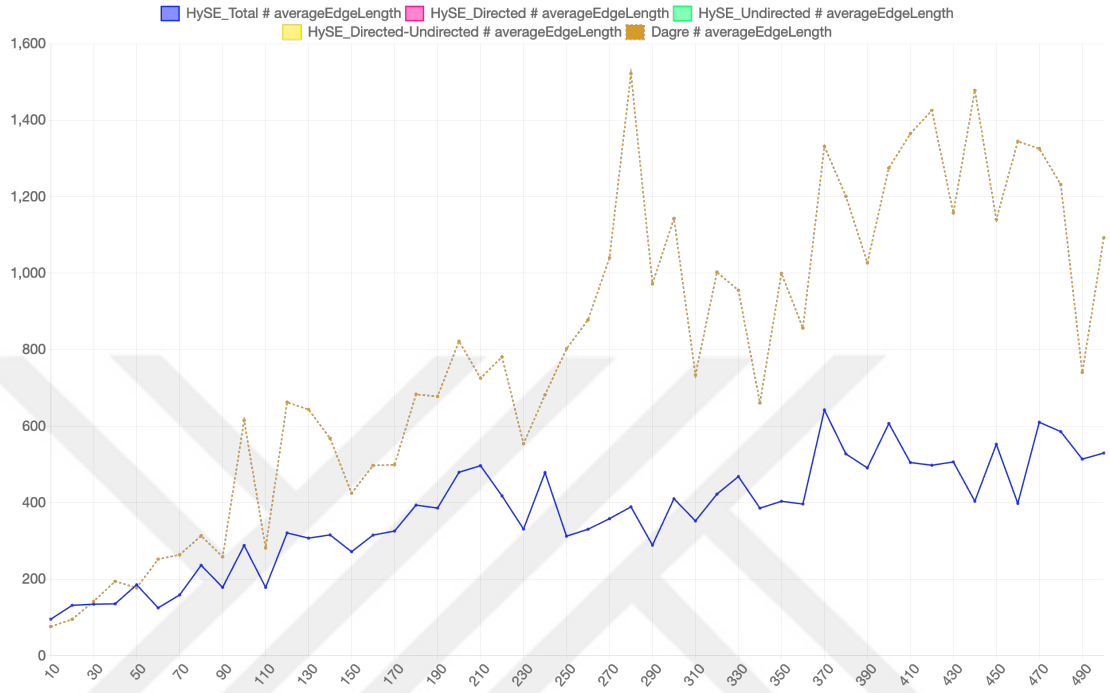



Figure 4.23: Comparison chart of "Average Edge Length" between Dagre and HySE: No undirected nodes in test graph

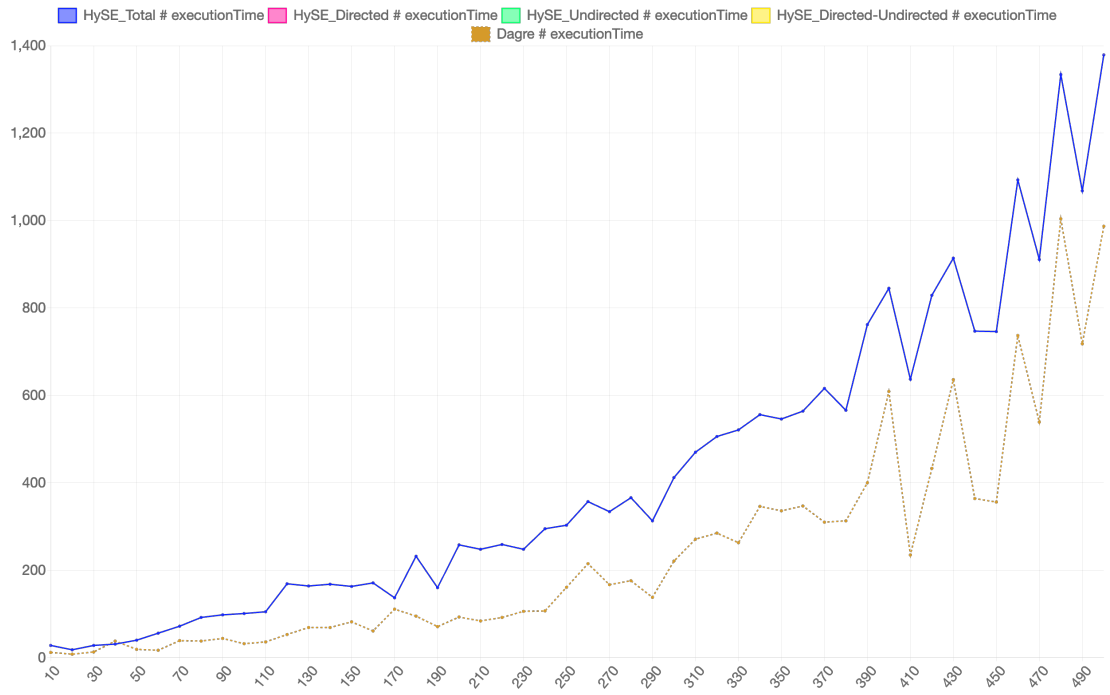



Figure 4.24: Comparison chart of "Execution Time (ms)" between Dagre and HySE: No undirected nodes in test graph.

# Chapter 5

# Conclusion & Future Work

We presented HySE, a new algorithm based on the spring embedder approach to lay out hybrid graphs, in this thesis. Our algorithm first finds some good initial placements for both the directed and undirected nodes before moving on to the force-directed part. This helps in converging the forces of the system in less number of iterations than otherwise. After the initial placements, the force-directed phase enhances the aesthetics by applying spring and repulsion forces. The directed part is constrained to move only along the $x$-axis. Then this phase also reduces the number of edge crossings in the directed part by making use of a swapping heuristic.

Results from our experiments show that HySE produces aesthetically pleasing graph layouts. It lays out graphs with comparable quality metric values as one can get from current implementations of traditional algorithms.

Regarding future work, HySE can be updated to build an incremental version of it, where the layout only polishes the already laid out graph. This can be tricky due to the hierarchical part of the graph, as placing new nodes, trying to maintain the order in layers, and finding positions where edge crossings are minimal can be challenging.

A publicly available and open-source implementation of HySE, used for this

thesis, can be found in its Github repository. The instructions for building and running the demo in the browser are given in the README file.

The directed graphs that we used to run the tests and generate metric results are available here.

# Bibliography


[1] Z. Du, X. Zhou, Y. Ling, Z. Zhang, and Z. Su, "Agrigo: A go analysis toolkit for the agricultural community," *Nucleic acids research*, vol. 38, pp. W64–70, 07 2010.

[2] flexera, "Cloud computing system architecture diagrams." `https://docs.rightscale.com/cm/designers_guide/cm-cloud-computing-system-architecture-diagrams.html`. (accessed: July 2, 2023).

[3] 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.

[4] AllThingsGraphed.com, "How to visualize your facebook friend network." `http://allthingsgraphed.com/2014/08/28/facebook-friends-network/`, 2014. (accessed: July 2, 2023).

[5] 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.

[6] Cytoscape.js, "Using layouts." `https://blog.js.cytoscape.org/2020/05/11/layouts`. (accessed: July 23, 2023).

[7] N. S. Nikolov, *Sugiyama Algorithm*, pp. 2162–2166. New York, NY: Springer New York, 2016.

[8] dagrejs, "Dagre." `https://github.com/dagrejs/dagre`. Accessed Sep. 1, 2021.

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

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

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

[12] E. Gansner, E. Koutsofios, S. North, and K.-P. Vo, "A technique for drawing directed graphs," *IEEE Transactions on Software Engineering*, vol. 19, no. 3, pp. 214–230, 1993.

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

[14] U. Dogrusoz, E. Giral, A. Cetintas, A. Civril, and E. Demir, "A layout algorithm for undirected compound graphs," *Information Sciences*, vol. 179, no. 7, pp. 980–994, 2009.

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

[16] 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.

[17] 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.

[18] K. Sugiyama, S. Tagawa, and M. Toda, "Methods for visual understanding of hierarchical system structures," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 11, no. 2, pp. 109–125, 1981.

[19] M. Jünger and P. Mutzel, "2-layer straightline crossing minimization: Performance of exact and heuristic algorithms," *Journal of Graph Algorithms and Applications*, vol. 1, 01 1997.

[20] U. Brandes and B. Köpf, "Fast and simple horizontal coordinate assignment," in *International Symposium Graph Drawing and Network Visualization*, 2001.

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

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

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

[24] K. Sugiyama and K. Misue, "Visualization of structural information: automatic drawing of compound digraphs," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 21, no. 4, pp. 876–892, 1991.

[25] G. Sander, "Layout of compound directed graphs," Tech. Rep. A/03/96, University of Saarlandes, CS Dept., Saarbrücken, Germany, 1996.

[26] P. Eades, Q.-W. Feng, and X. Lin, "Straight-line drawing algorithms for hierarchical graphs and clustered graphs," *Algorithmica*, vol. 44, pp. 1–32, 2006.

[27] B. Genc and U. Dogrusoz, "A constrained, force-directed layout algorithm for biological pathways," in *Graph Drawing* (G. Liotta, ed.), (Berlin, Heidelberg), pp. 314–319, Springer Berlin Heidelberg, 2004.

[28] Y. Zhang and A. Pang, "Graph layout with boundary constraints," *Journal of Graph Algorithms and Applications*, vol. 20, pp. 435–459, 07 2016.

[29] T. Dwyer and Y. Koren, "Dig-cola: directed graph layout through constrained energy minimization," in *IEEE Symposium on Information Visualization, 2005. INFOVIS 2005.*, pp. 65–72, 2005.

[30] K. Sugiyama and K. Misue, "A simple and unified method for drawing graphs: Magnetic-spring algorithm," in *Graph Drawing* (R. Tamassia and I. G. Tollis, eds.), (Berlin, Heidelberg), pp. 364–375, Springer Berlin Heidelberg, 1995.

[31] D. Holten, "Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data," *IEEE Transactions on Visualization and Computer Graphics*, vol. 12, no. 5, pp. 741–748, 2006.

[32] P. Eades, X. Lin, and W. Smyth, "A fast and effective heuristic for the feedback arc set problem," *Information Processing Letters*, vol. 47, no. 6, pp. 319–323, 1993.

[33] F.-J. Brandenburg and K. Hanauer, "Sorting heuristics for the feedback arc set problem," Tech. Rep. MIP-1104, Department of Informatics and Mathematics, University of Passau, 2011.

[34] V. Geladaris, P. Lionakis, and I. G. Tollis, "Computing a feedback arc set using pagerank," in *Graph Drawing and Network Visualization* (P. Angelini and R. von Hanxleden, eds.), (Cham), pp. 188–200, Springer International Publishing, 2023.

[35] U. Dogrusoz, A. Cetintas, E. Demir, and O. Babur, "Algorithms for effective querying of compound graph-based pathway databases," *BMC Bioinformatics*, vol. 10, p. 376, Nov 2009.

[36] P. Eades, X. Lin, and W. Smyth, "A fast and effective heuristic for the feedback arc set problem," *Information Processing Letters*, vol. 47, no. 6, pp. 319–323, 1993.

[37] M. Franz, C. T. Lopes, G. Huck, Y. Dong, O. Sumer, and G. D. Bader, "Cytoscape.js: Graph theory (network) library for visualisation and analysis." `"https://js.cytoscape.org/"`. (accessed: July 15, 2023).

[38] iVis, "cose-base." `https://github.com/iVis-at-Bilkent/cose-base`. (accessed: Aug 21, 2022).

[39] S. Bridgeman, "GD Toolkit RND/BU4P." http://www.dia.uniroma3.it/~gdt/gdt4/test_suite.php. (accessed: June 15, 2023).

[40] 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.