AN ORTHOGONAL LAYOUT ALGORITHM FOR SMALL COMPOUND 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
Mubashira Zaman
September 2021

AN ORTHOGONAL LAYOUT ALGORITHM FOR SMALL COMPOUND GRAPHS

By Mubashira Zaman

September 2021

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)

Can Alkan

Tolga Can

Approved for the Graduate School of Engineering and Science:

Ezhan Karaşan
Director of the Graduate School

ABSTRACT

AN ORTHOGONAL LAYOUT ALGORITHM FOR SMALL COMPOUND GRAPHS

Mubashira Zaman
M.S. in Computer Engineering
Advisor: Uğur Doğrusöz
September 2021

Information visualization is the study of different approaches that aid in the visualization and examination of data. Among the broad variety of different options and techniques available in this field is "Graph Drawing", which is regarded as the algorithmic foundation of relational information or graph visualization. Graph drawing fuses graph theory and visualization for presenting data as geometric shapes and for laying them out in a 2-D or 3-D space. There exist many different types of automatic graph layouts. One such layout is the orthogonal graph layout in which edges are made up of horizontal and vertical segments.

A specialized version of graphs called compound graphs are used to represent grouping or clustering of graph objects. Many orthogonal layout approaches have been presented for simple graphs but there is considerably less research available for orthogonal layout algorithms for compound graphs. In this thesis, we present C-TSM, which takes the already existing Topology-Shape-Metrics (TSM) approach and extends it to cater to 4-degree small compound graphs with uniform node sizes. First, compound graphs are converted to simple graphs and then the TSM approach is applied to it. The resulting output is compacted again in a post-processing step and then the graph is converted back to a compound graph. The results of performance tests on our algorithm show that C-TSM works considerably well on small-sized graphs and gives the output in up to a few seconds. This algorithm has been implemented in Javascript and Python and is available as a Cytoscape.js extension. The source code and a demo application are available on a GitHub repository.

Keywords: Information Visualization, Orthogonal graph layout, Automatic Layout Algorithm, Compound Graphs.

ÖZET

KÜÇÜK BİLEŞİK ÇİZGELER İÇİN DİKEY DÜZEN YERLEŞTİRME ALGORİTMASI

Mubashira Zaman
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Danışmanı: Uğur Doğrusöz
Eylül 2021

Bilgi görselleştirme, verinin görselleştirmesine ve incelenmesine yardımcı olan farklı yaklaşımların incelenmesidir. Bu alanda bulunan çok sayıda seçenek ve yöntem arasında ilişkisel verinin veya çizge görselleştirmenin algoritmik temeli olan "çizge çizimi" yer almaktadır. Çizge çizimi, veriyi geometrik şekiller ile sunmak ve 2 boyutlu veya 3 boyutlu bir alanda sergilemek için çizge teorisini ve görselleştirmeyi birleştirmektedir. Çok sayıda otomatik çizge düzeni bulunmaktadır. Bu düzenlerden biri kenarları dikey ve yatay bölümlerden oluşan ortogonal düzendir.

Çizge nesnelerinin gruplandırılmasını veya kümelenmesini temsil etmek için bileşik çizge (compound graph) adı verilen özelleştirilmiş bir çizge türü kullanılmaktadır. Basit çizgeler için çok sayıda ortogonal düzen yaklaşımı sunulmuştur fakat bileşik çizgeler için ortogonal yerleştirme algoritması ile ilgili az sayıda araştırma mevcuttur. Bu tezde, halihazırda var olan Topoloji-Şekil-Metrik (Topology-Shape-Metrics (TSM)) yaklaşımını alıp tek tip düğüm büyüklüğüne sahip 4 dereceli küçük bileşik çizgelere hitap edecek şekilde genişleten C-TSM yaklaşımını sunuyoruz. İlk olarak, bileşik çizgeler basit çizgelere dönüştürülüyor ve ardından bu çizgeye değiştirilmiş TSM yaklaşımı uygulanıyor. Ortaya çıkan çıktı, bir sonradan işleme adımında bir araya toplanıyor ve ardından tekrar bir bileşik çizgeye çevriliyor. Algoritmamız üzerinde yaptığımız performans testlerinin sonuçları C-TSM yaklaşımının küçük boyutlu çizgelerde oldukça iyi çalıştığını ve çıktıyı birkaç saniye içinde verdiğini gösteriyor. Bu algoritma Javascript ve Python'da geliştirilmiştir ve Cytoscape.js uzantısı olarak mevcuttur. Algoritmanın kaynak kodu ve demo uygulaması GitHub'da bulunmaktadır.

Anahtar sözcükler: Bilgi Görselleştirme, Ortogonal Çizge Düzeni, Otomatik Yerleştirme Algoritması, Bileşik Çizgeler.

Acknowledgement

I would like to extend my sincere gratitude and appreciation to my supervisor Prof. Dr. Uğur Doğrusöz for his constant guidance and patience during my research. Without his persistant help and effort, this thesis would not have been possible.

I am thankful to my parents for their emotional support and for always having my back, and to the friends in Bilkent who stood by me in my good and bad days.

I would also like to acknowledge the suggestions of my thesis committee members, who reviewed my work and provided valuable feedback on how to improve it.

Lastly, I express my gratitude to the Scientific and Technological Research Council of Turkey, TÜBİTAK (grant number 118E131) for their extensive support during my years in M.Sc. study.

Contents

List of Figures

Chapter 1

Introduction

1.1 Information Visualization

With its involvement in actions taking place in the outer world, the human mind accumulates a great deal of knowledge. The limitations of what humans know, pay attention to, and reason with, severely limit their mental capacity. To compensate, people devised novel ways and devices that aid in the enhancement of their cognition beyond its normal state, and this ability is perhaps the most powerful tool for improving cognitive function [1]. Visualization of data is one of those powerful tools.

As the saying goes, "a picture is worth a thousand words". Graphical depictions aid in strengthening human memory by generating easily understandable patterns. Data visualization makes it simple to explore large volumes of data. They not only produce an abstract picture of situations, but they also assist in maintaining track of multiple things simultaneously.

Every day, vast amounts of data are created, processed, and stored; attributable to technological advancements. It was estimated that at the start of 2020, the total amount of data in the world was approximately in zettabytes [2].

Such an enormous amount of data necessitates tools that aid in its precise and efficient analysis. Performing analysis on such massive scale information reservoirs has become quite easier with the advent of graphical analysis tools. These technologies filter and organize raw data and transform it into intelligible visual forms such that the underlying information is revealed.

1.2 Graph Drawing

There are numerous types of information visualization techniques, each of which is distinct, but all of which aid in the easy perception and understanding of data. Some of these types are tree mapping, multidimensional scaling, concept mapping, dendrogram, cartogram, etc. Graphs are defined as abstract mathematical constructs that are used to depict relationships between different elements. They are used to represent relational data and to map their connections as entity-relationship diagrams. The objects in a graph are termed as *nodes* and the links are called *edges*. The edges between nodes might be straight, curved, or made up of many segments, while the nodes themselves can be of various forms and sizes.

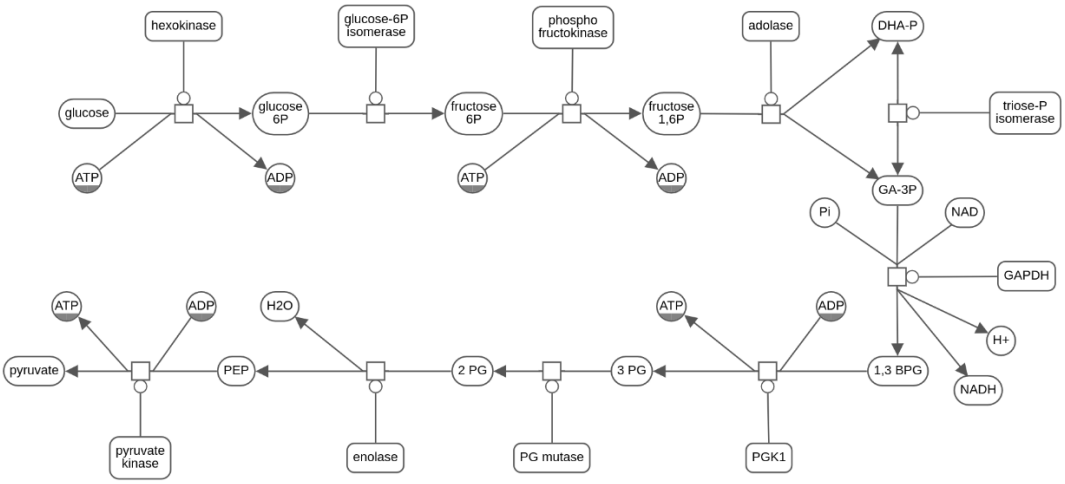


Figure 1.1: Graphical representation of the Glycolysis pathway [3]

A *drawing* or *layout* of a graph is a pictorial representation of its vertices and edges. Graphs can be used to represent many different types of data. They can be used to model social networks and to figure out how their users are connected

to each other. They can also be used to create VLSI (very large-scale integration) designs and to design layouts for PCBs and electrical components. Bioinformatics also widely uses graphs to depict the interactions between different biological components. Figure 1.1 depicts the overview of the glycolysis pathway by means of a graphical representation. Entity modeling in databases is another application of graphs.

1.3 Graph Layouts

Placing nodes and drawing edges in such a way that the user is easily able to extract and understand relations between the objects, is the main focus in creating a layout. In short, the *readability* of the graph is the main goal of this problem. If the objects linked to each other are too far apart, it will be hard to find the relationships between them. If the links are too long, the graph will expand and the area will increase, which will again make it harder to focus on the graph. These factors have to be kept in mind while creating a layout algorithm. Some established performance criteria for graph drawings are overall area, average edge length, node-to-node overlaps, edge-to-edge overlaps, symmetry, and the number of edge crossings.

Hand-drawn graphs have the potential to be the most aesthetically pleasing graphs. But unfortunately, this task is easier only for smaller graphs. As the sizes of the graphs grow bigger, so does the difficulty of the drawing, and it becomes increasingly hard to decide a place for the next node. It requires not just a lot of time, but also a lot of effort. As the technology evolved over time, the task of drawing graphs was also handed over to the machines and they were programmed to produce the best possible graph drawings. There now exist many graphic visual tools that create automated layouts for data analysis. Circular, orthogonal, force-directed, and spectral layouts are among the various types of available graph sketching layouts. Figure 1.2 shows an example of automatic circular spring embedder layout (CiSE) [4] on clustered data.

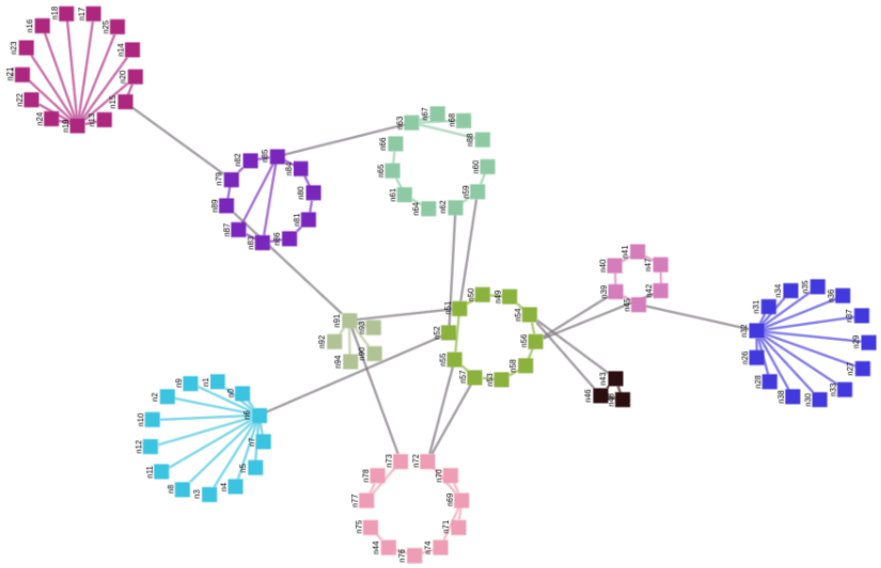


Figure 1.2: CiSE (Circular spring embedder) layout [4] on a clustered graph

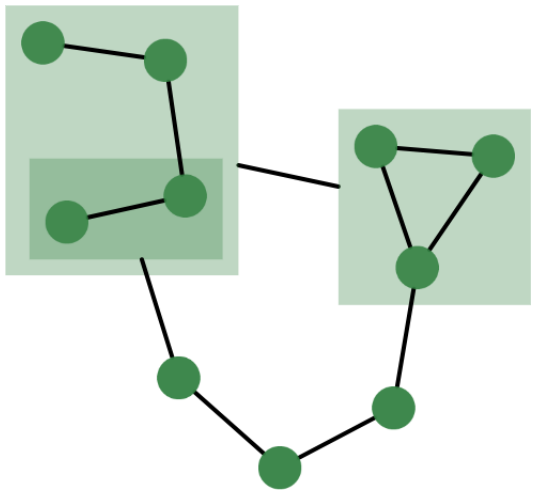


Figure 1.3: An example of a compound graph with inter-graph edges

1.4 Compound Graphs

Compound graphs are graphs with multiple levels of nesting (i.e. graph within a graph) and they contain inter-graph edges. Figure 1.3 depicts an example of a compound graph. In the graph layout domain, there has been a lot of research on general graphs or *simple graphs* [5]; however, there is relatively little research done on compound graphs [6]. Similarly, while there exist abundant studies on orthogonal layout algorithms for simple graphs; there is no such algorithm that is specialized for compound graphs.

1.5 Contributions of this Thesis

There exist several orthogonal layout strategies for simple graphs. As for compound graphs, there are some commercial products [7, 8] that address compound graphs in terms of this layout, but there is no public or published research work addressing compound graphs fully. Apart from this, some research works discuss orthogonal layouts for layered graphs [9, 10, 11], but they either do not support general compound graphs and/or are limited to specific types of graphs such as signaling pathways. For instance, the work by Siebenhaller et al. [11] assumes a sub-set of compound graphs (i.e., molecular complexes that do not admit inter-graph edges and compartments that do not admit edges) applicable to signaling pathways. In this thesis, we present an approach for performing orthogonal layout on general compound graphs.

Originally, the Human-Like-Orthogonal-Layout algorithm (HOLA) [12] was employed to perform the orthogonal layout on compounds, however, our implementation did not perform well enough to meet the desired performance criteria. So, for solving this problem, the baseline technique was switched to the Topology-Shape-Metrics [13] approach.

The orthogonal layout described in this thesis is achieved by reducing an input compound graph into a simple graph, to which the TSM layout is applied. The

output from the TSM approach is then postprocessed and converted back into a compound graph. The results of this work suggest that this method works well for small compound graphs.

1.6 Organization of this Thesis

The remaining sections of the thesis are organized as follows:

Chapter 2 explains the background information on graph theory and graph layouts. It then proceeds to explain the basic terminologies that have been used in this thesis. After that, the performance criteria of graph layouts are mentioned. Next, the related research work on compound graph layouts and orthogonal layouts is discussed. In the end, it goes on to provide some insight into Cytoscape.js; the platform on which this algorithm has been developed.

Chapter 3 firstly mentions the prerequisites for our layout approach. It then summarizes the Topology-Shape-Metrics approach that is the basis of our algorithm. It then explains our proposed algorithm *Compound-TSM* that extends and builds on the existing TSM approach to present an orthogonal layout algorithm for compound graphs. The limitations of this algorithm are also mentioned at the end.

Chapter 4 initially discusses the implementation of the C-TSM approach. It proceeds to analyze the performance of our algorithm against a compound graph layout approach. Furthermore, the results of the evaluation are discussed, based on the chosen performance criteria. The chapter also mentions the limitations of the algorithm.

Chapter 5 summarizes this thesis and discusses the possible future work.

Chapter 2

Background and Related Work

This chapter provides the background information related to this research work. Initially, some insight is offered on graphs and graph layouts and then, the common terminologies used in this work are discussed. After that, we mention the performance criteria for graph layouts. Next, the existing compound graph layout algorithms and some orthogonal layout approaches are discussed. Finally, Cytoscape.js and its extensions are mentioned.

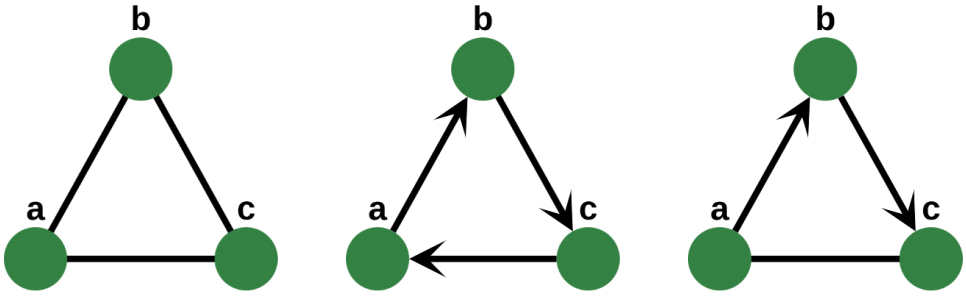
2.1 Graphs

Graphs are node-and-edge-based mathematical data structures that are used to depict relationships between entities. *Nodes*, also known as points or vertices, hold data components, whereas *edges*, also known as arcs or links, refer to the interactions between nodes. Graphs are commonly used in a variety of fields. Among other applications, they are used to create models of social, biological, and circuit networks.

In mathematical terms, a graph is denoted as $G = (V,\ E)$, where V is the set of vertices in the graph and $E = \{\{u,\ v\} \mid u, v \in V\}$ is a set containing pairs of nodes that are connected to each other via edges. Each node in a graph can be

represented by a structure having multiple attributes that store necessary data.

Edges can be classified into two types based on their direction. The first are undirected or unordered edges, which connect vertices asymmetrically. These edges can be traversed in both directions, indicating a two-way link. The second type of edges are directed or ordered edges that originate from a source and point towards a target. Based on these edge types, graphs can be classified into three different types: undirected graphs, directed graphs, and mixed graphs. Let us take an example of a graph $G = (V, E)$ where $V = \{a, b, c\}$ and $E = \{\{a, b\}, \{b, c\}, \{c, a\}\}$. Figure 2.1 depicts this graph in the context of different types of edge settings.



(a) Undirected graph (b) Directed graph (c) Mixed Graph

Figure 2.1: Different types of graphs based on the edge directions

2.2 Graph Layouts

Graphs can be visualized by many different types of drawings or layouts, either in two dimensions or three dimensions. Although 3-D drawings can give a better idea of a graph's structure, they are not that common for practical reasons. Mostly, node-link graph drawings in a 2-D plane are used for better visualization and analysis of data.

Nodes of a graph can be of various shapes and sizes. They are mostly drawn as geometrical shapes like circles, rectangles, triangles, and rhombus, whereas edges can be straight, curved, or composed of horizontal and vertical segments. If edges are directed, they can have arrows pointing from source to target. The

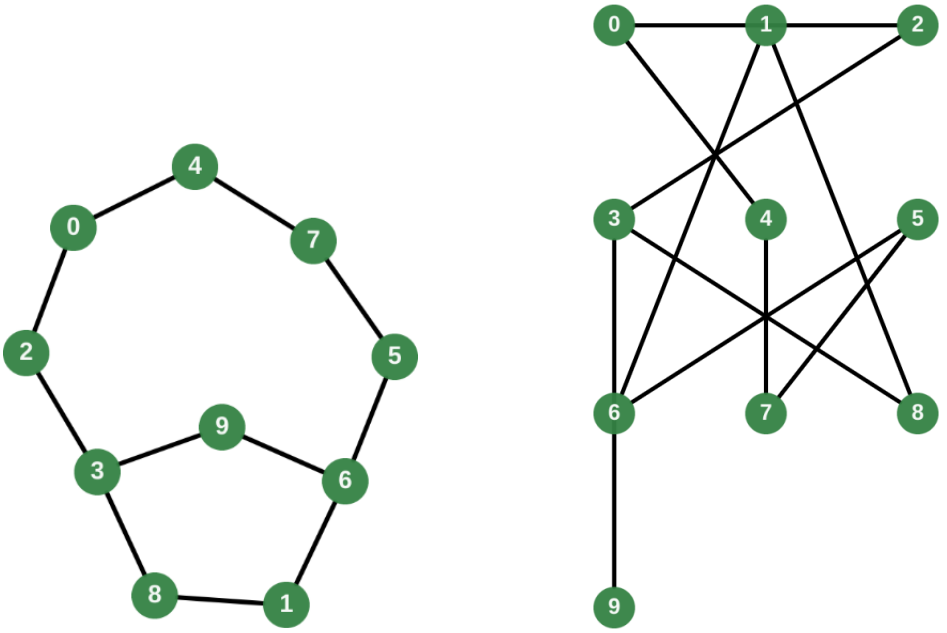
overall visual representation of the graph should be such that it clearly conveys the relationships between different elements and makes the comprehension of the scenario easier for the user. Different graph properties such as sizes of the nodes, thickness of the edges, placement of the labels, and color schemes of the different elements, are all some examples that greatly affect the overall outlook of the graph. These properties are to be chosen carefully so that they add to the visual design instead of making it worse.

Research of graph layouts is mainly focused on designing automatic graph layouts and interactive graphics applications. The required output of a graph layout is to determine the final positions of nodes and edges such that the underlying information is clearly revealed. The positions of the nodes make a notable difference in the overall outlook of the graph. Different application areas require different types of layout styles. Sometimes, a layout that is considered good for a particular graph, will not give feasible results for a different graph. For example, as shown in Figure 2.2a, the relationship between nodes for a small graph of 10 nodes can be easily understood by CoSE layout [6]. But when a grid layout is applied on the same graph (Figure 2.2b), it becomes a bit difficult to understand the relations between the elements.

One of the most popular concepts in graph layouts is the force-directed approach. P. Eades presented the Eades model of force-directed approach [14], which was then improved by T.M.J. Fruchterman and E.M. Reingold [15]. This algorithm considers nodes as charged physical objects and edges as springs. Forces of attraction and repulsion are calculated between nodes. Nodes that are adjacent, attract each other, whereas other nodes repel each other. This algorithm aims to have the nodes spread evenly in the final layout. The runtime complexity of this algorithm is $\mathcal{O}(|V| + |E|)$.

2.2.1 Interactive Visual Tools

It is difficult to discern relationships between data just from static drawings. This task can be greatly simplified if the graph drawings are interactive. There



(a) CoSE layout on a graph of 10 nodes

(b) Grid layout on the same graph as in Figure 2.2a

Figure 2.2: Results of different layouts on the same graph

are numerous graph visualization tools available that can produce and present interactive graphs to a user. They enable the user to create, move, delete, hide and highlight nodes. They also provide easy access to many styling properties for nodes and edges. A more detailed overview of interactive techniques in graphs is available in [16].

When working with large-scale graphs, complexity management is critical. There are numerous complexity management techniques available in interactive graph editors. Zooming in and out, panning, hiding, and collapsing compound nodes are all examples of these techniques [17]. These methods add a great amount of ease in navigating through larger graphs.

2.3 Basic Terminologies

Edge endpoints: An edge $e = \{u, v\}$ is said to be *incident* to the nodes u and v. The points to which the edge is attached on these incident nodes are known as *endpoints*. When drawing graphs, edges may be specified to be connected to either the node's center or to a point on the node's outer perimeter (a connection point or a port).

Multi-edges: If two nodes have more than one edge connected between them, such edges are known as *multi-edges*. Figure 2.3a shows that there exist multi-edges between two nodes.

Self-loop edges: A *self-loop* is an edge that connects a node to itself. An example is shown in Figure 2.3b.

(a) Multi-edges (b) Self-loop

Figure 2.3: Examples of multi-edges and loops

Adjacency: The term *adjacent* is used for nodes as well as edges. Two nodes are said to be adjacent if they are connected by the same edge. Similarly, two edges are said to be adjacent if they are connected to or incident upon the same vertex.

Node degree: The *degree* of a node u is defined as the number of edges incident to node u. There is some additional terminology specific to directed graphs; in-degree and out-degree. *In-degree* of a node u is the number of edges that have node u as their target. Similarly, the number of edges that have node u as their source is the *out-degree* of u.

Neighbors: Each node v that lies on the other end of the edges E incident

to a node u, is a *neighbor* of the node u.

Node-angle: A *node/vertex-angle* is defined as the counter-clockwise angle between two consecutive adjacent edges of a node.

Graph degree: *Graph degree* is defined as the maximum number of edges that are connected to a single node in a graph. If the maximum degree from the graphs' nodes is n, then the graph is said to be an n-degree graph.

Sub-graph: *Sub-graphs* are the subsets of the original graph. If there exists a graph $G = (V,\ E)$, then it can have a subset graph $G' = (V',\ E')$ such $V' \subseteq V$ and $E' \subseteq E$.

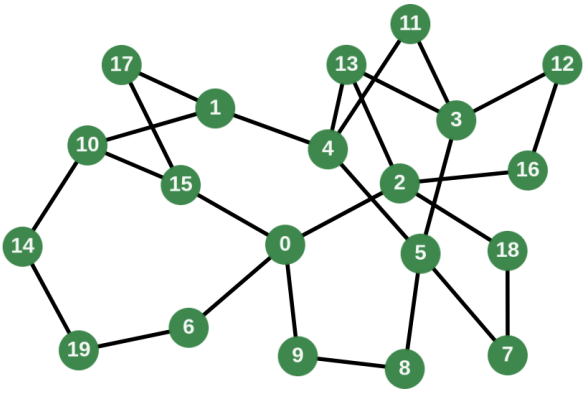
Path: A *path* is a route consisting of a sequence of edges that connect node u to node v such that no node in the route is traversed twice.

Connectivity: Graphs can be connected or disconnected. If there exists a path between every pair of vertices $(u,\ v)$ in a graph, then the graph is said to be *connected*. If this is not the case, then the graph is disconnected.

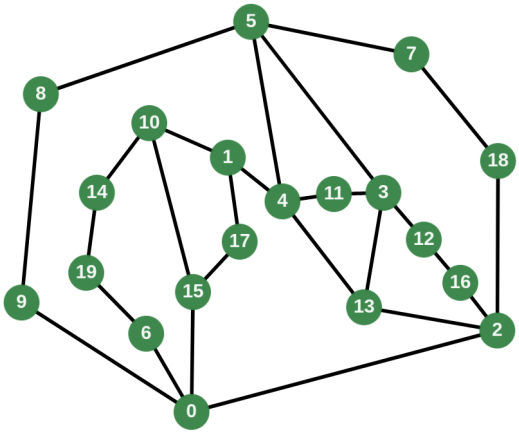
Planarity: If a graph can be drawn in such a way that it has no edge crossings, then such a graph is known as a *planar* graph. Many graph layout algorithms expect the input graph to be planar. If needed, a non-planar graph can be converted into a planar graph.

Graph Embedding: A planar *embedding* of a graph G can be defined to be an equivalence class of graph drawings that are determined by the cyclic order of the edges incident to each node n in the graph [18]. Figure 2.4b shows a planar embedding of the planar graph shown in Figure 2.4a.

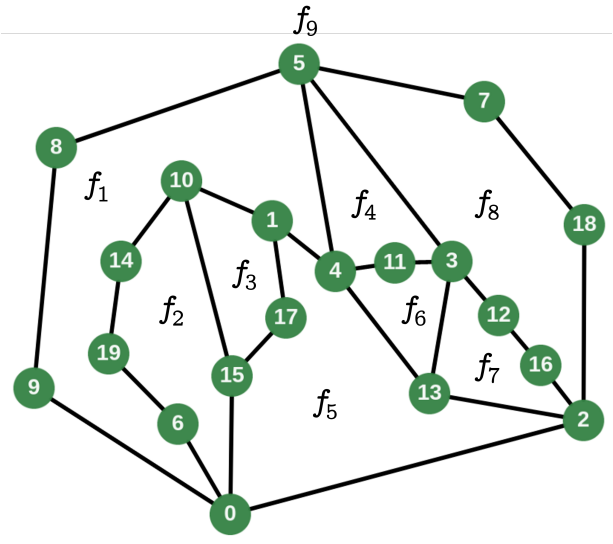
Faces: The regions of area that are created in planar graphs are termed as *faces* of the graph. Faces can be *external* or *internal* in nature. Each graph has only one external face which is the unbounded area around the graph. All bounded faces are internal faces. Figure 2.4c shows the faces of a planar embedding where f_9 is the external face and the other faces are internal faces.



(a) Planar graph



(b) Planar embedding of Figure 2.4a



(c) Faces of the planar embedding in Figure 2.4b

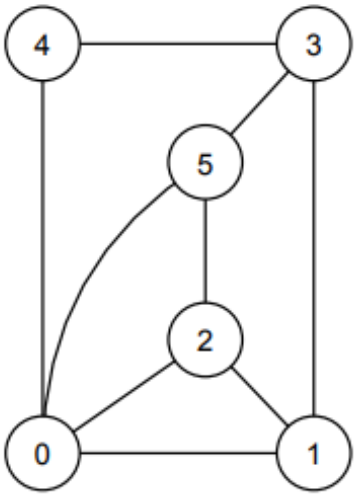
Figure 2.4: Creation of planar embedding of a graph and determination of its internal and external faces

Maximal face: A face f of a graph G is said to be *maximal* if it has the highest number of incident edges compare to any other face in graph G.

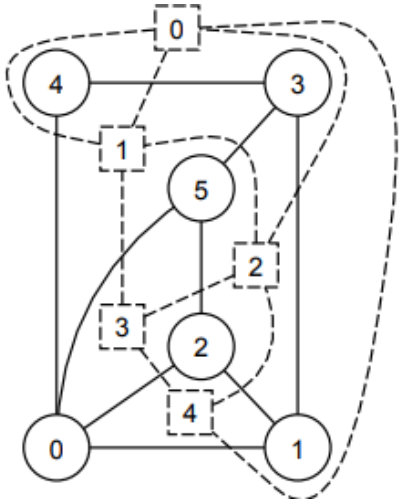
Half-edge: For each edge in a graph, it has two *half-edges* associated with it. If an edge e connects nodes u and v, it has a half-edge $e_1 = \{u, v\}$ directed from u and v and another half-edge $e_2 = \{v, u\}$ that points from v to u. e_1 and e_2 are said to be *twins* of each other, and they point in opposite directions.

Doubly Connected Edge List (DCEL): *DCEL* is a data structure for storing half-edges of a planar embedding of a graph.

Dual graphs: *Dual graphs* are the graphs that are created from the faces f of a graph G such that each face in G is a node in the dual graph G^*. Furthermore, there exists an edge $\{a, b\}$ between every two faces a and b that are neighbors to each other through an edge in the original graph G [18]. Figure 2.5 shows the creation of a dual graph from a planar embedding. In Figure 2.5b, face 0 is the external face and 1, 2, 3, 4 are the internal faces of the graph.



(a) Planar embedding G



(b) Dual graph G* constructed from Figure 2.5a

Figure 2.5: Creation of a dual graph from a planar embedding [18]

Compound Graphs: Graphs with compound nodes are known as *compound graphs*. The phrase *compound node* has not been frequently used in the graph drawing field in the past. A compound node is defined as a node that has *child* nodes placed inside of it. As a result, the compound node becomes the *parent* of those child nodes. Compound graphs maintain a level of hierarchy among their various sub-graphs and contain an arbitrary level of data nesting. Furthermore, a compound node in itself is a sub-graph of the original graph. Hence, compound graphs can also be coined as nested graphs. They are mostly used to handle the complexity of graphical networks and for managing containment relationships in data models. Figure 2.6 [3] shows an example of a compound graph.

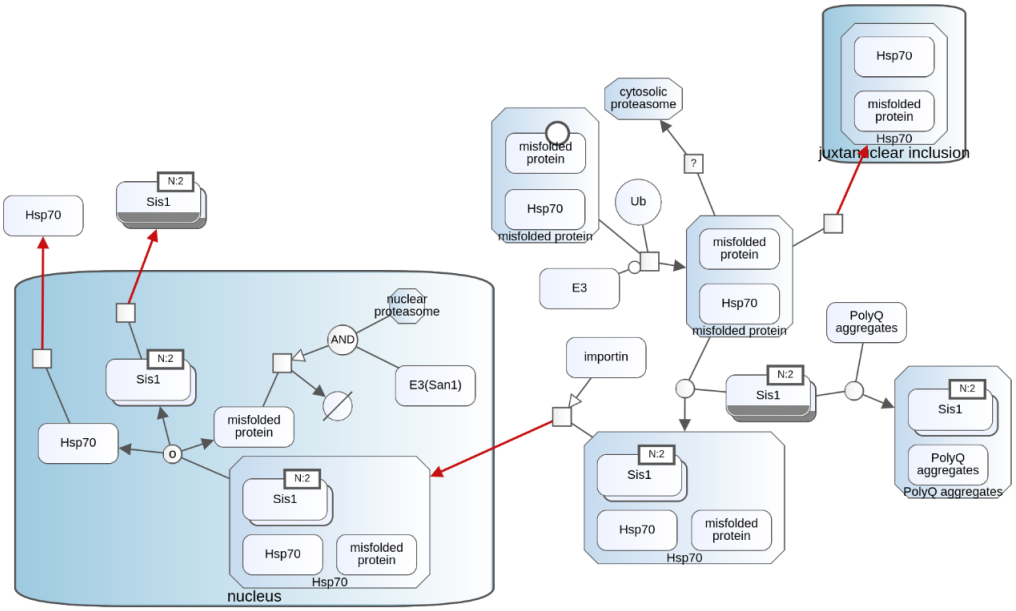


Figure 2.6: PolyQ Proteins Interference [3]

In mathematical terms, a compound graph $C = (V, E, F)$ contains a vertex set V, adjacency edges E and, inclusion edges F [6]. A compound node cannot have a link with any of its children or nested children. These graphs also contain intergraph edges or hierarchy crossing edges whose source and target lie in different sub-graphs.

Intra-graph edges: The edges whose source node and target node lie in the same graph are known as *intra-graph edges*. The black edges in Figure 2.6 are intra-graph edges.

Inter-graph edges: The edges whose source and target nodes lie in different graphs are known as *inter-graph edges*. The red-colored edges in Figure 2.6 are inter-graph edges.

Flow Networks: Formally, a flow network N is modeled as a tuple $N = (G,\ c,\ s,\ t)$, where $G = (V,\ E)$ is a directed graph containing node-set V and edge set E. Capacity of edge $e = \{u,\ v\}$ is denoted as $c(e)$ where $c(e) > 0$ [19]. Finally, s and t are the source and sink of the graph where $s \neq t$. Typically, such networks are used to model networks with some form of traffic flow. City water pipelines, electrical circuits, data network traffic, financial flows, and other networks constitute some of the examples of these networks.

Flow networks are directed transportation graph networks that consist of nodes and arcs. Nodes are considered as distribution centers connected to each other by arcs. Each arc has a *capacity* $c(e)$ and a *flow value* $x(e)$ associated with it. The capacity of an arc is defined as the maximum amount of traffic that can be moved on the arc, whereas flow refers to the amount of traffic *flow* that is sent on the arc. The flow value of the arc cannot be a negative number and can never exceed its capacity/limit (eq. 2.1).

$$0 \leq x(e) \leq c(e) \tag{2.1}$$

Flow networks have a *source* from which the traffic starts flowing into the outgoing nodes and the total traffic is collected at the end in a *sink*. Flow networks are built on the rule that for a node, the amount of incoming traffic must be equal to the amount of outgoing traffic. Except for the source and sink, this rule is applicable on all other nodes and is shown in equation 2.2 where $(\{u, v\}, \{v, u\}) \in E$.

$$x(\{u,\ v\}) = -x(\{v,\ u\}),\ u, v \notin \{s,\ t\} \tag{2.2}$$

A flow network is generally considered to be a mapping of edges onto positive real numbers ($x : E \rightarrow {\mathbb{R}_0}^+$). A flow network must always satisfy the constraints of capacity (eq. 2.1), skew symmetry (eq. 2.2) and flow conservation (eq. 2.3).

$$\sum_{\{v,\, w\} \in E} x(\{v,\, w\}) = 0,\; v \neq s,\; w \neq t \tag{2.3}$$

Figure 2.7 shows a sample flow network. "a" is the source of the network which sends out 6 units. "f" is the sink of the graph which receives all of those 6 units. All of the above-mentioned constraints are strictly followed in this network.

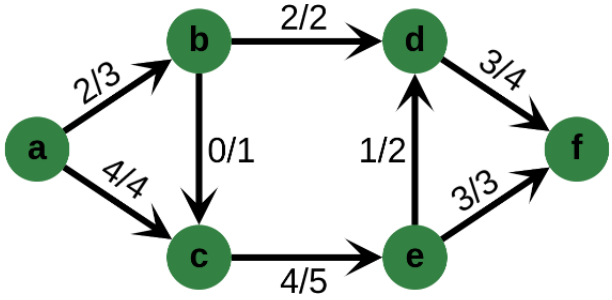


Figure 2.7: A sample flow network where each edge e has a *flow/capacity* attribute

Minimum Cost Flow (MCF) Problem: Flow networks can be modeled to solve the minimum cost flow problem. In this problem, each edge e in the network is said to have a cost $k(e)$ associated with it. The total cost of flow for using this path ends up being $k(e) * x(e)$. The basic goal is to determine the network path that will transfer the flow from source to sink at the lowest cost. Linear programming can be used to address this problem because this problem contains linear constraints and a linear cost function.

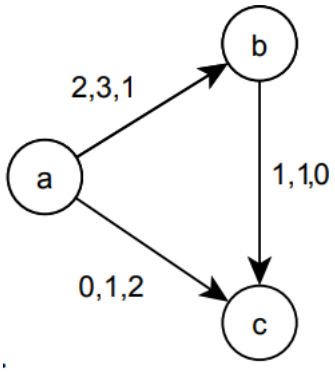
Residual capacity: *Residual capacity* $r(e)$ is the difference of an arc's capacity and its flow value (eq. 2.4).

$$r(e) = c(e) - x(e) \tag{2.4}$$

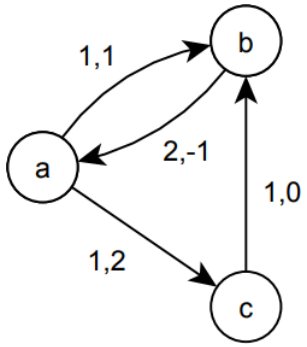
Residual Network: For a *residual network* G, each of its edges $\{u, v\} \in E$ are replaced by half-edges $\{u, v\}$ and $\{v, u\}$ such that they have the following properties:

- Half-edge $\{u, v\}$ has a cost k_{uv} and its residual capacity is $r_{uv} = c_{uv} - x_{uv}$
- Half-edge $\{v, u\}$ has a cost $k_{vu} = -k_{uv}$ and its residual capacity is $r_{vu} = r_{uv}$

In these networks, the arcs/half-edges with a positive residual capacity are taken into account for future calculations. Arcs with negative values of residual capacity are not considered in the network. Residual network is solved for the minimum cost flow problem. Figure 2.8 shows the extraction of a residual network from a flow network [18]. The values of residual capacity and cost in Figure 2.8b have been extracted from the previously mentioned properties.



(a) A sample flow network where each arc is labelled as (flow x_{uv}, capacity c_{uv}, cost k_{uv})



(b) Residual network of the graph in Figure 2.8a where each arc is labelled as (residual capacity r_{uv}, cost k_{uv})

Figure 2.8: Creation of a residual network from a flow network [18]

2.4 Performance Criteria of Graph Layouts

The performance of layout algorithms can be judged by some measurable properties or metrics. Many research works have presented their take on the different performance criteria of layout algorithms [5, 13, 20, 21, 22, 23]. The most notable criteria are mentioned as follows:

- Maximizing symmetry
- Minimizing edge crossings
- Minimizing edge bends
- Maximizing angle between adjacent edges of a node
- Minimizing graph area
- Minimizing node-to-node overlaps
- Minimizing edge-to-edge overlaps

Not all of these criteria are suitable to be used for the same kinds of graphs. The choice of performance criteria for layout algorithms depends on the type of application for which graphs are being built. Based on the application type, some criteria take preference over others. Hence, the performance criteria for each target graph type should be chosen carefully. In any case, as each of these criteria are hard to satisfy (known to be NP-hard problems), satisfying multiple of them is even harder [5].

2.5 Related Work

2.5.1 Layouts for Compound Graphs

Drawing compound graphs by hand is a very tedious task. An average user is expected to spend up to 25 percent of their time on manual layout adjustments [24]. If a node has to be added inside a compound, the parent has to be resized to accommodate this new addition. Hence, there is a dire need to perform such layouts automatically. Compound graph layouts have garnered very little attention because these layouts are relatively harder as compared to simple graph layouts, and there exist only a few layout algorithms for them.

When drawing compound graphs, there are some additional constraints regarding the geometry of graph elements. For example, the children of a compound node can not be placed outside their parent node as this would be misleading. If a child node is moved outside of the current compound parent boundary, the parent must expand to accommodate it.

CoSE layout, a force-directed compound spring embedder layout for undirected graphs, is one of the most popular algorithms for compound graph layout [6]. It takes the force-directed approach of T.M.J. Fruchterman and E.M. Reingold [15] and builds on their idea to handle multi-level nesting of nodes, edges between nodes of different sub-graphs, non-uniform node sizes, and other constraints. In addition to the forces of attraction and repulsion, weaker gravitational forces are also introduced in this layout, which allow the components of the same graph to remain close to each other. The combinations of these three forces (spring forces for edges and repulsion and gravitational forces for nodes) decide the final positions of the graph elements at the end of the layout. The runtime complexity of one run of this algorithm is $\mathcal{O}(|V|^2 + |E|)$.

fCose [25] is an improved faster version of the CoSE algorithm [6] with additional support of constraints. The supported constraints are horizontal and vertical alignment, relative node placement (towards right, left, top or bottom), and fixed node positions. It combines the previous compound spring embedder layout with the spectral drawing technique. The algorithm creates a fast draft layout and enforces constraints on it in the following phases. It is a faster approach to perform layout on compound graphs.

2.5.2 Orthogonal Layout Algorithms

Recently, a lot of work has been done to create algorithms that generate aesthetically pleasing graph drawings. Numerous layout algorithms have been created in this regard. However, the layout algorithms that are relevant for this thesis are the approaches towards orthogonal layout algorithms.

Orthogonal drawings are those diagrams in which the edges are composed of chains of horizontal and vertical segments only. They are used in many application areas like entity-relationship diagrams in data modeling, PCB layouts, and UML class diagrams. Orthogonal layouts should produce aesthetically pleasing drawings with fewer bends, fewer crossings, fewer overlaps, and small area.

The three-phase method was presented by C. Biedl et al. [26] for creating an orthogonal layout for graphs. Apart from the preprocessing and postprocessing steps, this approach consists of three main middle steps: node placement, edge routing, and port assignment. The first phase treats the nodes as points and places them into a grid. Edge routing chooses the most feasible path between two connected nodes such that there are minimum number of bends in the edge. In the last phase of port assignment, node dimensions are modified according to the number of ports on each side. This algorithm is termed as the *TSS-model*. K. Freivalds and J. Glagolevs [27] improve the node placement phase of the TSS model by assigning locations to the nodes in such a way that total edge length is minimized. They further improve the graph quality by applying 1-D compaction in horizontal and vertical directions.

Human-Like-Orthogonal Layout (HOLA) algorithm is a layout algorithm by S. Kieffer, et al. [12] that strives to create automated orthogonal graph layouts that look like they were made by a human. The paper initially presents the results of a conducted study in which users create orthogonal layouts of presented graphs. Based on the user-created layouts, the authors establish aesthetic criteria that should be followed by layout algorithms. Their presented algorithm HOLA creates good-looking drawings that are quite similar to hand-drawn graphs.

The algorithm first iteratively removes the leaf nodes of the graph until there are none left. The pruned nodes are combined to form tree-type structures that are attached back to the drawing at the end. The resulting graph after pruning is called the core of the graph. After a stress-minimizing layout on the core, the orthogonalization step is performed. This step is divided into two phases: node orthogonalization (which orthogonalizes the core of the graph), and chain orthogonalization (which creates straight-looking layout of chains of 2-degree nodes).

Finally, the trees pruned in the first step are attached back to the core at the most feasible location. Multiple stress-minimizing constraint layouts are run in this algorithm at different steps. Figure 2.9 shows a sample HOLA layout.

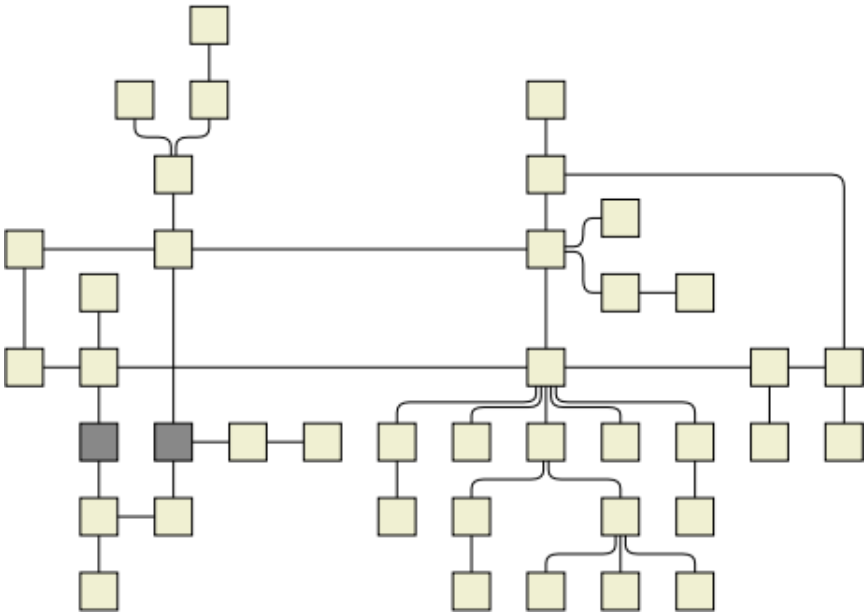


Figure 2.9: A sample HOLA layout [12]

R. Tamassia presented the topology-shape-metrics (TSM) approach [13] for creating a grid-based orthogonal layout such that minimum number of bends and edge crossings exist in the final drawing. The algorithm makes use of network flow diagrams and has a complexity of $\mathcal{O}(n^2/log\, n)$. There are three phases of this algorithm, namely planarization, orthogonalization, and compaction. Numerous other research works on orthogonal layout algorithms have either used the TSM approach as a background model or tried to build upon the TSM approach to improve its results [28, 29, 30, 31]. This algorithm has been used as the baseline for the work in this thesis. It is described in more detail in section 3.1.

2.5.3 Orthogonal Layout on Layered and Hyper Graphs

C. D. Schulze et al. present the KLay Layered orthogonal layout algorithm [10] which combines port constraints with the layered-based approach for graph drawing. The approach firstly eliminates cycles in the graph and then assigns nodes

to different layers. After that, the nodes in each layer are organized such that the graph has minimum edge crossings. Based on this organization, nodes are placed in vertical positions so that straight edges can be drawn between them. Finally, based on the chosen routing, bendpoints are created in the edges.

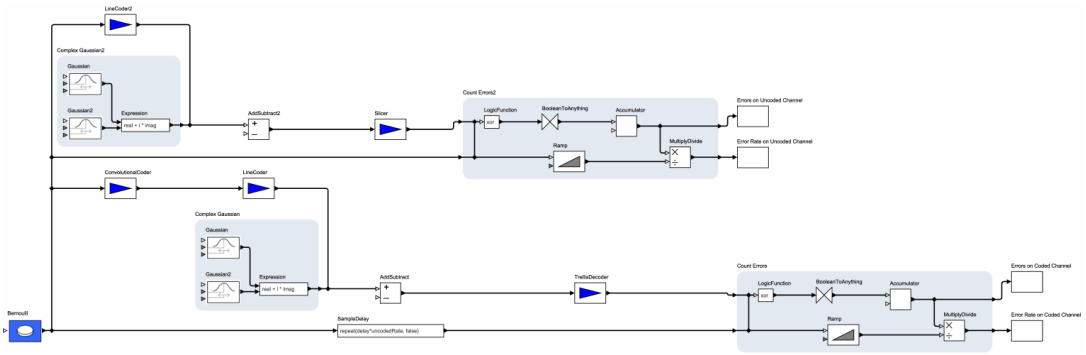
U. Ruegg et al present also present an approach for the orthogonal layout of data flow diagrams with ports [9]. They use a stress majorization technique with constraints to achieve this goal. Their algorithm CoDaFlow first positions nodes in such a way that stress between them is minimized. It then aligns the nodes on a grid and then performs orthogonal routing of edges at the end. The algorithm also handles compound graphs by means of a global approach of positioning compounds together with their child nodes or by using a bottom-up approach; i.e. initially working on innermost nodes and then moving on to outer nodes. Figure 2.10 shows the results of the KLayered approach and CoDaFlow on the same diagram.

M. Siebenhaller et al. present an approach for orthogonal layout on signalling hypergraphs [11]. They combine the TSM approach [13] with the HOLA algorithm [12] to create hand-drawn-looking drawings. Their enhancement approach consists of complex nesting, complex packing, and flipping of the sub-components of the graph. The runtime complexity for this approach is $\mathcal{O}(n^3)$ for sparse graphs.

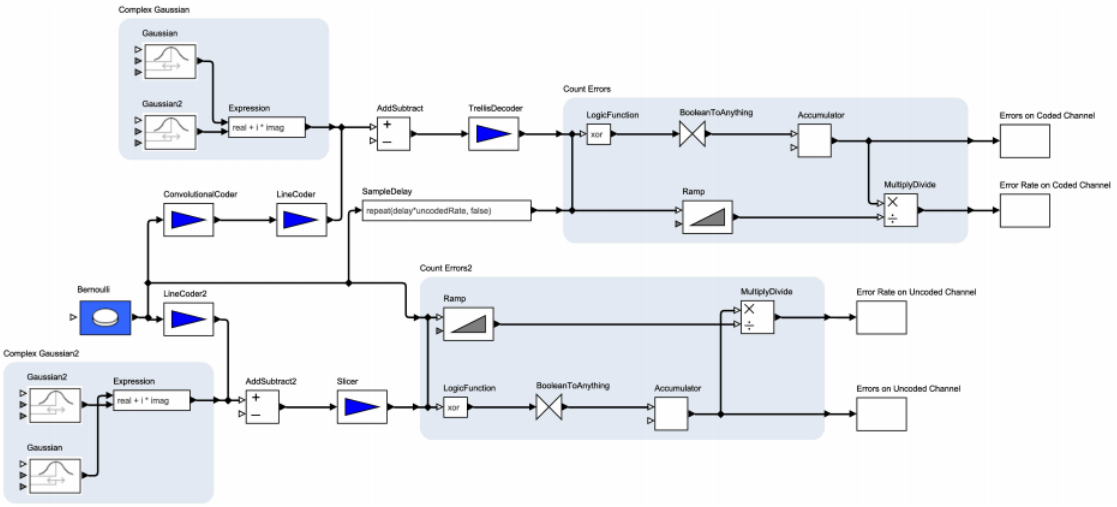
As explained previously, there exists no explicit research work for an orthogonal layout algorithm for general "compound" graphs. Hence, in this thesis, we present an orthogonal layout algorithm for compound graphs.

2.6 Cytoscape.js

Cytoscape.js is an open-source network library that was originally built to construct biological network diagrams [32]. However, it is now widely used for general graph visualization and the analysis of complex networks. This platform is



(a) Orthogonal layout with KLayered Approach [10]



(b) Orthogonal layout with CoDaFlow [9]

Figure 2.10: Results of different orthogonal layouts on the same data flow diagram [9]

constantly improving and evolving owing to the contributions of numerous authors [33]. It is in use by many big-tech firms, government organizations, research institutes, and many other applications and services.

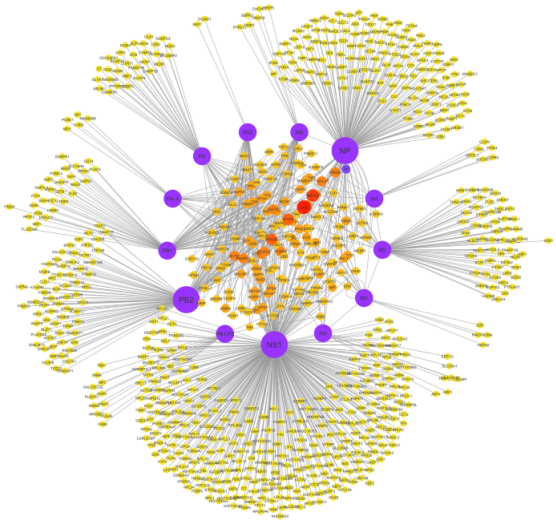


Figure 2.11: Depiction of interaction between proteins of human host with proteins of Influenza virus visualized in Cytoscape.js [32]

Cytoscape.js is a Javascript-based platform that allows developers to incorporate graphical representations into their websites and data models [34]. This library provides an application programming interface (API) that allows the creation of a variety of graphs, including simple, compound, and multigraphs, with uniform or non-uniform node sizes, and directed, undirected or mixed edges. Users can create interactive graphs with Cytoscape.js, which means they have the freedom to navigate nodes around on the web interface. Applications can readily integrate Cytoscape.js into their software models and use it on the node.js runtime environment. Figure 2.11 shows a sample biological network [35] created with Cytoscape.js.

The architecture of Cytoscape.js contains two major components that programmers are mainly concerned with. The first is the *core* of the library, which allows the user to access its features. These features include graph data retrieval, viewport manipulation (zooming, locking, panning, etc.), and layout execution (random layout, CoSE layout, concentric layout, grid layout, and so on). It also allows nodes and edges to be styled by changing their graphics. The collections of

data elements like nodes and edges, which are returned by the library functions, comprise the other main component of Cytoscape.js. Popular built-in clustering and search techniques are also included in these functions [34].

Developers can create their extensions on top of the basic library of Cytoscape.js. One example of such extensions is `cytoscape.js-view-utilities`. With this plugin, the user can conceal or show nodes and edges, highlight certain graph elements, perform marquee zoom, and freely select data with a Lasso tool. The demonstration of the Lasso tool has been shown in Figure 2.12. The algorithm presented in this thesis (i.e., C-TSM) has also been created as a Cytoscape.js extension `cytoscape.js-c-tsm`. Cytoscape.js library has been used to create the majority of the figures in this thesis.

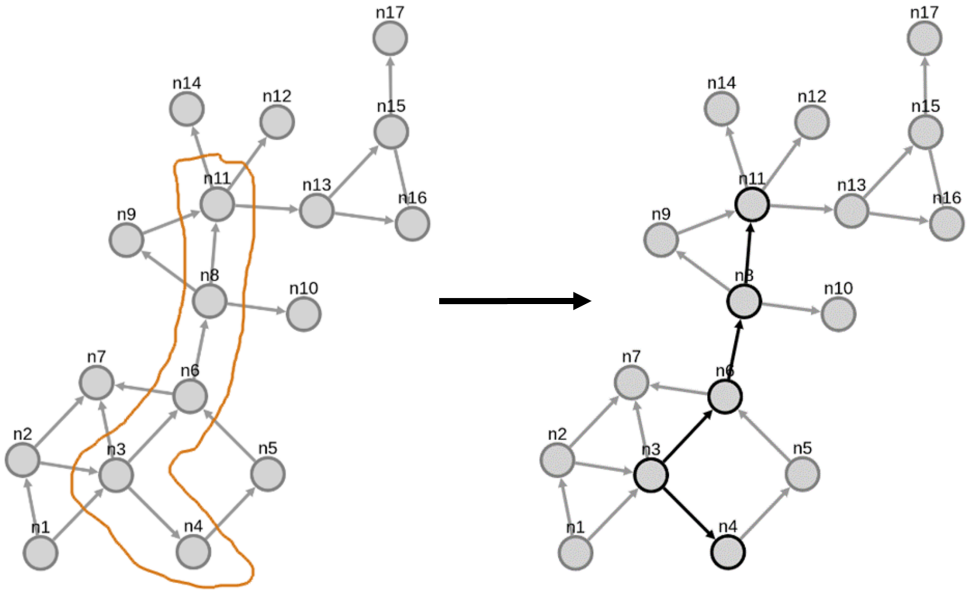


Figure 2.12: Demonstration of the Lasso Tool in cytoscape.js-view-utilities extension for free-hand selection of elements

Chapter 3

C-TSM Layout Algorithm

This chapter explains how the TSM approach has been utilized to perform an orthogonal layout on compound graphs. The main contributions of our algorithm are to the preprocessing and postprocessing steps to prepare the compound graph for the layout and reconstruct the compound at the end of the layout as well as modifications needed during the application of the original TSM algorithm.

3.1 Topology-Shape-Metrics Approach

The algorithm presented in this thesis is based on the implementation of the approach explained by P. Klose [18] who presents a generic framework for the Topology-Shape-Metrics approach presented by Robert Tamassia [13]. We utilize this work to cater to compound graphs.

The TSM approach is divided into the following three phases:

1. Planarization
2. Orthogonalization
3. Compaction

The workings of these phases are explained in more detail in the following sections.

3.1.1 Planarization

The first step of TSM works on extracting the *topology* of the graph. This can be achieved by creating an embedding of the graph's initial drawing. The embedding is then converted into a planar one by replacing the edge crossings with dummy nodes. These dummy nodes and edges can be removed and replaced by the original edges in the final stages of the layout.

The task of planarization is divided into three further steps. The first step provides an algorithm for checking the planarity of the graph, whereas the second step embeds those edges back into the graph that are violating the planarity. The violations are removed by replacing the edge crossings with dummy nodes and edges. The last phase calculates the faces in the graph by traversing along the edges and finding the adjacent faces (left or right faces) of each edge. This method results in a correct traversal of faces because, in a planar embedding, the adjacent edges of a node are always ordered in the counter-clockwise direction.

3.1.2 Orthogonalization

After the faces have been determined in the planarization phase, the next step is to create a flow network with those faces. The goal is to solve the flow network in the context of the minimum cost flow problem. Solving this network gives us the angles between node edges and the required bendpoints.

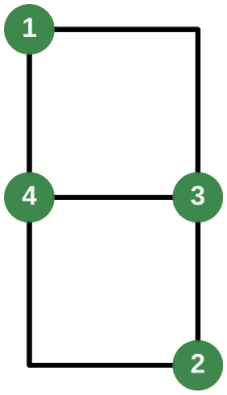
Tamassia's Orthogonalization

Tamassia's approach [13] uses flow networks to reduce the number of bends in an orthogonal drawing. The technique also calculates the angles between edges that

are incident on a node. This approach is only applicable to 4-degree graphs.

The algorithm calculates an "orthogonal representation" H of the graph. In addition to the topology-data from the planar embedding, the orthogonal representation also stores the *shape* of the drawing by including information of *angles* in the drawing. These angles may be vertex-angles or angles at the bendpoints. Since the drawing is made up of faces, each face is given a *shape* such that it forms an orthogonal polygon.

Each node u is assigned a list of its adjacent edges and the angles between these consecutive edges are determined. In addition, each edge is assigned a list of bend-angles. Figure 3.1 shows the creation of angle-data from a given drawing. In Figure 3.1a, the $\{1, 3\}$ edge's successor in the anti-clockwise direction is $\{1, 4\}$ and the counterclockwise-angle (ccw-angle) between these two edges is 270°. Furthermore, the ccw-angle from $\{1, 4\}$ to $\{1, 3\}$ is 90°. The angle data for all nodes of the graph is shown in Figure 3.1b.



(a) Planar orthogonal drawing

Node	Incident Edge	Angle
1	{1, 3}	270°
	{1, 4}	90°
2	{2, 3}	90°
	{2, 4}	270°
3	{3, 1}	90°
	{3, 4}	90°
	{3, 2}	180°
4	{4, 3}	90°
	{4, 1}	180°
	{4, 2}	90°

(b) Angle data of graph in Figure 3.1a

Figure 3.1: Calculation of angle data for an orthogonal representation

The previously mentioned shape data is obtained by constructing a directed network flow model N for the planar embedding G_p and solving it for minimum cost. The network is created such that the faces and vertices in the graph G_p are considered as nodes in the flow network N. The flow values of the arcs are always taken to be positive values. The other properties such as capacity, cost,

and residual capacity are given integer values. The flow network is constructed as follows:

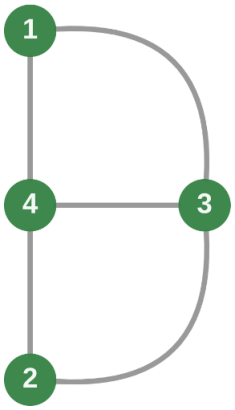
- For each vertex and face in G_p, a node is created in N (Figure 3.2b).
- For each adjacent face f of a vertex u in G_p, a directed edge is created from vertex u to face node f in the graph N (Figure 3.2c). The flow value of an arc $\{u, f\}$ from a vertex u to its adjacent face f gives the value of the angle formed at vertex u inside the face f. This angle can range from 90°to 360°. The cost of such arcs is considered to be zero because this angle is at a vertex.
- All adjacent faces in G_p are connected to each other in N via each edge that they are adjacent to in G_p. The edges between faces are shown in dashed lines in Figure 3.2d. The flow value of an arc $\{f, g\}$ where f and g are adjacent to each other through an edge e, represents the number of bends that will be created in edge e. The cost of such arcs is equal to the number of bends.

Figure 3.2 shows the creation of a flow network from a planar embedding.

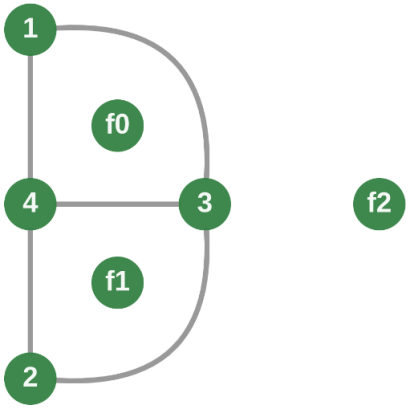
The flow network is solved to get the flow values that produce minimum cost for the network. There exist many different approaches to solve this problem. P. Klose suggests using a shortest path algorithm [18] to solve it. Initial values are given to the flow of each arc, and the condition of mass-balancing is tried to be achieved. When this condition is reached, the final flow values are used to create the orthogonal representation of the graph.

3.1.3 Compaction

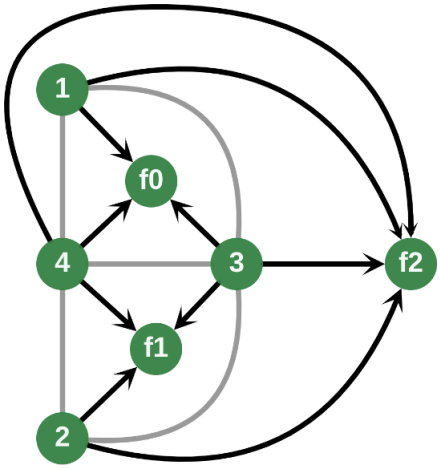
The third and the last phase of TSM works on determining the final layout positions of the nodes. The goal of this step is to minimize the lengths of edge



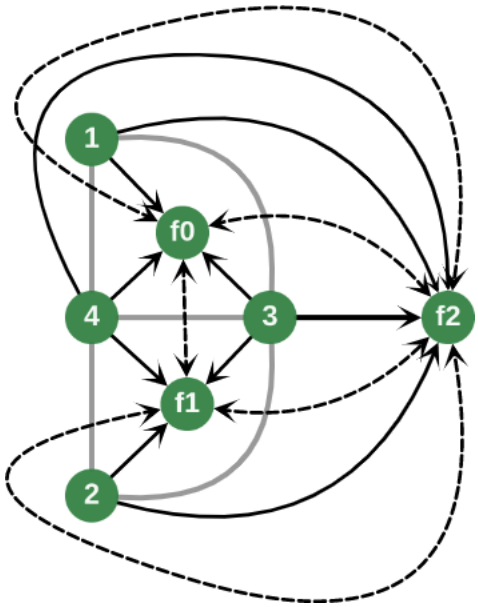
(a) A planar embedding



(b) Faces of the planar embedding in Figure 3.2a where $f0$ and $f1$ are the internal faces and $f2$ is the external face



(c) Creation of directed edges from vertices to adjacent faces

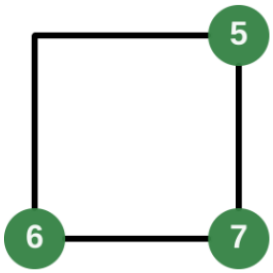


(d) Creation of edges between faces (in dashed lines)

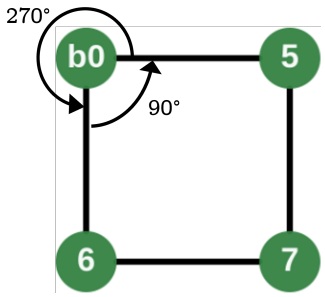
Figure 3.2: Creation of a flow network from a planar embedding

segments without creating node-to-node overlaps and edge crossings. This phase also uses a network flow model to compact the graph.

The compaction approach presented by Tamassia requires that there should be no bends in the graph and all of the faces must be rectangular in shape [13]. To prepare the graph for compaction, there are a few preprocessing steps that have to be implemented. The first is the creation of dummy nodes for edge bendpoints. The flow network in the orthogonalization step gives the number of bendpoints that have to be created for each edge. Based on that information, dummy nodes are created along with incident dummy edges. Figure 3.3 shows the creation of a dummy node at a bendpoint. Furthermore, the "angle" data is also updated for these dummy nodes. This helps to keep updated data of the orthogonal representation. Since a bendpoint is connected to two other nodes, its angles between its adjacent edges are always 90°and 270°(Figure 3.3b).



(a) A simple graph with a bendpoint between edge (5, 6)



(b) The edge {5, 6} is removed and the bendpoint in Figure 3.3a is replaced by a dummy node $b0$. New dummy edges $\{5, b0\}$ and $\{b0, 6\}$ are created to secure the new connections

Figure 3.3: Creation of a dummy node for a bendpoint

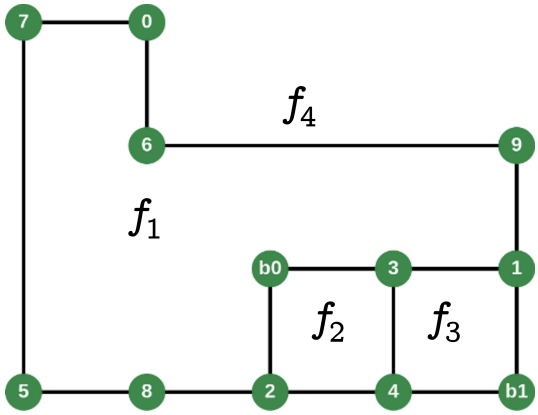
The next step is to convert the non-rectangular faces into rectangular faces. Such faces are repeatedly split into rectangular faces until all faces are rectangular. This is done by introducing a dummy node and a dummy edge for each split. P. Klose mentions in his work that the technique for making faces rectangular is based on the work of G. D. Battista et al [5].

If a face is not rectangular, we need to determine the following properties for

each edge e in the face:

1. $next(e)$: the next counter-clockwise edge in the face
2. $corner(e)$: the shared node between e and $next(e)$
3. $turn(e)$: the change in direction based on the angle between e and $next(e)$ in the face.
   - If $\angle(e,\ next(e)) = 90°$, turn = +1
   - If $\angle(e,\ next(e)) = 180°$, turn = 0
   - If $\angle(e,\ next(e)) = 270°$, turn = -1
4. $front(e)$: the edge for which the sum of turn values for all edges between e and $front(e)$ gives the value of "1". The turn value of $front(e)$ is not counted in this sum.

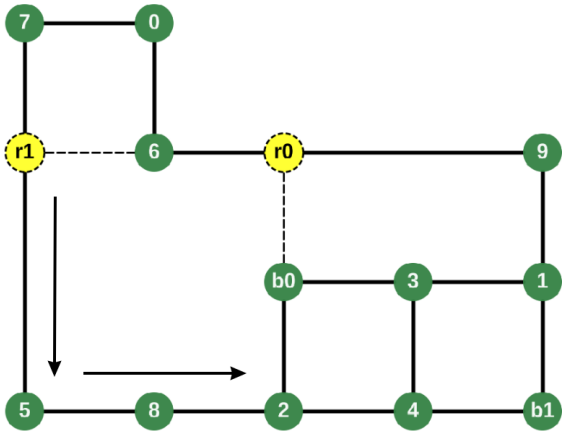
The edges which have a turn value of "-1" are the ones that destroy the rectangularity of a face. Such edges are extended with a dummy edge such that the dummy edge has an angle of 180° with the edge. The extension is made from $corner(e)$ onto a rectangular dummy node in $front(e)$. This algorithm is only applicable to internal faces. Figure 3.4 shows the creation of rectangular faces in face f_1 of the graph by determining the dummy nodes and edges from the data in Figure 3.4b. Edges {2, $b0$} and {9, 6} have a turn value of -1 and are extended onto their fronts. This results in the conversion of a non-rectangular face into multiple rectangular faces. The edges of the face are traversed in the counter-clockwise direction, as shown by the direction of arrows in Figure 3.4c.



(a) Planar orthogonal drawing with marked faces

Edge(e)	Next(e)	Corner(e)	Turn(e)	Front(e)
{7, 5}	{5, 8}	5	+1	{5, 8}
{5, 8}	{8, 2}	8	0	{2, b0}
{8, 2}	{2, b0}	2	+1	{2, b0}
{2, b0}	{b0, 3}	b0	-1	{9, 6}
{b0, 3}	{3, 1}	3	0	{1, 9}
{3, 1}	{1, 9}	1	+1	{1, 9}
{1, 9}	{9, 6}	9	+1	{9, 6}
{9, 6}	{6, 0}	6	-1	{7, 5}
{6, 0}	{0, 7}	0	+1	{6, 0}
{0, 7}	{7, 5}	7	+1	{0, 7}

(b) Edge data of the internal face f_1 in Figure 3.4a



(c) Extending edges with negative turn values to create rectangular faces

Figure 3.4: Splitting non-rectangular internal faces into rectangular faces

The above mentioned method is only applicable to internal faces. A different approach has to be used for the external face because some edges might not have a front in the external face. For dealing with a non-rectangular external face f_{ext}, a new rectangular face f'_{ext} is constructed around the drawing as shown in Figure 3.5a. After that, the original external face f_{ext} is traversed in clockwise direction (as shown by the arrows in Figure 3.5b), and a dummy node and edge is placed if the angle of an edge with its following edge is 270°. The edge pairs ($\{7,0\}$, $\{0,6\}$), ($\{r0,9\}$, $\{9,1\}$), ($\{1,b0\}$, $\{b0,4\}$), ($\{8,5\}$, $\{5,r0\}$) and ($\{r1,7\}$, $\{7,0\}$) have an angle of 270° between them and are thus extended to the outer boundary to create rectangular faces.

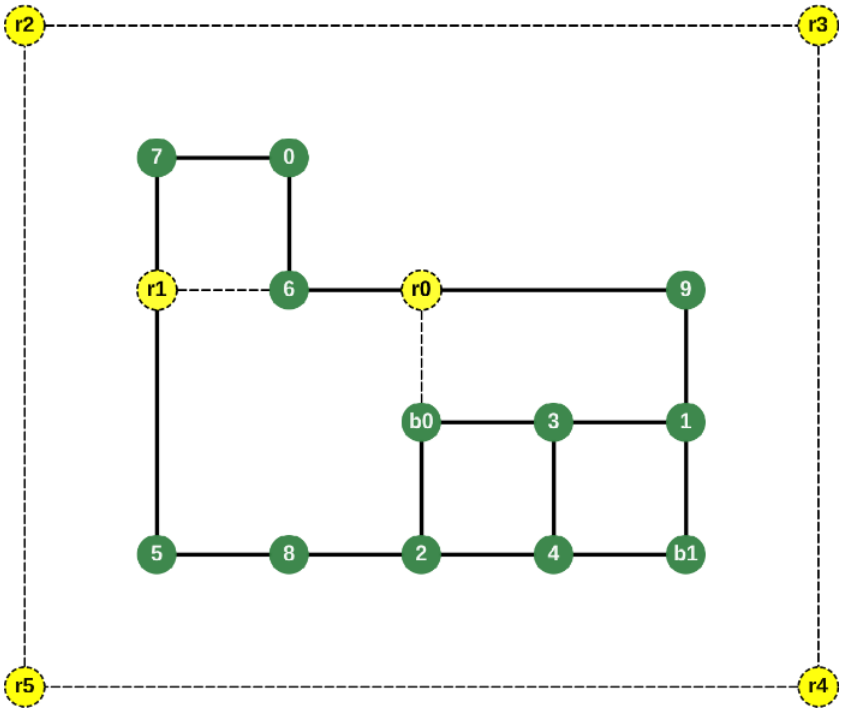
The final preprocessing step of compaction is to assign indices to the sides of the faces. Assigning indices to faces makes the process of compaction much easier since for each type of compaction (horizontal/vertical), only two sides have to be considered. The top and bottom sides are considered for vertical compaction and the left and right sides are considered for horizontal compaction. This notation is also helpful in the layout phase where nodes are finally assigned positions on a grid. The following indices are assigned to the face sides:

- Left side index = 0
- Top side index = 1
- Right side index = 2
- Bottom side index = 3

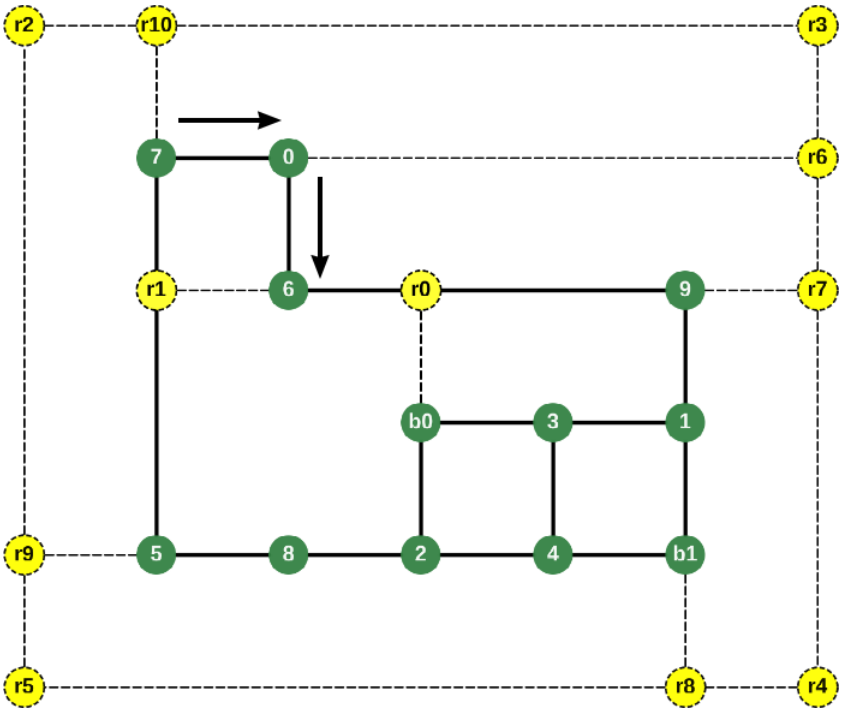
Figure 3.6 shows the assignment of indices to each edge in a face.

After the previously mentioned steps, the graph is finally ready for compaction. The goal of this compaction step is to produce a grid layout with minimum area and minimum average edge length. The final lengths of the edge segments are derived by solving a flow network.

Compaction is performed in two directions: horizontal and vertical. The horizontal compaction calculates the lengths of the horizontal edge segments and



(a) Creation of a new external face f'_{ext} around the actual external face f_{ext}



(b) Extending edges with an angle of 270° to create rectangular faces

Figure 3.5: Splitting the non-rectangular external face into rectangular faces

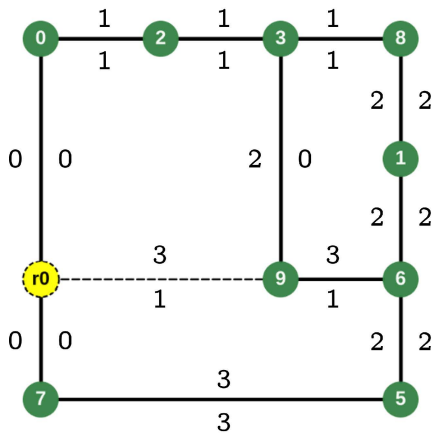


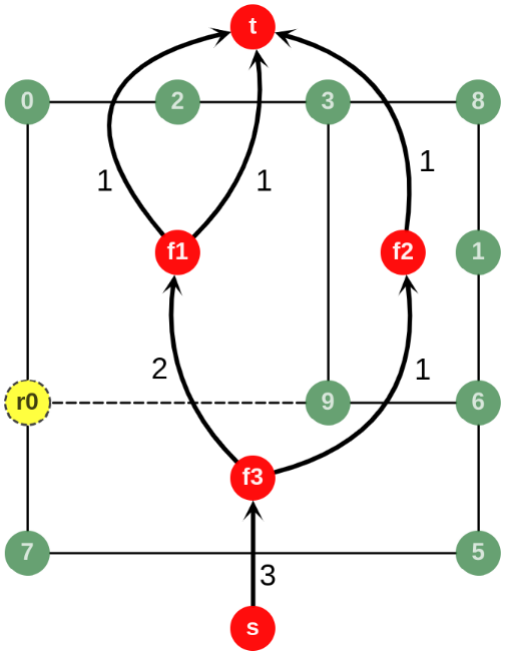
Figure 3.6: Marked face sides in an orthogonal graph

vice versa. Two separate flow networks are created for both compactions and are solved to find the minimum value of flow.

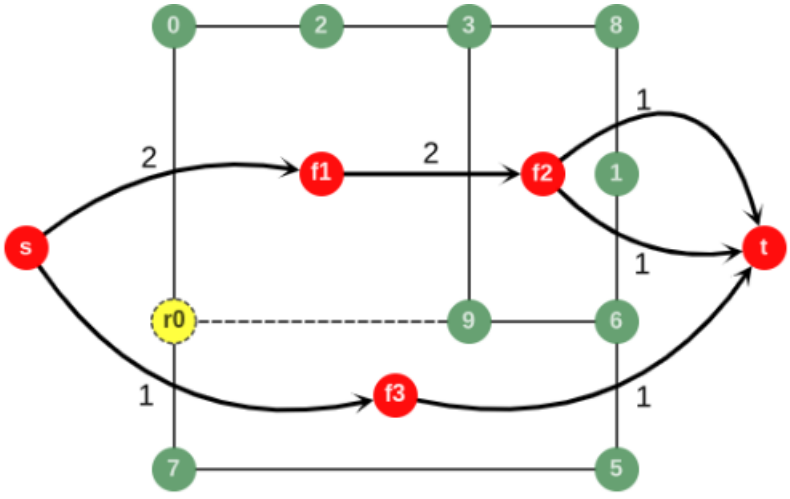
As in the flow networks produced for orthogonalization, the networks in this phase are also calculated in the same way, but with some minor adjustments. The first modification is that the external face is not considered in this network. Secondly, the horizontal flow network N_{hor} has the source on the top side of the graph and the sink on the bottom side of the graph. Similarly, the vertical flow network N_{ver} has the source on the left and the sink on the right. Thirdly, the faces that are connected to each other only through the related sides, are the ones which we aim to compact. An arc $\{f, g\}$ in N_{hor} is only created for faces adjacent to each other through horizontal edges and vice versa. The lower bound on the flow value of each edge is equal to 1 since this signifies the minimum unit length of the edge. The final flow values of the arcs between faces that are obtained after solving the network for minimum cost flow determine the final lengths of the edges that are crossed by those arcs.

Figure 3.7 shows the output lengths of the edges determined by solving the horizontal and vertical flow networks for minimum cost flow.

After the compaction has been performed, then come the postprocessing steps. Firstly, the nodes (including all dummy nodes) are mapped on a grid at unique



(a) Horizontal flow network N_{hor} solved for minimum cost flow



(b) Vertical flow network N_{ver} solved for minimum cost flow

Figure 3.7: Solving the horizontal and vertical flow networks for compaction of the graph

positions. This ensures that there are no node-overlaps in the drawing. To do this, the algorithm processes the faces one by one, starting with the external face. It firstly finds the bottom-left corner node of the face. Starting with this position, it moves along the edges of the face, assigning a new position to each subsequent node. The change in node direction is determined from the side of the edge that it belongs to. The new position is decided based on the length of the edge obtained from the flow network for compaction and the information of the side of the face that an edge belongs to. The direction of new placement is chosen by the following criteria:

- If face side index = 0, place node in the positive y direction
- If face side index = 1, place node in the positive x direction
- If face side index = 2, place node in the negative y direction
- If face side index = 3, place node in the negative x direction

The final phase of the algorithm is to remove the dummy nodes for edge crossings, bendpoints, and the dummies for creating rectangular faces. Edges are reconstructed after removing the bendpoint and edge crossings.

Our research work mainly makes modifications to the planarization and the compaction phase in order to make the algorithm work for compound graphs. These changes are mentioned in detail in the further sections.

3.2 Prerequisites

The TSM approach has a set of conditions that must be followed. Those conditions state that the input graph must be:

- A connected graph

- A maximum 4-degree graph
- A planar graph

The concept of *connectivity* changes its meaning when being discussed in the context of compound graphs. A compound graph may, in essence, be connected but it cannot be used as it is with the TSM approach. The main idea behind this algorithm is that the compound graph is essentially converted into a "simple graph" so that the TSM approach can be used on it.

As for the degree of the graph, if the degree is greater than four, then all except 4 neighbors will end up having non-orthogonal edges. Hence, the input graph must always be a maximum 4-degree graph. One can, however, extend the support to arbitrary degree graphs by simply introducing a clone of a high degree node where edges are split across these clone nodes to satisfy the degree constraint. Other than that, the edges of the input graph must not be:

- Self-loops
- Multi-edges

Such edges can be handled similarly with additional preprocessing of the input graph.

3.3 C-TSM Approach

The main idea behind this work is to convert compound graphs into simple graphs. The converted graphs still maintain the same topology of nodes, as in the original graph. Therefore, when the TSM approach is applied to such graphs, the child nodes of the compounds always remain inside the boundary of their parent compounds. These simple graphs can then be converted back to compound graphs.

As mentioned earlier, this work is an adaptation of the approach presented by P. Klose. We have used some parts of his TSM approach to work for compound graphs. This section firstly mentions the preprocessing steps that are applied to the graph before applying the TSM layout to it. After that, we explain the modifications made in the actual TSM approach. And finally, the postprocessing steps after the TSM layout are explained. Algorithm 1 shows the high-level overview of this work.

Algorithm 1 Compound-TSM Layout Algorithm

```
function C-TSMLayout(G)
    Input graph, G
    Simple graph, G_s ← null
    G_s, compoundNodes ← PreProcessGraph(G)
    ApplyModifiedTsmLayout(G_s)
    PostProcessGraph(G, G_s, compoundNodes)
end function
```

3.3.1 Preprocessing

A compound graph has to be modified before the TSM layout can be applied to it. This subsection explains the preprocessing steps that have been performed to achieve this idea. As mentioned in section 3.2, the input graph must have some fixed properties. We assume that the input graph is a maximal 4-degree graph and does not contain self-loops or multi edges. Apart from this requirement, the graphs should also be planar and connected.

Before the TSM layout can be applied, the compound graph must be given an initial layout and converted into a simple graph. Algorithm 2 shows the pseudocode for this step.

Initial Stress-Minimizing Layout

The first step of preprocessing is to give an initial layout to the input graph. For this purpose, one can typically apply some kind of a planarization heuristic but

Algorithm 2 Preprocessing algorithm pseudocode

```
function PreProcessGraph(G)
    compoundNodes ← GetCompoundNodes(G)
    RunCoseLayout(G)
    SortCompounds(compoundNodes)
    G_s ← ConvertToSimpleGraph(G, compoundNodes)
    return G_s, compoundNodes
end function
```

for practical reasons, we apply CoSE layout on the graph to evenly distribute the nodes across the plane. The number of edge crossings produced in the graph after CoSE layout are propagated to the final layout.

Other alternatives for the initial layout are CoLa layout [36] and fCoSE layout [25]. Implementations of these layout algorithms are not currently suitable for use as sub-algorithms and hence, we could not use them in our implementation.

Sorting Compounds

Most parts of our algorithm require that the compound nodes should be worked on from the innermost nesting level. In short, an inside-out approach is used. For this purpose, we sort the compounds based on their area. This is done because the output from the initial layout phase ensures that each child graph has a smaller area than its parent node. Sorting in this way ensures that we always work on a child graph before working on the parent graph in case there are multiple nesting levels.

Conversion of Compound Graph to Simple Graph

The main idea behind using the TSM approach for compound graphs is to convert a compound graph to a simple graph. If the graph is simple (i.e., has no compound nodes), then the TSM layout can be directly applied to the input graph. The graph can later be mapped back to the original compound graph.

To convert a compound graph into a simple one, each of the compound nodes must be worked on to replace their boundaries and edge endpoints with simple nodes. For this purpose, we create a new graph G_s to store the "simple" version of the compound graph G. Initially, the simple/non-compound nodes of G are copied to G_s. After this, all intra-graph edges whose source and target both are simple nodes, are also copied to G_s. Then, each compound node is worked on one-by-one to convert it into its simple graph counterpart. The main steps in the conversion of a compound node to a simple node are explained as follows:

1. The four corner points of the compound node are replaced by four corner dummy nodes. These dummy nodes are added to the boundary list of the compound.

2. For each edge connected to the compound node, it is replaced by a dummy node at its point of contact with the compound boundary. Each of these dummy nodes is also added to the boundary list. The information of edge source and target is stored for later use.

3. A compound graph can be connected in its own context, but when converted into a simple graph, it may not be connected in the context of simple graphs. For this purpose, it is first determined if the graph is connected or not. A compound node is connected if there exists an edge from any of its nested children to any node outside the compound node. If this is not the case, then we need to connect one of the children of the compound node to the boundary of this compound. For this purpose, the left-most child node of the compound is detected and connected to the closest corner dummy node of the compound boundary. If the left-most node is a compound node, then an edge is created between the closest corners of the two compound nodes.

4. Edges are constructed between the boundary nodes of the compound node such that they form a rectangle just like the compound's original boundary.

When all of the above-mentioned steps have been performed for all compound nodes, new edges are created in G_s that connect the compound edges (edges

whose source or target might be a compound) to their new source or target. In this way, the compound structure of the graph is converted into a simple graph. The pseudocode for this conversion is provided in Algorithm 3.

Algorithm 3 Conversion of Compound Graph to Simple Graph

```
function ConvertToSimpleGraph(G, compoundNodes)
    Simple Graph, G_s ← null
    CopySimpleNodes(G, G_s)
    CopySimpleEdges(G, G_s)
    newEdgeList ← []
    for n in compoundNodes do
        n.boundaryList ← []
        n.boundaryList ← CreateCompoundCornerDummyNodes(G_s)
        for e in n.edges do
            dummyNode ← CreateDummyNodeForEdgeEndpoint(G_s)
            n.boundaryList.push(dummyNode)
            if e.source == n then
                newEdgeList.push(dummyNode, e.target)
            else
                newEdgeList.push(e.source, dummyNode)
            end if
        end for
        connected ← FindConnectivity(n)
        if connected ← false then
            ConnectGraph(G_S)
        end if
        ConnectCompoundBoundaryNodes(G_s, n.boundaryList)
    end for
    CreateNewEdges(G_s, newEdgeList)
    return G_s
end function
```

3.3.2 Modifications to the TSM Layout

After preprocessing the graph, the TSM layout is applied to the graph. This step is structured as shown in Algorithm 4.

Algorithm 4 TSM Layout Algorithm

```
function ApplyModifiedTSMLayout(G_s)
    Planarization(G_s)
    Orthogonalization(G_s)
    Compaction(G_s)
end function
```

Planarization

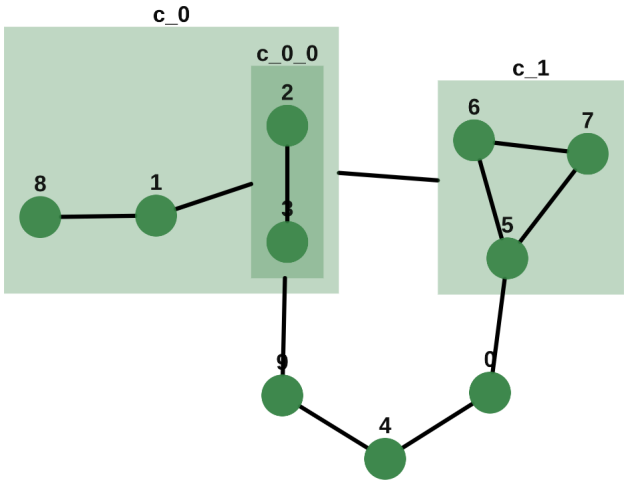
After the compound graph has been converted into a simple graph, we can now work on finding a planar embedding for the graph. Our algorithm makes a few modifications to the planarization method explained in 3.1.1. The implementation of the planarization step of TSM is explained as follows.

The initial CoSE layout can introduce edge crossings in the graph. We need to remove these edge crossings so that we can attain a planar embedding of the graph. Each edge crossing is replaced by a dummy node and dummy edges that connect the sources and targets of the intersecting edges. If any of the intersecting edges is an edge that belongs to the boundary of a compound, then the dummy node for such a crossing is also added to the boundary list of that compound node.

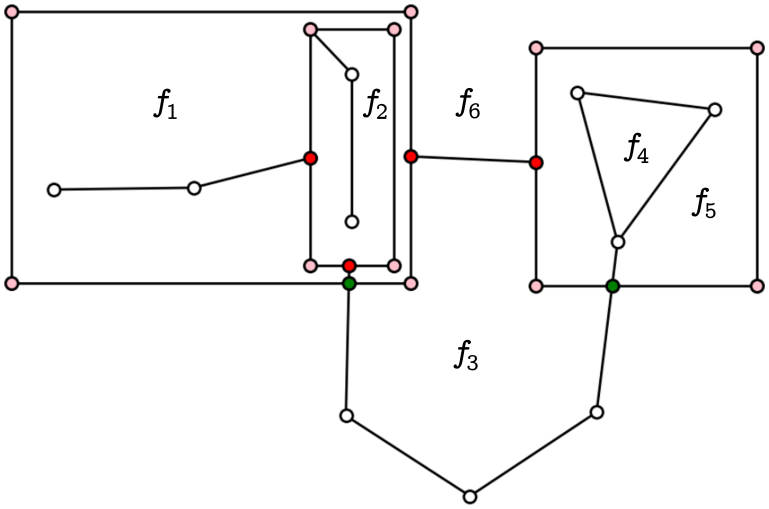
Figure 3.8 shows the resulting graph after the conversion of a compound graph to a simple graph and the insertion of dummy nodes for edge crossings.

After we have removed the edge crossings from the graph, the algorithm creates its planar embedding. This is done by finding the neighbors of each node n and sorting the neighbors in the counterclockwise direction. For each neighbor of the node n, a half-edge is added between these two nodes.

The next step is to create a doubly connected edge list for the connected planar graph. This structure contains the nodes of the original graph G, the half-edges for each edge e of G, and the faces in the graph. The faces are determined by traversing along the half-edges.



(a) A compound graph



(b) Planarized simple graph version of the compound graph in Figure 3.8a with marked faces

Figure 3.8: Conversion of Compound Graph to Simple Graph: Boundary of the compound nodes in Figure 3.8a are replaced with corner nodes (pink) in Figure 3.8b. Dummy nodes (red) are created on the boundary of compounds to mark the edge endpoints of compound nodes. The green nodes represent the dummy nodes for edge crossings. In Figure 3.8a, the child nodes of the compound node c_0_0 are not connected to any node outside of their parent. This causes the child nodes of c_0_0 to be "disconnected" from the rest of the graph. Node 2 is connected to the top left corner of c_0_0 to avoid disconnectedness.

Then the external face of the graph is found. The algorithm mentioned by P. Klose [18] tries to find a "maximal" external face because a non-maximal external face can result in a higher value of average edge length. However, in our case, we do not use the maximal face, but the actual outer face of the graph. This is necessary because we need the graph embedding to retain the topology of the actual graph.

We calculate the external face by finding the top-left corner node u in the graph and finding its slope with all of its neighbors. The neighbor v which has the largest slope with u belongs to the external face as well. The incident face of the half-edge connecting u and v is concluded to be the external face of the graph. This can be observed in Figure 3.8b. The top-left corner boundary node of c_0 (c_0-tl) is taken as starting reference. Then, we find its slope with its neighbors i.e. top-right corner of c_0 (c_0-tr) and bottom-left corner of c_0 (c_0-bl). Since the half-edge (c_0-tl, c_0-bl) has the largest slope, we take its incident face f_6 to be the external face.

Orthogonalization

The algorithm for orthogonalization is implemented as it is explained in section 3.1.2, except for one modification. The method for solving the minimum cost flow is different than the one that is described in the original paper.

Compaction

As mentioned in section 3.1.3, the first stage of compaction is to create dummy nodes at edge bendpoints. After that, the face side processor is used before making the faces rectangular. The working of these sub-algorithms is the same and changing their order does not affect the working of the overall algorithm. After the face sides have been determined, their data is updated side-by-side as the faces are turned rectangular. Initially, the algorithm works on turning the external face rectangular and then moves on to work on the internal faces.

When the compaction preprocessing is finished, the graph is compacted along the vertical and horizontal directions with the help of a network flow model. Then, the nodes are mapped on unique positions of a grid based on the edge lengths. The edge lengths determined by the TSM algorithm are unit values. These values are updated according to the ideal edge length.

As opposed to the original algorithm, we do not remove all dummy elements after grid layout because their data is needed by the further postprocessing steps of the C-TSM algorithm. In this regard, only the data of rectangular dummies is removed.

3.3.3 Postprocessing

This subsection explains the postprocessing steps unique to our algorithm that are performed after the TSM layout is finished. Since our algorithm converts the compound graph into a simple graph, it has to be turned back into a compound graph again. To achieve this, we firstly apply another compaction step to the graph. Then, the dummy elements are removed and the compound nodes are cropped and resized. Finally, we determine edge bendpoints and endpoints and display the final layout of the graph. The pseudocode of this step is explained in Algorithm 5.

Algorithm 5 Postprocessing algorithm pseudocode

```
function PostProcessGraph(G, G_s, compoundNodes)
    RemoveConnectivityEdges(G_s)
    CompactGraph(G_s)
    RemovePlanarizationDummies(G_s)
    UpdateElementData(G_s, G)
    Crop&ReshapeCompounds(compoundNodes)
    DetermineCompoundEdgeEndpoints(compoundNodes)
end function
```

Removing Connectivity Edges for Compounds

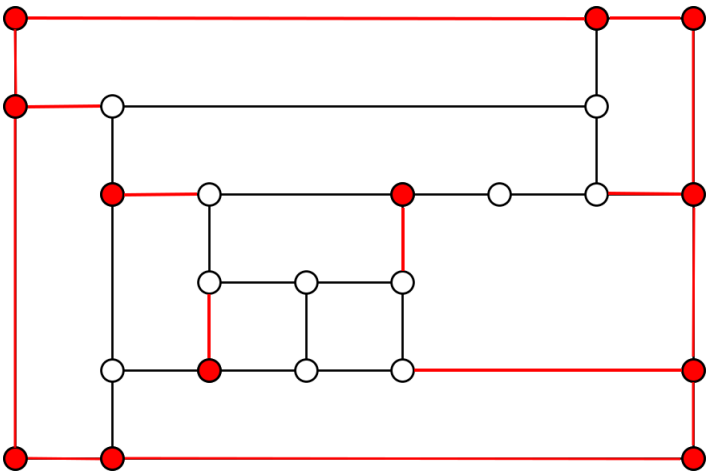
As mentioned in section 3.3.1, if none of the children of a compound node are connected to any node outside of the compound, a dummy connectivity edge is introduced from the compound node boundary to its leftmost child node. After the TSM layout is finished, the graph no longer needs such edges. These connectivity edges are hence removed.

Visibility Based Graph Compaction

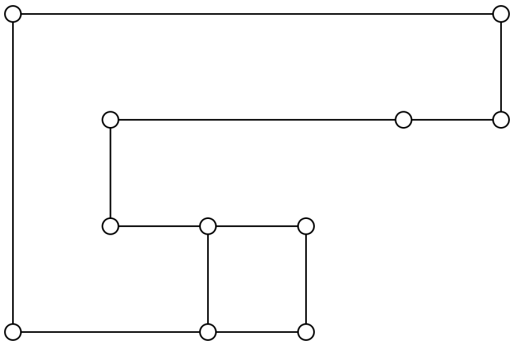
Even though the TSM approach does apply compaction on the graph, the graph still has extra space that can be removed. This is due to the creation of rectangular dummy nodes and edges which were generated to turn the faces of the graph rectangular. When their dummy data is removed in the compaction step of TSM, it creates extra space in the graph. The example of this phenomenon is shown in Figure 3.9. The three topmost edges in Figure 3.9b are long and can be further compacted. To deal with this issue, we perform another compaction step to reduce the extra spaces.

Since the compaction algorithm of TSM works only with rectangular faces, it cannot be used again here. Here, we use a simpler approach of compaction which can be used without creating extra dummy nodes or edges. We utilize the idea of "bar visibility graphs" which construct bars with the vertices and determine lines of visibility between those bars such that the source and target of these lines are in direct line of sight to each other. There has been a lot of work on this technique [37, 38, 39]. Hence, we utilize this method to perform compaction on our graph.

Our utilized technique is a one-dimensional compaction method. So, four different types of compactions are applied to the graph in a single pass of compaction. These compactions are as follows:



(a) Graph with rectangular dummy nodes and edges (marked in red)



(b) Graph after removal of the rectangular dummy nodes and edge in Figure 3.9a

Figure 3.9: Example of the creation of a non-compact drawing because of the rectangular face dummy elements

- Top-to-bottom compaction
- Bottom-to-top compaction
- Left-to-right compaction
- Right-to-left compaction

All of these four types of compaction are applied in a single pass, however, their order is alternated between the multiple compaction passes. Top-to-bottom and bottom-to-top are for vertical compaction of the graph and the other two are for performing horizontal compaction. The method for applying compaction in any of these directions is the same, but with a few minor changes. Algorithm 6 shows the working of this module.

Algorithm 6 One-dimensional compaction algorithm in a given direction

```
function CompactGraphInDirection(G_s, direction)
    axisDict ← CreateAxisDictionary(G_s, direction)
    bars ← Sort&CreateBars(axisDict, direction)
    CreateVisibilityGraph(bars, direction)
    DetermineNewBarLocations(bars, direction)
    UpdateBarLocations(bars, direction)
end function
```

The following steps explain the working of left-to-right horizontal compaction. Other directions are similar and have been left out for brevity.

1. Axis dictionary is created for the direction of compaction. For horizontal compaction, all nodes at similar x positions are determined. For example, the x-axis dictionary for Figure 3.10a is shown in Figure 3.10b.

2. For each value of x, we sort the nodes at this position in the ascending order of their y values. Starting with the first sorted node u, we add it to a new bar and then check if it is connected to the next sorted node v. If they are connected, the nodes are combined together to form a vertical "bar" whose length is the difference of y values of the top-most and bottom-most node in the bar. If these nodes are not connected, we move on to find the

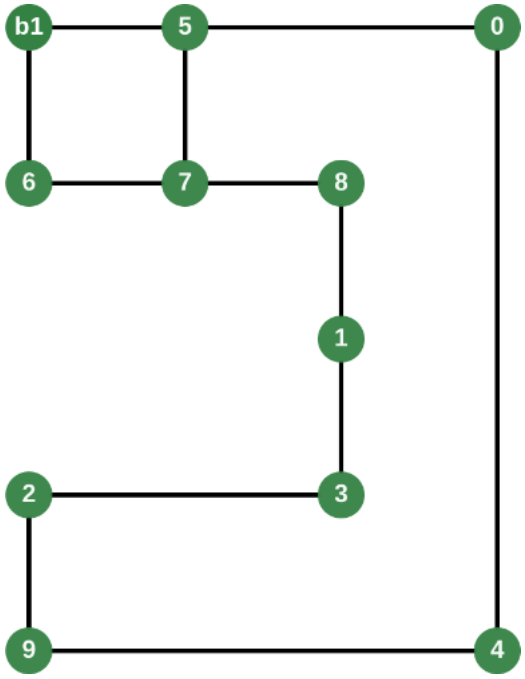
connectivity between v and the next sorted node w in the list. If v and w are also disconnected, then v is considered to be a point bar. This method is performed on all nodes, such that we have a collection of connected or unconnected nodes in the y-direction. Each bar is given a unique index value. Figure 3.10c shows the creation of these bars.

3. The next step is to create a visibility graph for the bars. The algorithm works on each bar b starting from the point (x, y_1) and ending at the point (x, y_2) and finds the bars that are directly visible to bar b in the y-range of (y_1, y_2) without any obstructions. The visibility of the bars mentioned in Figure 3.10c can be seen in Figure 3.10d. The arrows between bars show that they are visible to each other while going from the left to the right direction. This is how the visibility graph is constructed.

4. The location of the bars is updated based on the data from the visibility graph. Since the compaction is being done towards the right, the last bar (towards the right) is considered to be the reference bar. The bars are traversed in the opposite order, starting from the last one to the first. A bar is assigned one unit length to the right of the closet bar that it depends on. For example, in Figure 3.10d, bar 5 is the rightmost bar and its position is chosen as the starting reference. Bar 4 "sees" only bar 5 and is placed at one unit to the left of bar 5. Bar 3 sees 4 and 5 and is assigned to the right of the closest bar 4. Bar 2 sees 4 and 5 and is again moved one unit length to the right of 4. Bar 1 only sees 3 and is placed to its right. In this way, the edge lengths of {2, 3} and {9, 4} are compacted.

The final result of the left-to-right horizontal compaction can be seen in Figure 3.10e. The previously mentioned steps are modified slightly to work for the compactions in other directions. These modifications are mentioned in Figure 3.11.

Multiple passes for compaction are applied to the graph. After each pass, the overall area of the graph is calculated. If the area of the graph is the same as in the previous compaction pass, compaction is terminated. Otherwise, if the graph



(a) A sample graph that needs to be compacted

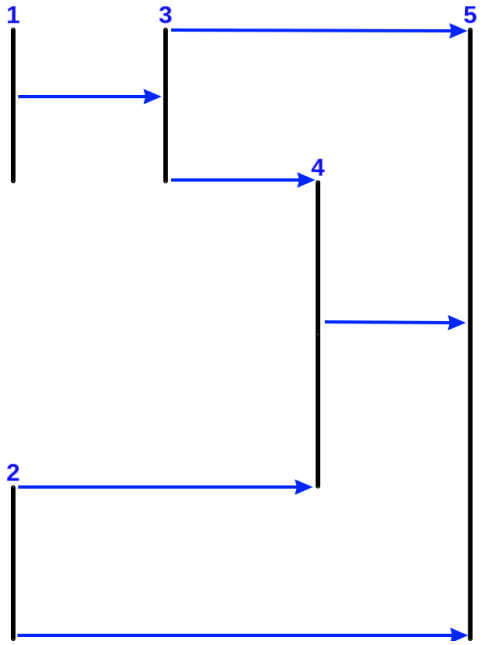
x-value	Nodes
100	2, 6, 9, b1
200	5, 7
300	1, 3, 8
400	0, 4

(b) x-axis node dictionary for Figure 3.10a

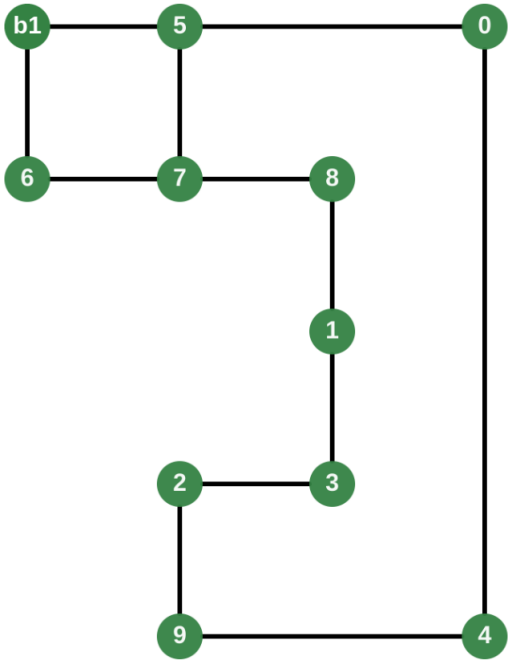
x-value	Bar Index	Bar Nodes
100	1	[b1, 6]
100	2	[2, 9]
200	3	[5, 7]
300	4	[8, 1, 3]
400	5	[0, 4]

(c) Vertical bars for horizontal compaction

Figure 3.10: Left-to-right horizontal compaction



(d) Visualization of the vertical bars mentioned in Figure 3.10c



(e) Final layout after compaction

Figure 3.10: Left-to-right Horizontal Compaction (cont.)

Compaction Direction	Axis Dictionary	Sorting criteria	Bars Topology	Visibility Graph Data	Base Position Reference	Bars Reference Criteria
Left-to-right	x-axis	Nodes at a x-location sorted according to increasing y-value	Vertical bars	Unobstructed bars inside the y-range lying towards the right	Right-most bar	Closest bar $b \in B$ to the right
Right-to-left	x-axis	Nodes at a x-location sorted according to increasing y-value	Vertical bars	Unobstructed bars inside the y-range lying towards the left	Left-most bar	Closest bar $b \in B$ to the left
Top-to-bottom	y-axis	Nodes at a y-location sorted according to increasing x-value	Horizontal bars	Unobstructed bars inside the x-range lying towards the bottom	Bottom-most bar	Closest bar $b \in B$ towards the bottom
Bottom-to-top	y-axis	Nodes at a y-location sorted according to increasing x-value	Horizontal bars	Unobstructed bars inside the x-range lying towards the top	Top-most bar	Closest bar $b \in B$ towards the top

Figure 3.11: Graph compaction algorithm for different directions

area keeps changing even after a fixed number of passes (k), then compaction is terminated after k passes to avoid an infinite loop.

Removing Planarization Dummy Data

Now that the compaction has been performed, we remove the dummy nodes and edges for edge crossings since they are no longer needed in the graph. These are replaced by their original edges.

Updating Element Data of the Original Graph

The positions of simple nodes are copied from the simple graph G_s to the original graph G. Furthermore, the data of bendpoints is also transferred to G.

Cropping and Reshaping Compounds

Finally, we move on to recreating the compound graph G from the simple graph G_s. Each compound node is firstly cropped. This needs to be done because the original rectangular boundary of the compounds may no longer remain rectangular after the TSM approach. Due to this, there may be some extra nodes on the compound boundary that do not allow the compound size to be minimally reduced in the compaction step. Such nodes have to be dealt with to further reduce the area of the compound nodes. Algorithm 7 shows the pseudocode for this step.

Similar to the compaction step, the compound nodes are also cropped in the four orthogonal directions: top, bottom, left, and right. As in the preprocessing step, we again work on the compounds with increasing node size (i.e. the smallest or innermost compounds are worked on first; before working on their parents). The cropping of the compounds is performed by the following method:

Algorithm 7 Cropping and Reshaping Compounds

```
function CROP&RESHAPECOMPOUNDS(compoundNodes)
    for n in compoundNodes do
        cornerNodes ← FINDCORNERCHILDNODES()
        UPDATEBOUNDARYLIST(n.boundaryList)
        CROPFROMTOP(n, cornerNodes, iel)
        CROPFROMBOTTOM(n, cornerNodes, iel)
        CROPFROMLEFT(n, cornerNodes, iel)
        CROPFROMRIGHT(n, cornerNodes, iel)
        UPDATENODEDATA(n)
        RESHAPENODEINCYTOSCAPE(n)
    end for
end function
```

1. Firstly, we find the four corner child nodes of the compound node. If a corner child node is a compound node, then we consider its relative corner side. In Figure 3.12a, the corner most child nodes are {1, 2, 4, 5}.

2. In the next step, the boundary list of the compound node is updated. In the preprocessing step of C-TSM, dummy nodes were introduced at the intersection point of other edges with the edges of the compound node boundary. These dummy nodes were also added to the boundary list of the compound since they lie on the compound node boundary. These dummies are removed from the graph while removing the planarization dummy data, but this information is not yet updated in the compound boundary list. Hence, they are now removed from the boundary list as well. The nodes of the compound boundary that were connected to this dummy node, are reconnected to each other.

3. Now, the compound has to be cropped in all four directions. This technique works for all of the directions with a few minor modifications. Here, we explain the cropping of the compound from the right side.

   (a) The compound boundary nodes are sorted based on decreasing x values, from right to left.

   (b) Given that x_{max} is the highest x value among the compound boundary nodes, it is chosen as the first boundary limit ($bdLimit1$).

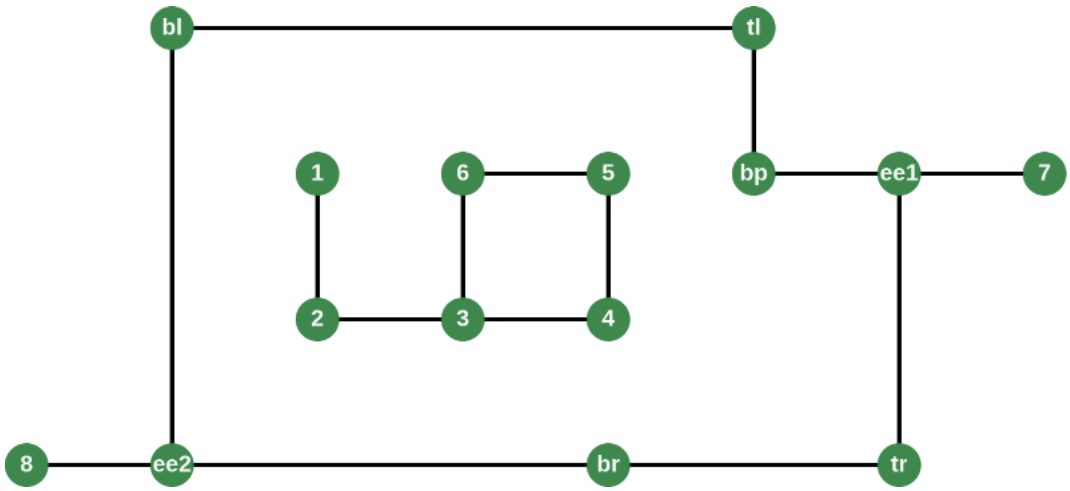
(c) Now, it is to be determined if it is possible to move the right boundary of the compound towards the center of the node. This is firstly determined by checking if the compound node has any edges that are connected to its top or bottom side at $bdLimit1$. If such nodes exist, then the compound cannot be cropped from the right direction, because those compound edges are connected to other nodes in the vertical direction, and moving them would disturb the whole layout of the graph. If there are no such edges, then we move on to confirm the next condition. To check the next condition, we need to determine the nodes lying at $bdLimit2 = bdLimit1 - iel$. If the nodes that are lying there are child nodes, then the cropping can again not be performed. If the above conditions are false and the nodes at $bdLimit2$ are all compound boundary nodes, then the compound can be cropped. To do this, the value of $bdLimit1$ is changed to $bdLimit2$ and the compound boundary nodes at $bdLimit1$ are pushed to $bdLimit2$. The above process is repeated until no more compaction is possible. The cropping algorithm from the right side is explained in Algorithm 8.

4. The height, width, and center of the compound node are updated based on the updated corner values of its boundary list.

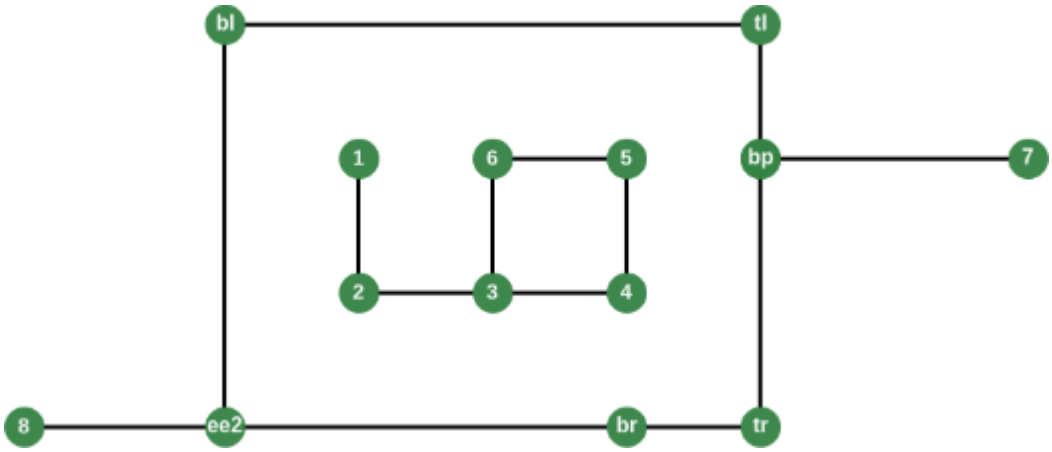
Figure 3.12 shows an example for cropping a compound node from the right side. The sorted boundary list for this graph is determined to be $(bl, ee2, br, tl, bp, ee1, tr)$. Since there are no edge crossings for this graph, we move on to the next step. We determine the boundary nodes at $bdLimit1 = x_{max}$ in the graph to be $ee1$ and tr. To conclude if the node can be cropped, we find the nodes (boundary nodes and child nodes) at $bdLimit2 = bdLimit1 - iel$, which are tl and bp, and they are both compound boundary nodes. Therefore, the compound node can be cropped by pushing the boundary nodes $ee1$ and tr to $bdLimit2$. $ee1$ overlaps with bp and is hidden behind it. Then, we perform the above process again by updating the value of $bdLimit1$ and $bdLimit2$ and checking the nodes at $bdLimit2$, which turn out to be child nodes. Hence, the compound node can not be compacted further from the right side and we terminate the process of compaction from this side.

Algorithm 8 Cropping Right Side of a Compound Node

```
function CropFromRight(n, cornerNodes, iel)
    right ← 0
    left ← 1
    SortBoundaryList(n.boundaryList)
    bdLimit1 ← GetCompoundBoundaryLimit(n.boundaryList)
    while true do
        for bdNode in n.boundaryList do
            if bdNode.x == bdLimit1 then
                edges ← bdNode.edges
                for e in edges do
                    bdNodeNbr ← GetOtherEnd(e, bdNode)
                    dir ← GetDirection(bdNode, bdNodeNbr)
                    if dir ≠ right || dir ≠ left then
                        return
                    end if
                end for
                bdLimit2 ← bdLimit1 − iel
                for c in cornerNodes do
                    if c.x == bdLimit2 then
                        return
                    end if
                end for
                for bdNode in n.boundaryList do
                    if bdNode.x == bdLimit1 then
                        bdNode.x ← bdLimit2
                    end if
                end for
                bdLimit1 ← bdLimit2
            else
                break
            end if
        end for
    end while
end function
```



(a) A simple graph version output of TSM for a compound node where *bl*, *tl*, *bp*, *ee*1, *tr*, *br*, *ee*2 comprise of the boundary of the compound, 1, 2, 3, 4, 5, 6 are the child nodes of the compound, and {*ee*1, 7}, {*ee*2, 8} represent the edges connected to the compound



(b) Cropped right side of the compound node in Figure 3.12a

Figure 3.12: Cropping of a compound node from its right side

These compactions are done from all four sides. After this is done, the next step is to redesign the compound nodes in Cytoscape.js. Based on the height, width, and center of a node, the compound node is reshaped and expanded in Cytoscape.js.

Determining Compound Edge Endpoints

Generally, the edges of nodes originate from the center of the node and they do not have a fixed endpoint on the node boundary. Since our conversion of a compound graph into a simple graph assumes a fixed endpoint for the compound boundary, this endpoint has to be realized on the compound boundary now. We take the position of the edge endpoints on the compound node boundary and convert them into edge endpoint values for Cytoscape.js.

3.3.4 Limitations

The limitations of the C-TSM approach are mentioned as follows:

- The boundary of a compound node does not retain its shape after TSM has been applied to it. It may change from a rectangular shape to an orthogonal polygon shape. The shape of a compound is always supposed to be rectangular so when such a polygon is converted into a rectangle based on the top-most, bottom-most, right-most, and left-most positions of its boundary, it overlaps with simple nodes or with other compound nodes.
- If a node is connected to a compound node with an edge e but overlaps with the compound due to the non-rectangular boundary issue, then e might overlap with other nodes or edges.

Chapter 4

Evaluation & Discussion

This chapter initially explains the implementation of our algorithm. After that, the algorithm is compared with the fCoSE algorithm and the results are explained and discussed.

4.1 Implementation

Our implementation of C-TSM is mainly based on Cytoscape.js in Javascript. The preprocessing and postprocessing parts of the C-TSM algorithm are implemented in Javascript, whereas the TSM approach implementation is in Python. The used TSM implementation is available at Github [40]. It uses the Networkx [41] library to manipulate graphs. Networkx also has a minimum cost flow solver that has been used to solve the flow networks in the orthogonalization and the compaction phase. The graph data is preprocessed and packaged and sent to the Python server where the TSM layout is applied on the graph. The server then returns the graph data to the Javascript client-side, where it is postprocessed. The final layout is then displayed using the Cytoscape.js styling options which are required to create bends in the graph and to expand the compound nodes beyond their normal sizes.

The testing has been performed on Linux Mint 20. The system contains an Intel i7-10510U @ 1.80GHz x 8 CPU with 16 GB RAM.

4.2 Comparison

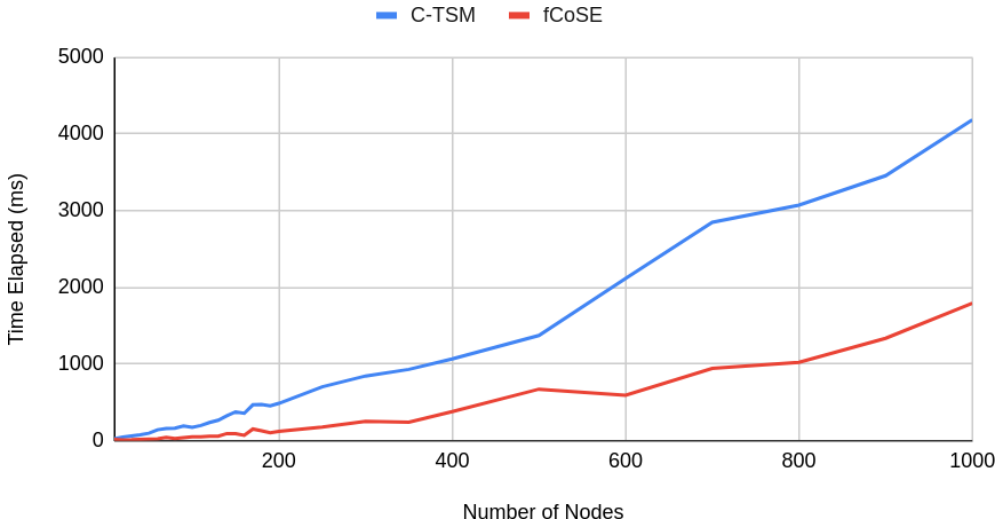
As mentioned before, there exists no explicit research work on an orthogonal layout for general compound graphs. Hence, we have compared our algorithm with fCoSE [25], which is a force-directed layout algorithm for drawing compound graphs with straight edges. The performance criteria which have been used for comparison are as follows:

- Runtime
- Number of edge crossings
- Node-edge overlaps
- Node-node overlaps
- Average edge length
- Total area

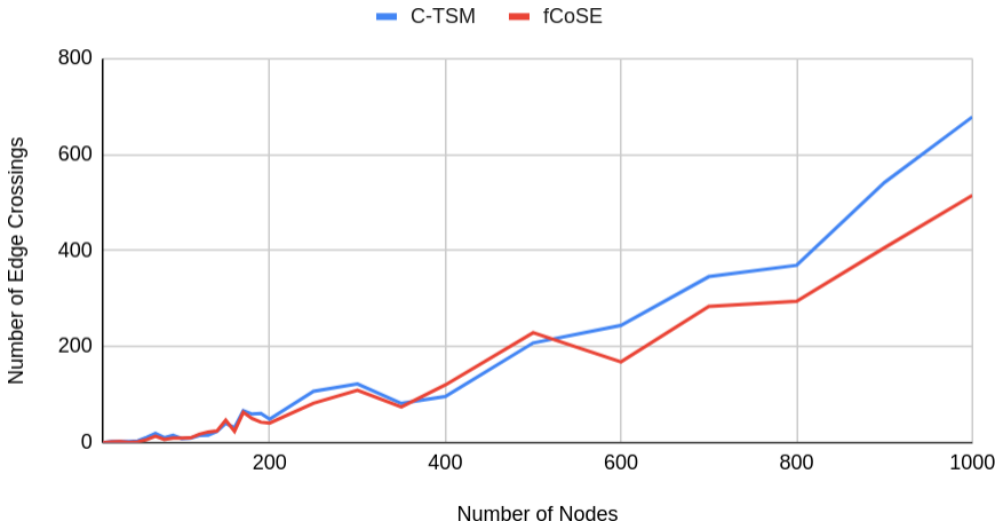
The test graphs have been generated by transforming the benchmark Rome graph dataset [42] into undirected, 4-degree compound graphs as detailed in [25]. The number of nodes in the test graphs vary as follows: 10, 20, 30, ..., 190, 200, 250, 300, 350, 400, 500, 600, ..., 1000. The data set has a total of 90 graphs where each graph size has 3 sample graphs.

Both C-TSM and fCoSE are run on each sample graph for five times. The average values of each criterion for a particular graph size are taken as the final evaluation values. The testing tries to determine the effect of the increase in the size of the graph on the chosen parameters, and how this increase affects the overall performance of the layout.

The results of the comparison of the two algorithms are shown in Figure 4.1.

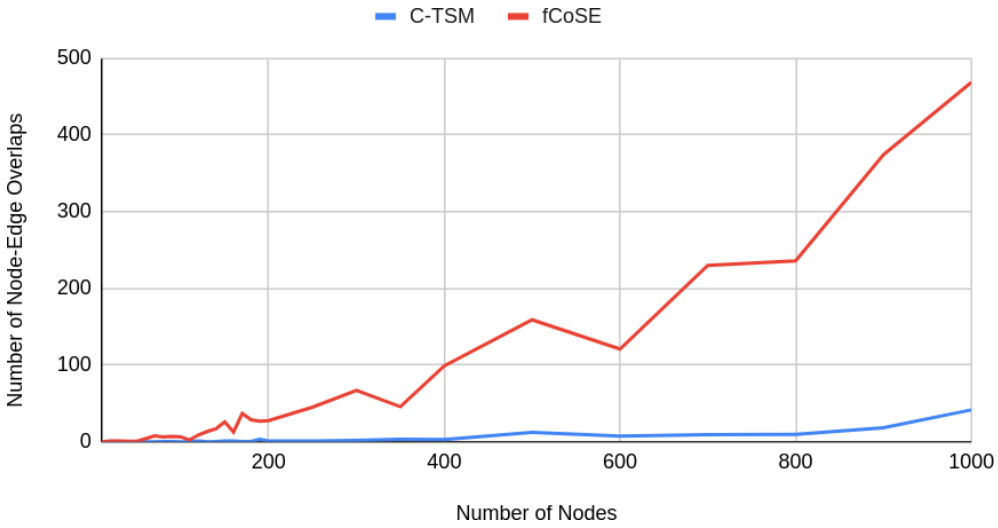


(a) Number of nodes vs time elapsed

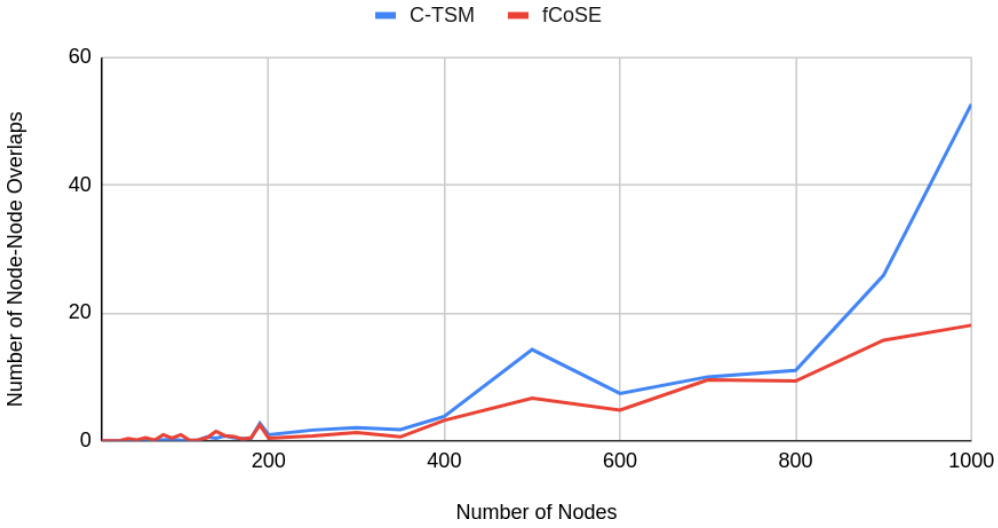


(b) Number of nodes vs number of edge crossings

Figure 4.1: Comparison of different performance criteria between C-TSM and fCoSE

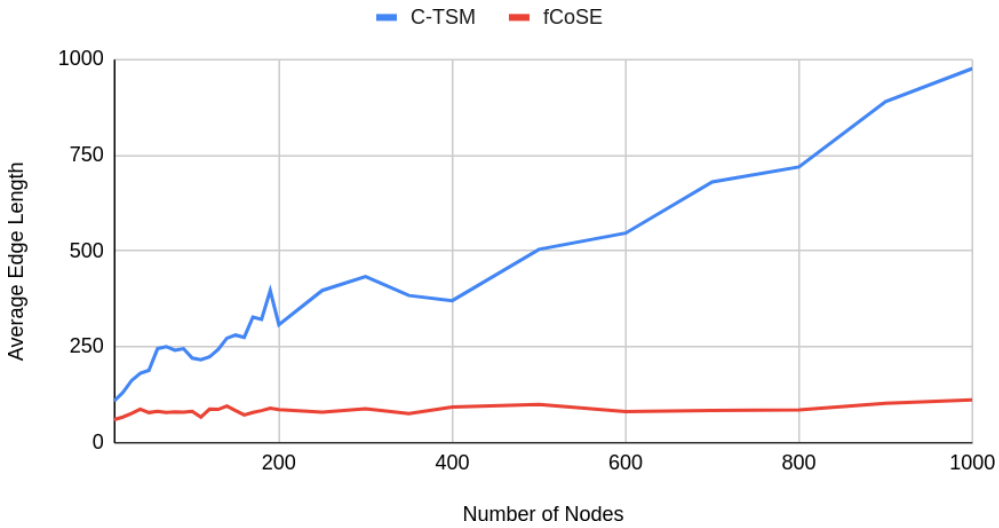


(c) Number of nodes vs number of node-edge overlaps

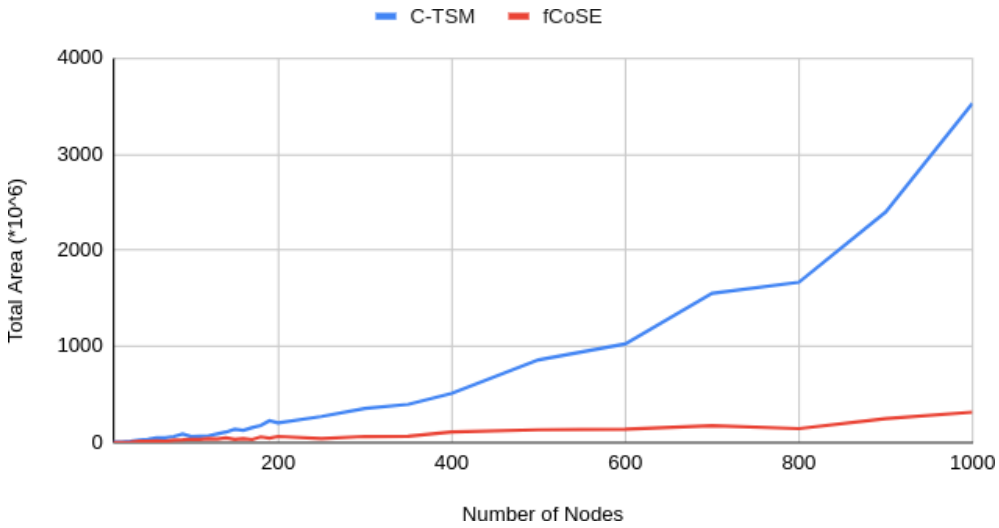


(d) Number of nodes vs number of node-node overlaps

Figure 4.1: Comparison of different performance criteria between C-TSM and fCoSE (cont.)



(e) Number of nodes vs average edge length



(f) Number of nodes vs total area

Figure 4.1: Comparison of different performance criteria between C-TSM and fCoSE (cont.)

4.3 Results & Discussion

The C-TSM algorithm takes a considerably greater time to run than the fCoSE algorithm. This is an expected output because orthogonal layouts are generally slower than straight edge drawings. Straight-edge (mostly force-directed) layout algorithms typically run in $\mathcal{O}(n^2)$ time (e.g., Frutcherman & Reingold runs in $\mathcal{O}(n^2)$ when used with the grid variant technique), whereas orthogonal layouts generally take a longer time to produce the output. The minimum cost flow solver for the flow networks that is utilized in the orthogonalization and compaction phase of the TSM algorithm, has a worst-case complexity of $\mathcal{O}(n^3)$. But overall, the complexity depends on the orthogonal representation of the data and can sometimes be even completed in linear time [18]. This is because network flow graphs are specialized simple graphs which rarely hit the worst-case scenario and work in linear time on average.

The number of edge crossings in both algorithms are similar to each other. This is because the number of edge crossings in C-TSM are dependant on the number of edge crossings in the initial CoSE layout. The edge crossings from the initial layout are carried forward to the final C-TSM layout. Since fCoSE is a faster and improved version of the CoSE algorithm, it also has a similar number of edge crossings in its output.

fCoSE has a visibly greater number of node-edge overlaps because it is a force-directed layout. Such layout algorithms often form clusters of simple nodes which have a lot of node-edge overlaps and they pay no attention to such overlaps explicitly. C-TSM has a lower number of these overlaps. Another point to note here is that the TSM algorithm ensures that there are no node-edge overlaps at the cost of increased drawing area. But when the graph is converted back into a compound graph, the compound nodes may overlap with the simple nodes and edges and this is where such overlaps come from in the C-TSM algorithm.

No two simple nodes ever overlap in the layout of C-TSM. The node-node overlaps originate from the compound nodes overlapping with other compound

nodes and simple nodes. These values are comparable to the fCoSE values for small-sized graphs but increase as the graph size approaches 1000 nodes.

As expected, the area and average edge lengths of orthogonal drawings are also larger as compared to straight-edge force-directed algorithms because the neighbors of a node are assigned to horizontal and vertical directions. Such assignments often increase the edge lengths considerably because two connected nodes may end up in totally different directions from each other. Furthermore, the creation of bendpoints further increases the length of an edge.

To summarize, our algorithm fares relatively well for graphs up to 400-500 nodes but does not scale well to medium or large-sized graphs.

Figure 4.2 through 4.9 show some layouts produced with the C-TSM algorithm.

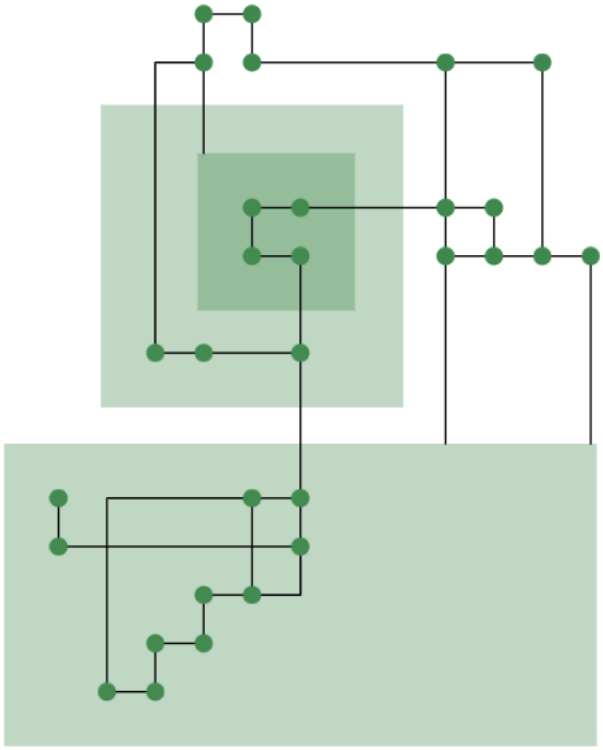


Figure 4.2: C-TSM layout on a graph with 30 nodes

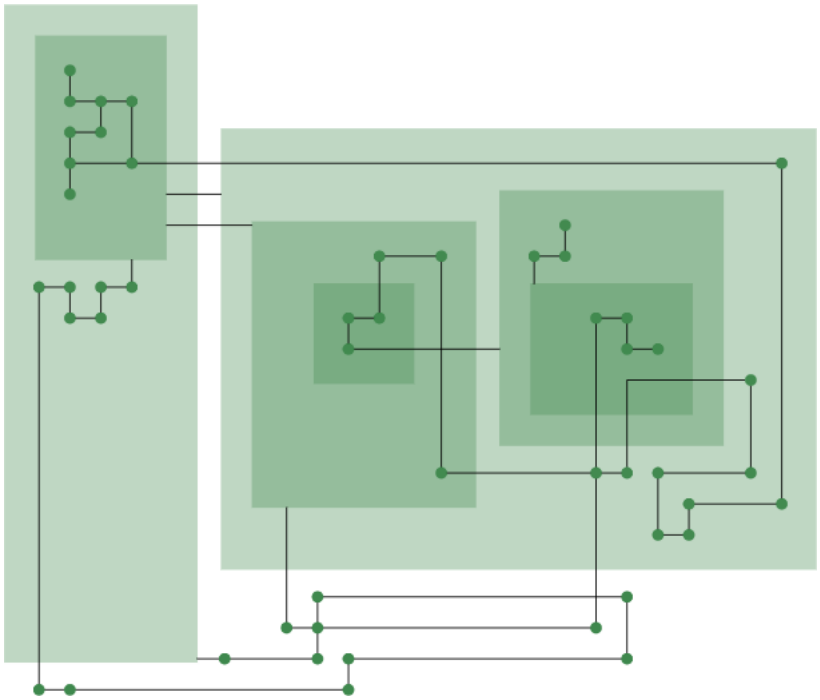


Figure 4.3: C-TSM layout on a graph with 50 nodes

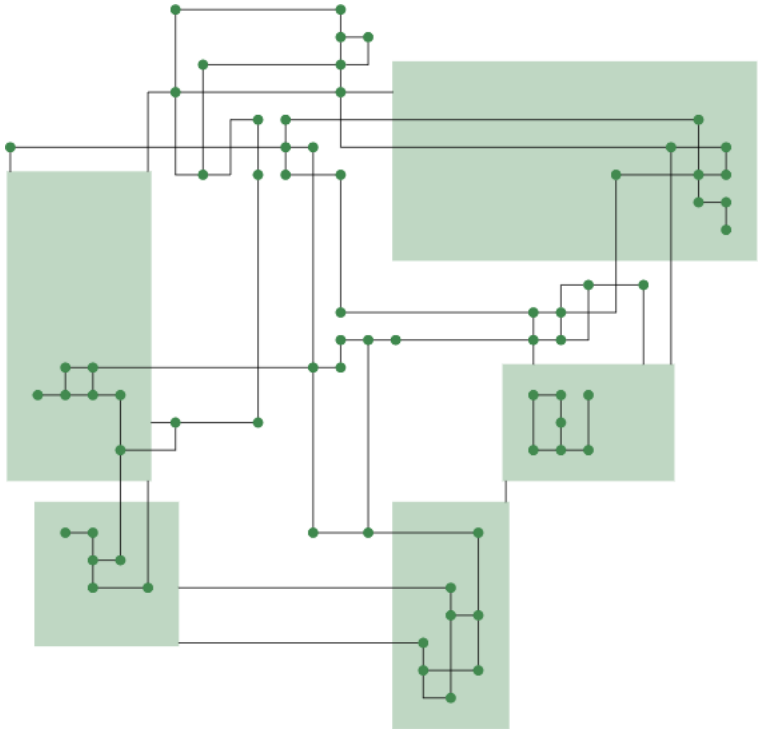


Figure 4.4: C-TSM layout on a graph with 70 nodes

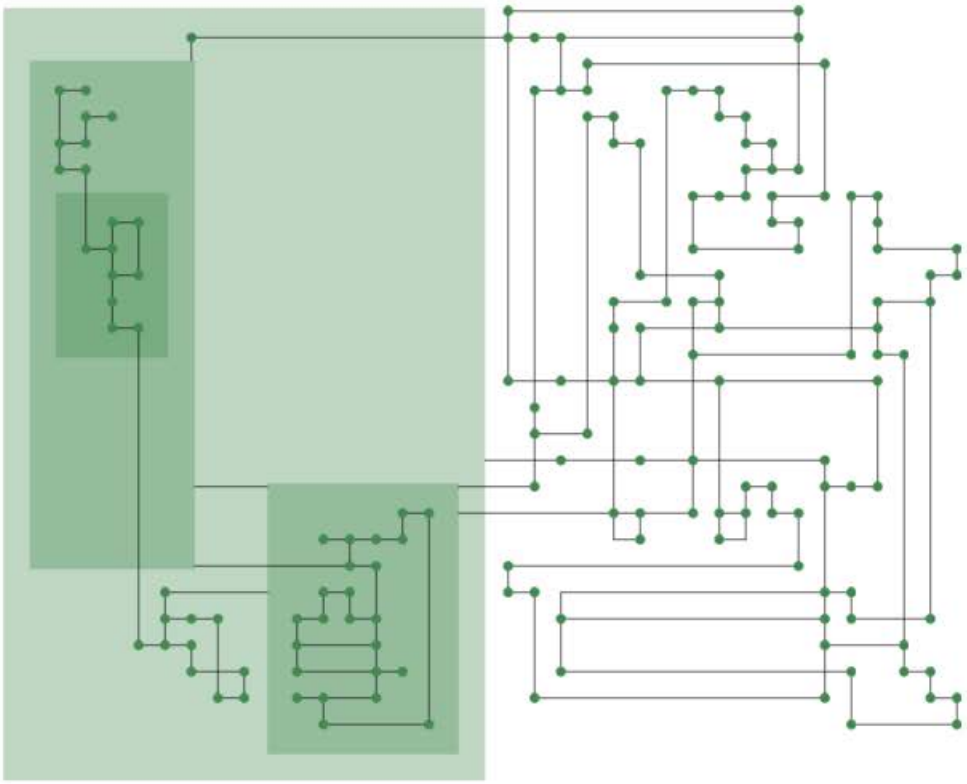


Figure 4.5: C-TSM layout on a graph with 160 nodes

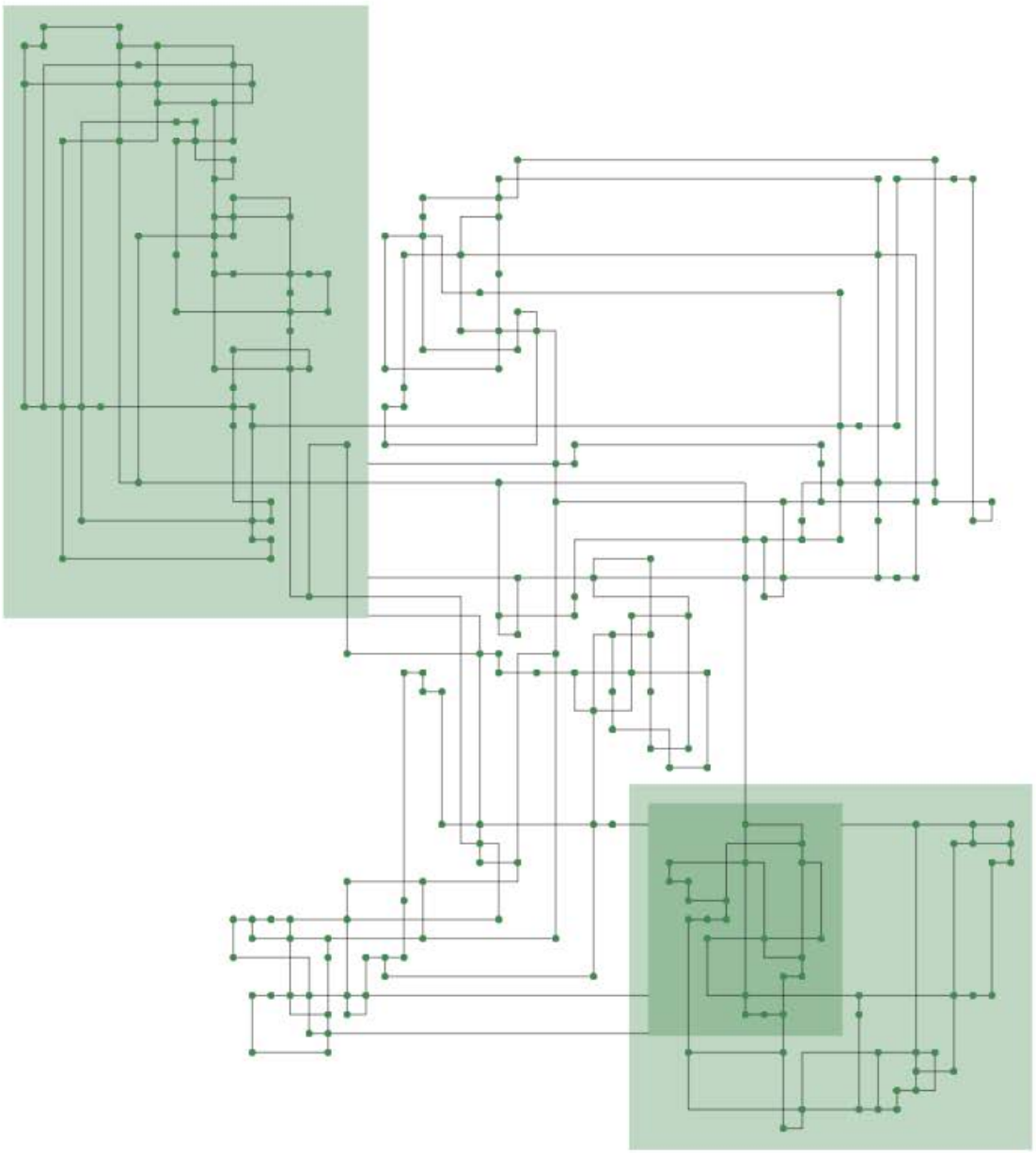


Figure 4.6: C-TSM layout on a graph with 250 nodes

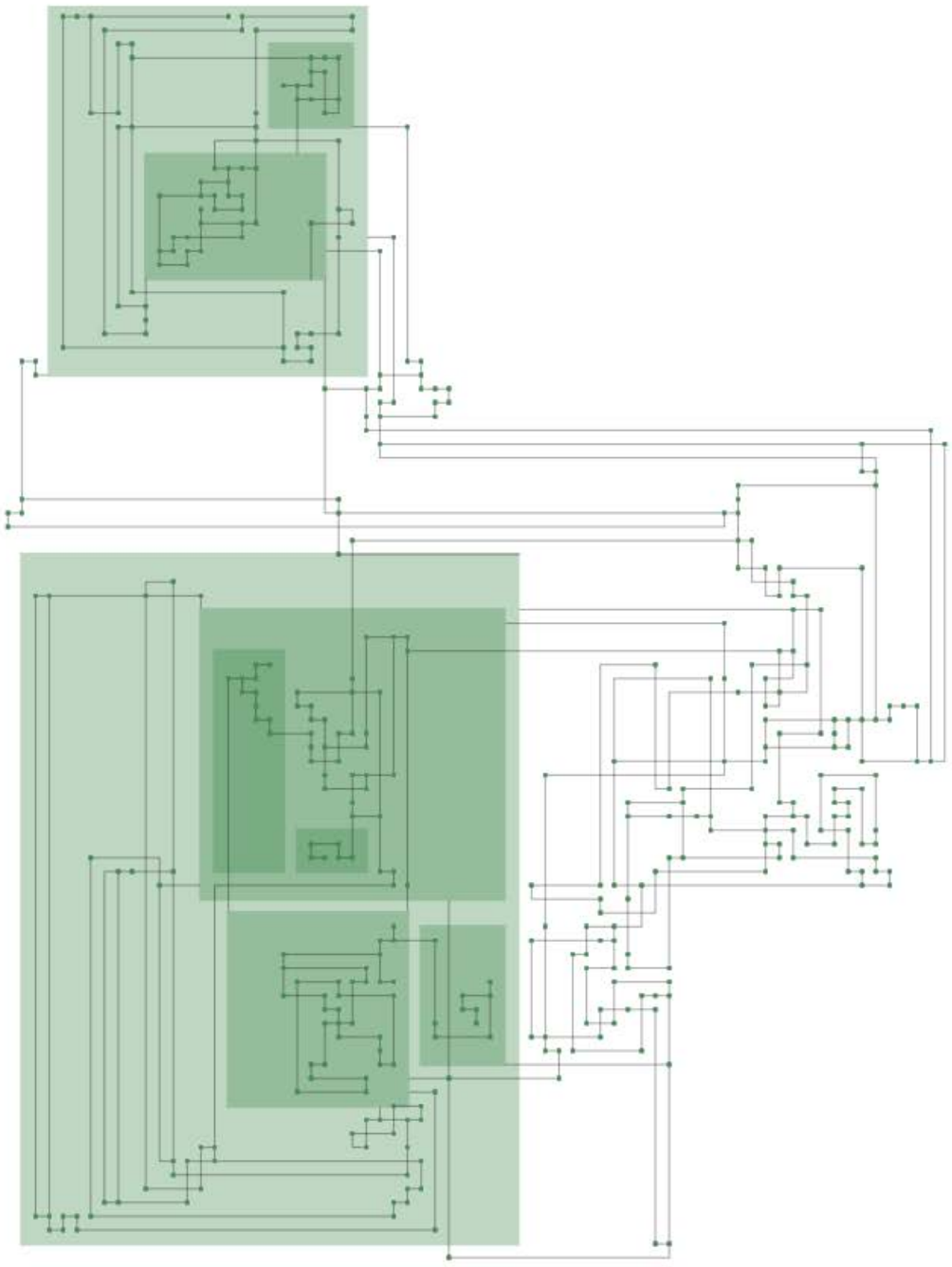


Figure 4.7: C-TSM layout on a graph with 400 nodes

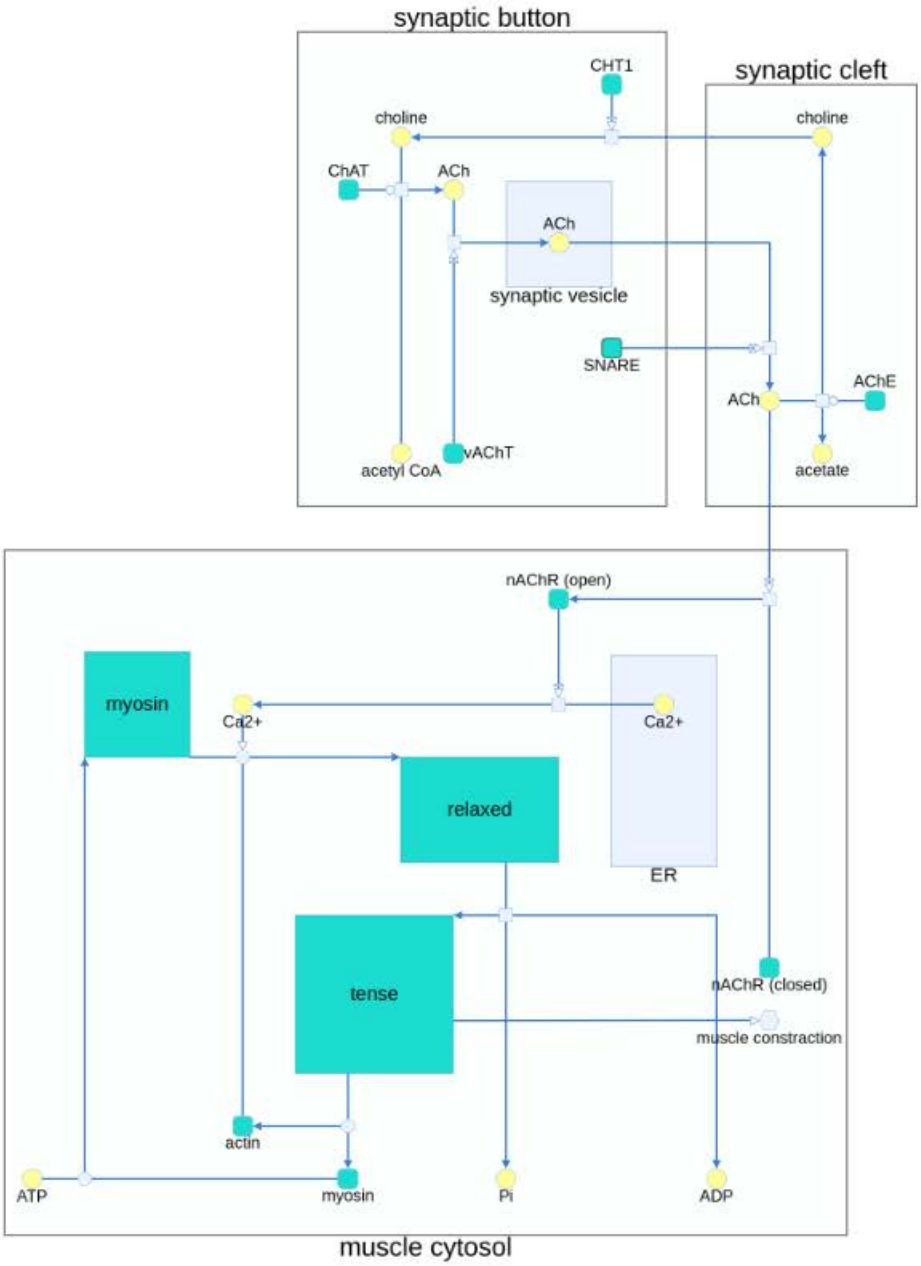


Figure 4.8: C-TSM layout on neuronal muscle signalling diagram [3]

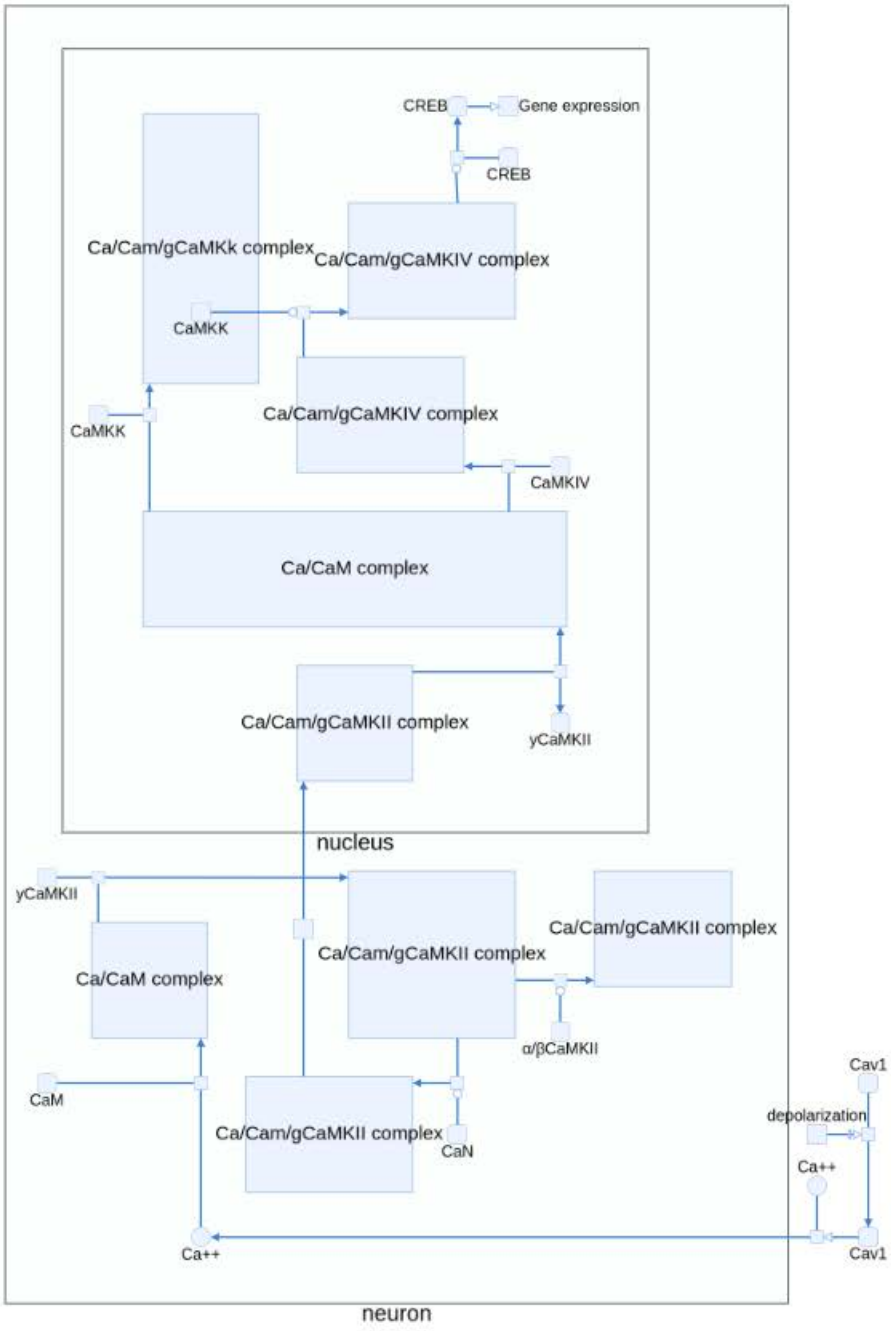


Figure 4.9: C-TSM layout on CaM/CamK dependant signalling to nucleus [3]

Chapter 5

Conclusion

The goal of this thesis was to create an orthogonal layout algorithm for small compound graphs with uniform node sizes. We introduced the C-TSM layout algorithm as the first publicly available orthogonal layout algorithm that works for general compound graphs. For this, we used the Topology-Shape-Metrics approach to work for compound graphs such that these graphs were converted into simple graphs and were converted back to compound graphs after the application of the modified TSM approach. The postprocessing steps include visibility-based compaction to further reduce the drawing area.

The algorithm was compared with the fCoSE layout algorithm for compound graphs. From the experimental results, it can be seen that the C-TSM algorithm is suitable to work for small-sized compound graphs with uniform node dimensions. These drawings can be obtained in execution times of at most a few seconds, appropriate for interactive usage, producing satisfactory results based on the performance criteria with room for improvements.

The implementation of the C-TSM algorithm can be found on GitHub as a Cytoscape.js extension named cytoscape.js-c-tsm [43].

5.1 Future Improvements

The following are the improvements that can be made to the layout algorithm to improve the layout.

- Algorithm can be modified to work for multi-edges and self-loops.
- The current layout still leaves some empty spaces after cropping of the compound nodes. This can be handled by two options. The first is that the compound nodes can be cropped before compaction, but this will require another postprocessing step of converting the graph into a compound graph and then converting it back into a simple graph for compaction. The other option is to apply another compaction step after the cropping of compounds.
- Our compaction algorithm sometimes causes an increase in the edge length which can be avoided. Resolving this issue is also part of future work.
- The current algorithm only works on 4-degree graphs. It can be expanded to work on higher degree graphs.
- In the present algorithm, we assume the simple nodes to be of uniform node sizes. Future work could entail catering for non-uniform node sizes. A trivial fix would be to increase the edge lengths based on the node with the greatest dimensions. But this would lead to non-uniform edge lengths and a higher graph area.

Bibliography

[1] S. Card, *Information Visualization*, pp. 509–543. 01 2008.

[2] J. Desjardins, "How much data is generated each day?." `https://www.weforum.org/agenda/2019/04/how-much-data-is-generated-each-day-cf4bddf29f/`, 2019.

[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," *Bioinformatics (Oxford, England)*, p. btaa850, 2020.

[4] U. Dogrusoz, M. E. Belviranli, and A. Dilek, "Cise: A circular spring embedder layout algorithm," *IEEE transactions on visualization and computer graphics*, vol. 19, no. 6, pp. 953–966, 2012.

[5] G. D. Battista, P. Eades, R. Tamassia, and I. G. Tollis, *Graph drawing: algorithms for the visualization of graphs*. Prentice Hall PTR, 1998.

[6] 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.

[7] T. Sawyer, "Tom sawyer software." `https://www.tomsawyer.com/`, 1992.

[8] R. Wiese, M. Eiglsperger, and M. Kaufmann, *yFiles — Visualization and Automatic Layout of Graphs*, pp. 173–191. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004.

[9] U. Rüegg, S. Kieffer, T. Dwyer, K. Marriott, and M. Wybrow, "Stress-minimizing orthogonal layout of data flow diagrams with ports," in *International Symposium on Graph Drawing*, pp. 319–330, Springer, 2014.

[10] C. D. Schulze, M. Spönemann, and R. Von Hanxleden, "Drawing layered graphs with port constraints," *Journal of Visual Languages & Computing*, vol. 25, no. 2, pp. 89–106, 2014.

[11] M. Siebenhaller, S. S. Nielsen, F. McGee, I. Balaur, C. Auffray, and A. Mazein, "Human-like layout algorithms for signalling hypergraphs: outlining requirements," *Briefings in bioinformatics*, vol. 21, no. 1, pp. 62–72, 2020.

[12] S. Kieffer, T. Dwyer, K. Marriott, and M. Wybrow, "Hola: Human-like orthogonal network layout," *IEEE transactions on visualization and computer graphics*, vol. 22, no. 1, pp. 349–358, 2015.

[13] R. Tamassia, "On embedding a graph in the grid with the minimum number of bends," *SIAM Journal on Computing*, vol. 16, no. 3, pp. 421–444, 1987.

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

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

[16] J. S. Yi, Y. ah Kang, J. Stasko, and J. A. Jacko, "Toward a deeper understanding of the role of interaction in information visualization," *IEEE transactions on visualization and computer graphics*, vol. 13, no. 6, pp. 1224–1231, 2007.

[17] 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, no. 5, p. e0197238, 2018.

[18] P. Klose, "A generic framework for the topology-shapemetrics based layout," *Christian-Albrechts-Universität zu Kiel*, 2012.

[19] Brilliant.org, "Flow network." `https://brilliant.org/wiki/flow-network/`.

[20] H. C. Purchase, R. F. Cohen, and M. James, "Validating graph drawing aesthetics," in *International Symposium on Graph Drawing*, pp. 435–446, Springer, 1995.

[21] R. J. Lipton, S. C. North, and J. S. Sandberg, "A method for drawing graphs," in *Proceedings of the first annual symposium on Computational geometry*, pp. 153–160, 1985.

[22] D. Ferrari and L. Mezzalira, *On drawing a graph with the minimum number of crossings*. Politecnico, 1969.

[23] H. C. Purchase, "Metrics for graph drawing aesthetics," *Journal of Visual Languages & Computing*, vol. 13, no. 5, pp. 501–516, 2002.

[24] L. K. Klauske, "Efficient editing of simulink models with the help of a specifically adapted layout algorithm," 2012.

[25] H. Balci and U. Dogrusoz, "fCoSE: a fast compound graph layout algorithm with constraint support," *IEEE Transactions on Visualization and Computer Graphics*, 2021.

[26] T. C. Biedl, B. P. Madden, and I. G. Tollis, "The three-phase method: A unified approach to orthogonal graph drawing," in *International Symposium on Graph Drawing*, pp. 391–402, Springer, 1997.

[27] K. Freivalds and J. Glagoļevs, "Graph compact orthogonal layout algorithm," in *International Symposium on Combinatorial Optimization*, pp. 255–266, Springer, 2014.

[28] P. Bertolazzi, G. Di Battista, and W. Didimo, "Computing orthogonal drawings with the minimum number of bends," *IEEE Transactions on Computers*, vol. 49, no. 8, pp. 826–840, 2000.

[29] T. Bläsius, S. Lehmann, and I. Rutter, "Orthogonal graph drawing with inflexible edges," *Computational Geometry*, vol. 55, pp. 26–40, 2016.

[30] M. Eiglsperger, M. Kaufmann, and M. Siebenhaller, "A topology-shape-metrics approach for the automatic layout of uml class diagrams," in *Proceedings of the 2003 ACM symposium on Software visualization*, pp. 189–ff, 2003.

[31] L. Barth, B. Niedermann, I. Rutter, and M. Wolf, "A topology-shape-metrics framework for ortho-radial graph drawing," *arXiv preprint arXiv:2106.05734*, 2021.

[32] P. Shannon, A. Markiel, O. Ozier, N. S. Baliga, J. T. Wang, D. Ramage, N. Amin, B. Schwikowski, and T. Ideker, "Cytoscape: a software environment for integrated models of biomolecular interaction networks," *Genome research*, vol. 13, no. 11, pp. 2498–2504, 2003.

[33] R. Saito, M. E. Smoot, K. Ono, J. Ruscheinski, P.-L. Wang, S. Lotia, A. R. Pico, G. D. Bader, and T. Ideker, "A travel guide to cytoscape plugins," *Nature methods*, vol. 9, no. 11, pp. 1069–1076, 2012.

[34] M. Franz, C. T. Lopes, G. Huck, Y. Dong, O. Sumer, and G. D. Bader, "Cytoscape. js: a graph theory library for visualisation and analysis," *Bioinformatics*, vol. 32, no. 2, pp. 309–311, 2016.

[35] Z. Shaukat, S. Aiman, T. Zhou, C. Li, *et al.*, "A systems biology-driven approach to construct a comprehensive protein interaction network of influenza a virus with its host," *BMC Infectious Diseases*, vol. 20, no. 1, pp. 1–10, 2020.

[36] T. Dwyer, Y. Koren, and K. Marriott, "Ipsep-cola: An incremental procedure for separation constraint layout of graphs," *IEEE Transactions on Visualization and Computer Graphics*, vol. 12, no. 5, pp. 821–828, 2006.

[37] F. Luccio, S. Mazzone, and C.-K. Wong, "A note on visibility graphs," *Discrete Mathematics*, vol. 64, no. 2-3, pp. 209–219, 1987.

[38] S. K. Wismath, "Characterizing bar line-of-sight graphs," in *Proceedings of the first annual symposium on Computational geometry*, pp. 147–152, 1985.

[39] T. C. Shermer, "On rectangle visibility graphs. iii. external visibility and complexity," in *Proceedings of the 8th Canadian Conference on Computational Geometry*, pp. 234–239, 1996.

[40] uknfire, "tsmpy." `https://github.com/uknfire/tsmpy`, 2019.

[41] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring network structure, dynamics, and function using networkx," in *Proceedings of the 7th Python in Science Conference* (G. Varoquaux, T. Vaught, and J. Millman, eds.), (Pasadena, CA USA), pp. 11 – 15, 2008.

[42] 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.

[43] M. Zaman, "C-tsm." `https://github.com/iVis-at-Bilkent/cytoscape.js-c-tsm`, 2021.