FAST COMPOUND GRAPH LAYOUT WITH CONSTRAINT SUPPORT

A DISSERTATION SUBMITTED TO
THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR
THE DEGREE OF
DOCTOR OF PHILOSOPHY
IN
COMPUTER ENGINEERING

By
Hasan Balcı
August 2022

FAST COMPOUND GRAPH LAYOUT WITH CONSTRAINT SUPPORT

By Hasan Balcı

August 2022

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

Uğur Doğrusöz(Advisor)

Fazlı Can

Alper Şen

Ali Aydın Selçuk

İsmail Hakkı Toroslu

Approved for the Graduate School of Engineering and Science:

Orhan Arıkan

Director of the Graduate School

ABSTRACT

FAST COMPOUND GRAPH LAYOUT WITH CONSTRAINT SUPPORT

Hasan Balcı
Ph.D. in Computer Engineering
Advisor: Uğur Doğrusöz
August 2022

Visual analysis of relational data becomes more challenging in today's world as the amount of data increases exponentially. Effective visual display of such data is therefore a key requirement to simplify the analysis process. Compound graphs present a practical structure for both representing the relational data with varying levels of groupings or abstractions and managing its complexity. In addition, a good automatic layout of these graphs lets users understand relationships, uncover new insights and find important patterns hidden in the data. To this end, we introduce a new layout algorithm named fCoSE (fast Compound Spring Embedder) for compound graphs with support for user-specified placement constraints. fCoSE combines the speed of spectral layout with the aesthetics and quality of force-directed layout while satisfying specified constraints and properly displaying compound structures. The algorithm first generates a draft layout with the help of a spectral approach, then enforces placement constraints by using newly introduced heuristics and finally polishes the layout via a force-directed layout algorithm modified to maintain enforced constraints. Our experiments performed on both real-life and randomly generated graphs verify that fCoSE outperforms its competitors in terms of both speed and generally accepted graph layout criteria and is fast enough to be used in interactive applications with small to medium-sized graphs.

Keywords: Information visualization, graph layout, visual analytics, compound graphs, constrained layout, spectral graph drawing.

ÖZET

KISIT DESTEKLİ HIZLI BİLEŞİK ÇİZGE YERLEŞTİRME

Hasan Balcı
Bilgisayar Mühendisliği, Doktora
Tez Danışmanı: Uğur Doğrusöz
Ağustos 2022

Günümüz dünyasında veri miktarı katlanarak arttığı için ilişkisel verilerin görsel analizi daha zor hale gelmektedir. Bu nedenle, bu tür verilerin etkili görsel gösterimi, analiz sürecini basitleştirmek için önemli bir gerekliliktir. Bileşik çizgeler, hem farklı düzeylerde gruplamalar veya soyutlamalar içeren ilişkisel verileri temsil etmek hem de onların karmaşıklığını yönetmek için pratik bir yapı sunar. Ek olarak, bu çizgelerin otomatik ve iyi yerleşimi, kullanıcıların ilişkileri anlamalarına, yeni içgörüler ortaya çıkarmasına ve verilerde gizlenmiş önemli kalıpları bulmasına olanak tanır. Bu amaçla, bileşik çizgeler için, kullanıcı tarafından belirlenen yerleştirme kısıtlamalarını da destekleyen fCoSE adlı yeni bir yerleştirme algoritması sunuyoruz. fCoSE, belirtilen kısıtlamaları karşılarken ve bileşik yapıları düzgün bir şekilde görüntülerken, izgesel yerleştirmenin hızını, kuvvet-yönlendirilmiş yerleştirmenin estetiği ve kalitesi ile birleştirir. Önce izgesel bir yöntem yardımıyla taslak bir yerleşim oluşturur, daha sonra ilk defa sunulan buluşsal yöntemleri kullanarak yerleştirme kısıtlamalarını sağlar ve son olarak, sağlanmış olan kısıtlamaları sürdürmek için değiştirilmiş kuvvet-yönlendirilmiş bir bileşik çizge yerleştirme yöntemi aracılığıyla yerleşimi güzelleştirir. Hem gerçek dünya hem de rastgele oluşturulmuş çizgeler üzerinde gerçekleştirilen deneylerimiz, fCoSE'nin hem hız hem de genel kabul görmüş çizge yerleşim kriterleri açısından rakiplerini geride bıraktığını ve küçük ila orta ölçekli çizgeleri destekleyen etkileşimli uygulamalarda kullanılabilecek kadar hızlı olduğunu göstermektedir.

Anahtar sözcükler: Bilgi görselleştirme, çizge yerleştirme, görsel analiz, bileşik çizgeler, kısıtlı yerleştirme, izgesel çizge yerleştirme.

Acknowledgement

I would like to express my deepest gratitude to Prof. Dr. Uğur Doğrusöz, who is more than an advisor to me, for his invaluable motivation, guidance and support throughout my PhD journey. Working with him was one of the best decisions I have ever made and he will be one of my role models for the rest of my life.

I would like to thank Prof. Dr. Varol Akman, Prof. Dr. Alper Şen and Prof. Dr. Fazlı Can for being in my thesis monitoring committee and for their valuable advice during the meetings. I would also like to thank Prof. Dr. Ali Aydın Selçuk and Prof. Dr. İsmail Hakkı Toroslu for accepting to be on my thesis jury.

I would also like to acknowledge The Scientific and Technological Research Council of Turkey (TÜBİTAK) for providing me with financial support for this thesis within the scope of ARDEB 1001 project with grant number 118E131 and TEYDEB 1505 project (in Bilkent-Huawei collaboration) with grant number 5180088.

Special thanks to my valuable friends Mustafa Can Çavdar, Arif Usta, Aytek Aman, Hüseyin Eren Çalık, Alihan Okka, İlkin Safarli and Fırat Gezer for making this difficult journey easy and enjoyable for me. I also feel lucky to have my dear friends Akifhan Karakayalı, Serkan Demirci and Sinan Sonlu who liven up our conversations with different perspectives. I would also like to thank all the faculty members, graduate students and staff in the CS Department who made my day, even if it was just by saying hello.

Last but not least, I would like to express my deepest gratitude to my parents, Alime and İbrahim Balcı, and my brothers Salih and Sacit Balcı, who have always supported me throughout my life.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

As the data generated increases exponentially in today's world, it becomes more challenging to derive the necessary insights from the data. Therefore, converting data into a comprehensible form is a key necessity to make it valuable. It is known that visually representing data makes it more understandable [12]. The field of *information visualization* comes into play at this point. It is the practice of representing data in a meaningful and visual way so that users can easily understand it. Information visualization comes in different flavors such as chart, histogram, cartogram, heatmap, and graph, each suitable for representing a specific type of data. If data elements contain relational connections among them, as with most real-world data, then they can be visualized with graphs, where data elements are represented with *nodes* and relations between them are represented with *edges*. *Graph visualization* is used in numerous domains from biological pathways to social networks and architecture diagrams (Figure 1.1 through Figure 1.3).

Sometimes, relational data may also contain nesting relationships as in Figure 1.1 and Figure 1.3. *Compound graphs* [8] are very useful to both represent such data and manage its complexity with their sophisticated structures.

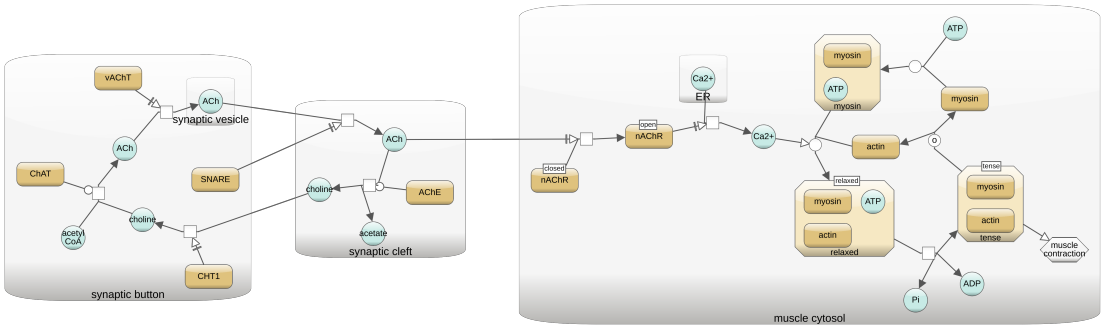


Figure 1.1: A sample biological pathway - neuronal muscle signaling [1]

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

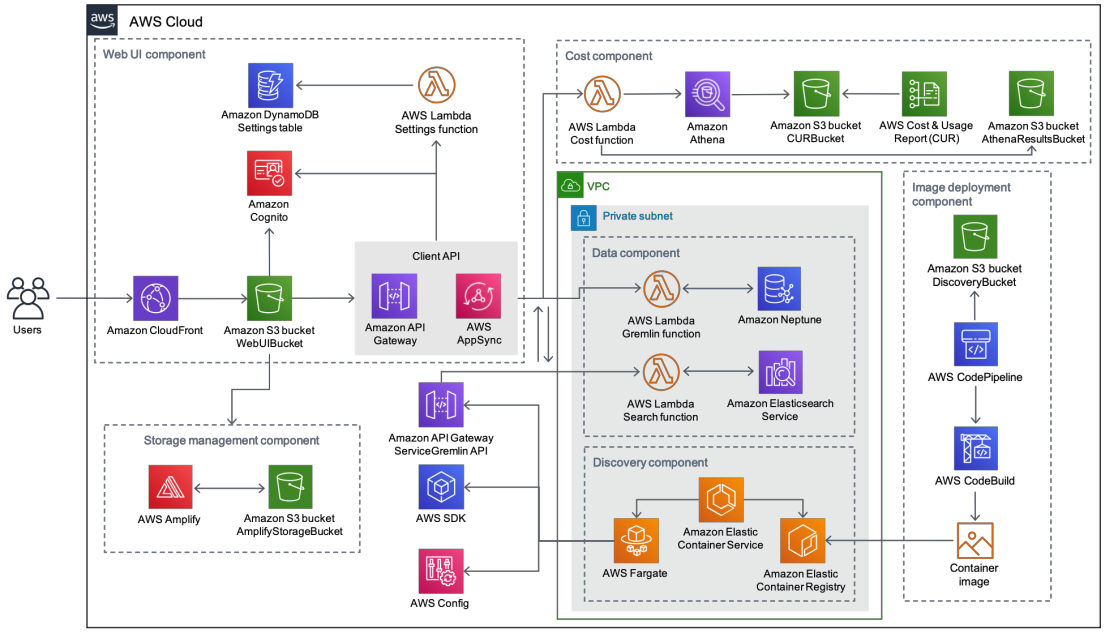


Figure 1.3: Architecture diagram of AWS Perspective, a visualization tool for cloud workloads. [3]

An important requirement in visually representing relational data via graphs is the arrangements of nodes and edges, i.e. the *layout* of the graph. A good layout is important since a poor one may mislead the user and manual layout adjustments can take up to 25% of a typical user's time [13]. The quality of the layout is generally decided by some commonly accepted graph layout criteria proposing metrics such as the number of edge crossings, average edge length and maximization of the symmetry. Even though layout algorithms are free to arrange graph elements to optimize these metrics, some domains may additionally require user-specified constraints to be applied on the layout (Figure 1.4 and Figure 1.5).

There have been numerous work done on graph layout [14]; however, the number of studies that focus on both compound graphs and layout constraints is very limited. The existing ones suffer from various weaknesses such as insufficient support for compound structures, ignoring varying node dimensions or high computational cost which are important for most real-life applications. In this thesis, we present a new graph layout algorithm named fCoSE to overcome these deficiencies of existing algorithms.

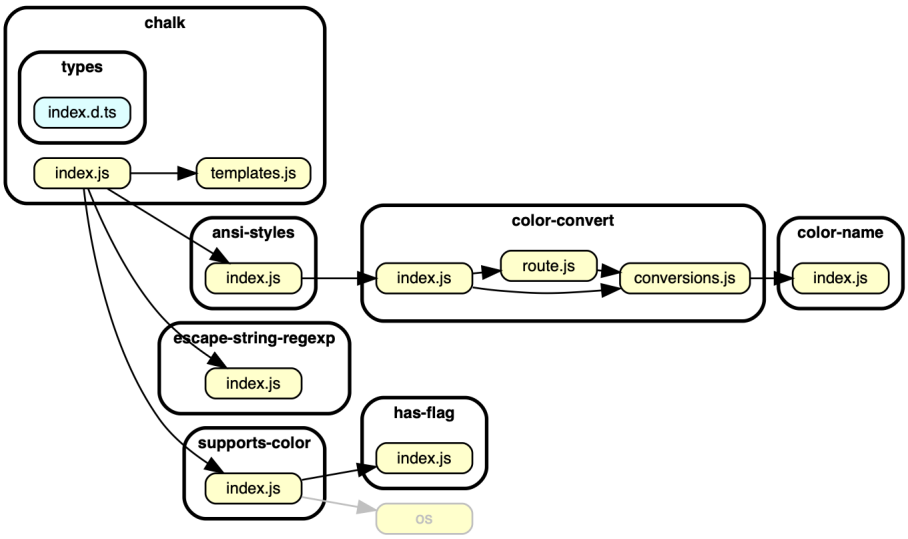


Figure 1.4: A sample layout of a dependency graph that positions nodes (JavaScript files) from left to right based on their dependency relationships and vertically aligns those at the same level [4]

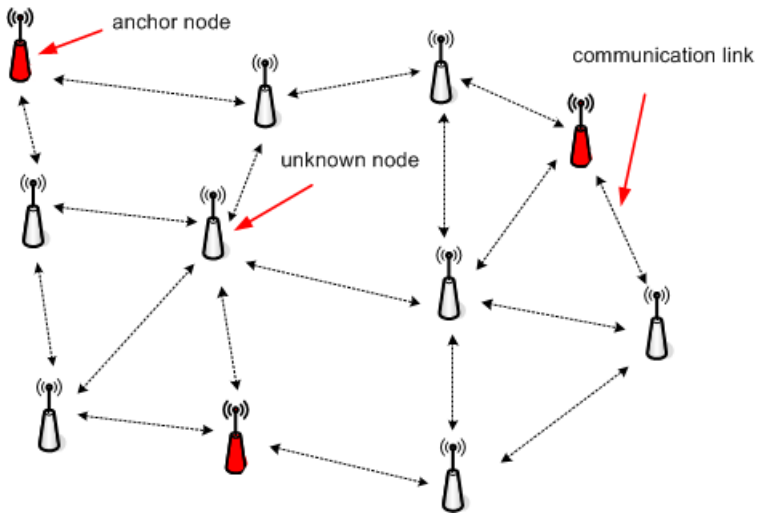


Figure 1.5: A sample layout of a wireless sensor network in which the anchor nodes are positioned to their predefined locations while the other nodes are laid out based on the measured inter-sensor distances [5]

1.2 Contribution

fCoSE proposes a new approach fully capable of handling compound graphs, supports a fairly rich set of user-specified placement constraints and runs fast enough on small to medium-sized graphs. This makes fCoSE suitable to be used in interactive visualization tools. It combines the speed of spectral layout with the aesthetics of force-directed layout, and in this way, it runs up to 2 times as fast as CoSE [8], which is its base method, and outperforms its competitors with support for both constraints and compound structures in terms of generally accepted aesthetic criteria and run time performance. In addition, to the best of our knowledge, fCoSE is the first algorithm that suggests a heuristic to apply the spectral layout approach on compound graphs.

The rest of the thesis is organized as follows. Chapter 2 explains some basic concepts related to graphs, layout algorithms and some mathematical concepts used in this thesis. Chapter 3 presents a literature review of related concepts. The details of the fCoSE algorithm are explained in Chapter 4, while Chapter 5 gives an evaluation of the experimental results. Chapter 6 concludes the thesis with possible future work and the availability of the algorithm.

Chapter 2

Basics

2.1 Graphs

A *graph* $G = (V, E)$ is an abstract structure where V is a non-empty set of nodes (vertices) and E is a set of edges that connect node pairs (u, v) where $u, v \in V$. The *source* and *target* nodes of an edge is said to be *adjacent* to each other. A node is *incident* to an edge if it is either source or target node of that edge. The number of edges incident upon a node is called the *degree* $d(v)$ of that node. Average degree of a graph $d(G) = \frac{1}{|V|}\sum_{v \in V} d(v) = \frac{2|E|}{|V|}$ can be used to decide how dense that graph is. A graph is called a *weighted graph* if its edges have associated numerical values (weights).

A *path* in a graph is a non-repeating sequence of nodes such that each of its nodes is connected to the next node in the sequence by an edge. A graph is *connected* if there is at least one path between any two nodes, *disconnected* otherwise. A special case of a connected graph where any two nodes are connected by exactly one path is called *tree*. A tree in which a node is assigned as the *root* is a *rooted tree*.

A graph is *directed* if its edges have a direction. A path that starts and ends

at the same node is called *cycle*. A *directed cycle* is a cycle in a directed graph whose all edges are oriented in the same direction. A directed graph with no directed cycles is called *directed acyclic graph (dag)*. A directed graph is *weakly connected* if its undirected version is connected.

A *topological sort* of a dag $G = (V, E)$ is a linear ordering of nodes in V in such a way that for each directed edge $e = (u, v) \in E$, u comes before v in the ordering.

The *induced subgraph* $G[X] = (V_x, E_x)$ of a graph $G = (V, E)$ is the graph where $V_x \subset V$ and $E_x = \{(u, v) \mid u \in V_x \wedge v \in V_x\}$.

A *compound* graph $G = (V, E, F)$, which is useful for representing graphs with both inclusion and adjacency relationships, consists of a set of nodes V, a set of (adjacency) edges E, and a set of inclusion edges F [15]. The inclusion tree $T = (V, F)$ is a rooted tree, defined on the set of nodes V and set of inclusion edges F, which represents the hierarchical structure of a compound graph. It is assumed that $E \cap F = \emptyset$; in other words, a node cannot be connected to one of its ancestors or descendants by an adjacency edge (Figure 2.1). A leaf node in the inclusion tree is called *simple* node, while a non-leaf node that contains a graph inside is a *compound* node. A compound node is the *parent* of the nodes inside the *child* graph or subgraph *nested* within it. An edge $e \in E$ is called *inter-graph* edge when it connects two vertices from different levels of the inclusion tree, and *intra-graph* edge otherwise. Compound nodes are useful structures to represent data with varying levels of groupings or abstractions and especially to manage the complexity of such large data through expand-collapse operations [16].

2.2 Graph Layout

Graph layout is the arrangement of the nodes and edges in a graph in order to achieve a usable, understandable and aesthetic representation of the graph. This arrangement can be achieved via a function that maps each node to a distinct

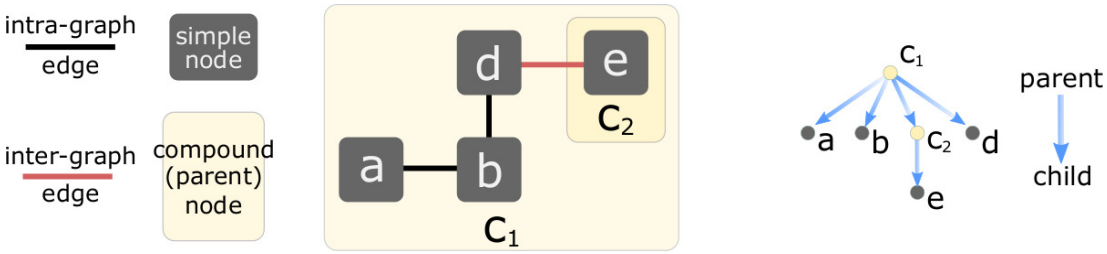
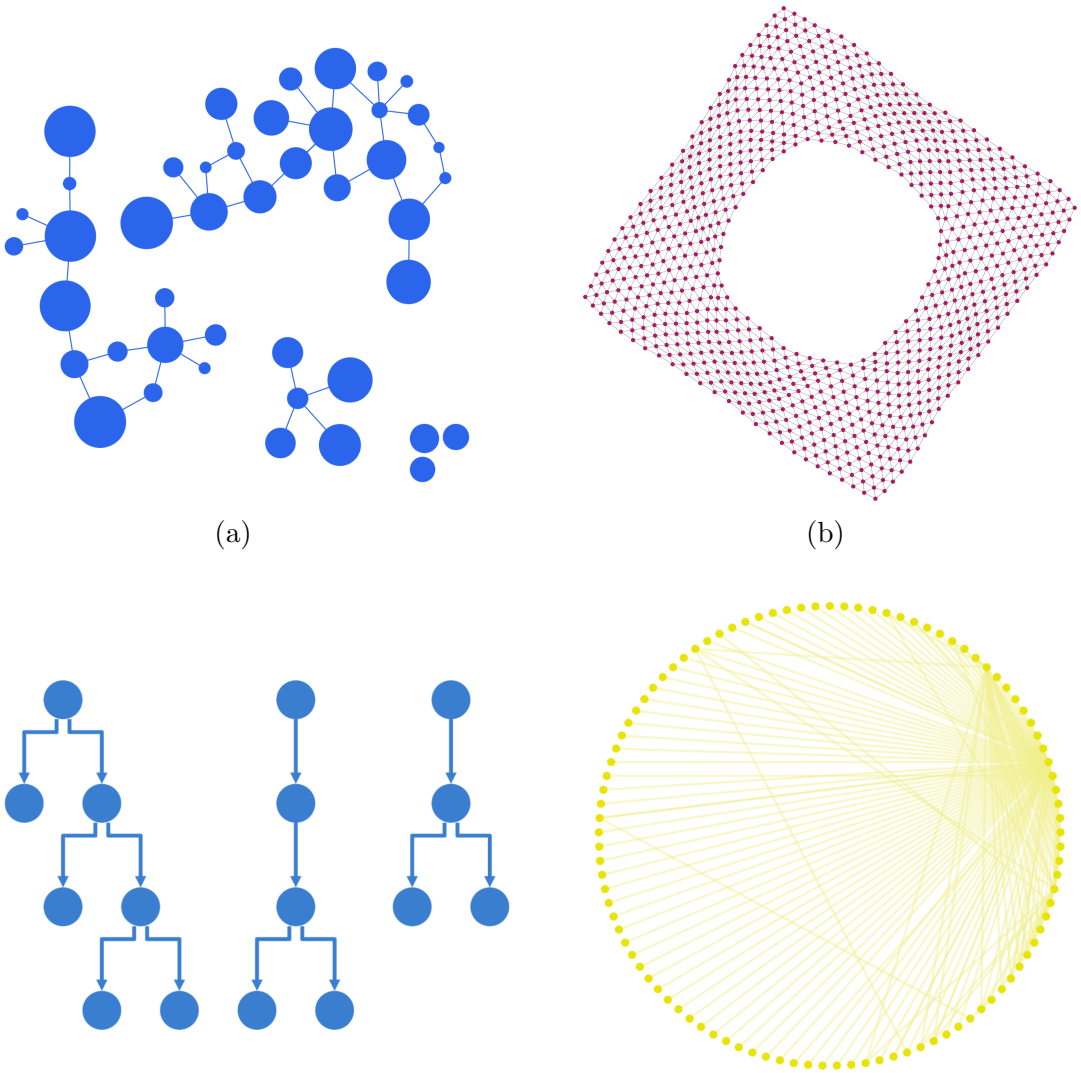


Figure 2.1: An example compound graph $G = (V = \{a, b, d, e, c1, c2\}, E = \{\{a, b\}, \{b, d\}, \{d, e\}\}, F = \{(c_1, a), (c_1, b), (c_1, d), (c_1, c_2), (c_2, e)\})$ where c_1 and c_2 are the compound nodes and $\{d, e\}$ is the single inter-graph edge, along with the corresponding inclusion tree on the right

point in 2D/3D space and each edge to a Jordan curve in the same space where endpoints correspond to the positions of the respective end nodes of the edge [14]. The quality of a graph layout is a subjective concept; however, there is some generally accepted criteria that define the aesthetics to achieve a quality layout [17]. According to these criteria, a good layout displays symmetry, avoids edge crossings, keeps edge lengths uniform and distributes vertices uniformly. A good layout algorithm is also expected to take into account any non-uniform node dimensions to avoid node-node overlaps. It is obvious that a child node is drawn within the bounding box of its parent compound node in compound graphs.

A layout of a graph is generally produced from scratch by assigning random positions to nodes initially and then adjusting them accordingly to achieve a good layout. However, if a user is satisfied with the current positioning of the nodes and just wants to "tidy them up" by maintaining his/her mental map, then an *incremental layout* can be applied by starting the layout from current positions of the nodes and applying incremental polishing changes.

There are many types of graph layout algorithms such as force-directed, spectral, hierarchical, orthogonal and circular, where some of these algorithms can be used for general purposes, while others are more appropriate for specific types of graphs (Figure 2.2). Of special interest in the scope of this thesis are force-directed and spectral layout algorithms, the first of which can produce aesthetically good results in both simple and compound graphs, while the other is renowned for its speed.



(a) (b)

(c) (d)

Figure 2.2: Sample results of various graph layout algorithms: a) force-directed layout, b) spectral layout, c) hierarchical layout with orthogonal edges and d) circular layout [6]

2.2.1 Force-directed Layout Algorithms

One of the most popular methods in automatic graph layout is the force-directed layout (aka spring embedders) [7], which is based on the idea of applying physical analogies to graph drawing. This idea dates back to Eades' study [18] where nodes are represented as metal rings that repel each other and edges are represented as springs that exert forces to the rings on both ends proportional to the deviation from the "ideal" length. To obtain a layout, nodes are first given random positions and then the system is released to apply a physical simulation where nodes are moved based on the total forces acting upon them iteratively. As a consequence of the repeated iterations, the system reaches a fairly stable state where the total energy in the system is minimized (Figure 2.3). A few remarks about Eades' approach are that it calculates repulsion forces only between non-adjacent nodes and uses a logarithmic formula to calculate spring forces instead of Hooke's law. The run time complexity of his approach is $\mathcal{O}(|V|^2 + |E|)$ per iteration where $\mathcal{O}(|V|^2)$ for the calculation of the repulsion forces and $\mathcal{O}(|E|)$ for the calculation of attractive (spring) forces.

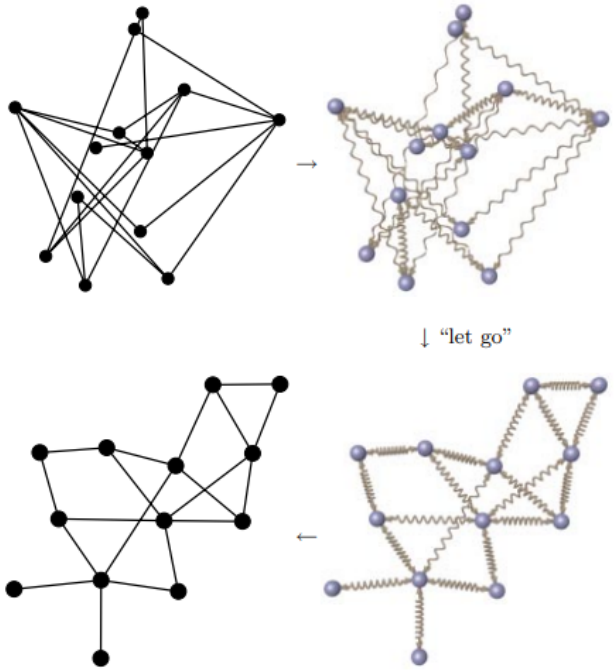


Figure 2.3: An illustration of the force-directed approach [7]

There are two pioneering studies inspired by Eades' force-directed approach. Kamada and Kawai [19] use only spring forces between all pairs of nodes by assuming each node pair is connected with a spring where the ideal lengths of the springs are calculated based on the graph-theoretical distances between node pairs. They treat the layout problem as a process of reducing the total energy of a system which is achieved by finding a new location for nodes in each iteration by solving partial differential equations.

Fruchterman and Reingold [20], on the other hand, mainly follow Eades' approach and make improvements on it. They apply repulsion forces between every pair of nodes and use a formula for spring forces that resembles Hooke's law. In addition, their algorithm tries to achieve an even distribution of the nodes on the drawing canvas. The equations used by Fruchterman and Reingold for repulsion and attractive forces are as follows respectively:

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

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

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

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

where $area$ is the area of drawing canvas, C is an experimentally found constant and $|V|$ is the number of nodes. They use a simulated annealing approach to reduce the energy of the system where the system starts with a high *temperature* and as the nodes move around in each iteration, the temperature of the system starts to cool down. After repeated iterations, the system reaches (or is forced to reach) a stable state and movement of the nodes finally stops.

Furthermore, Fruchterman and Reingold suggest a heuristic called *grid variant* to decrease the run time complexity of each iteration from $\mathcal{O}(|V|^2 + |E|)$ to $\mathcal{O}(|V| + |E|)$. This heuristic is based on the observation that repulsion forces produced between node pairs that are far away from each other have a negligible effect on the system. In the grid variant approach, the drawing area is divided

into a grid of squares and each node is placed in a grid square based on its position at each iteration. During the calculation of repulsive forces, only the nodes in the same square and neighboring squares of a node are considered. Since the algorithm tries to spread the nodes homogeneously by applying repulsion and attractive forces, in any iteration, the number of nodes in neighbor squares will be almost the same and relatively very small compared to the total number of nodes. Therefore, if the number of nodes in neighbor squares is considered to be constant, the complexity of the repulsive force calculation reduces to $\mathcal{O}(|V|)$ which makes the overall run time complexity $\mathcal{O}(|V| + |E|)$ per iteration.

2.2.1.1 Compound Spring Embedder (CoSE)

The CoSE algorithm [8] extends the force-directed model of Fruchterman and Reingold [20]. The main idea is similar in the sense that nodes are represented as electrically-charged particles that repel each other and edges are represented as springs that exert forces on the end nodes. CoSE adopts some important extensions on top of this force model to handle the layout of compound graphs.

CoSE represents a compound node as an "elastic cart" that can move freely in orthogonal directions, stretching its boundaries based on the movements of the nodes in its child graph (Figure 2.4). Unlike earlier force-directed models considering nodes as points or assuming that they have identical node dimensions and calculating distances between node pairs or the length of springs between two adjacent nodes as the distance between two node centers, CoSE uses the minimal Euclidean distance between the two nodes' boundaries in force calculations to both avoid node-node overlaps and support non-uniform node dimensions.

CoSE considers ideal edge lengths of intra-graph edges to be equal, while the ideal edge lengths of inter-graph edges that are used in spring force calculations are determined with a heuristic based on the depths of the edge's end nodes from their common ancestors in the inclusion tree.

In terms of force calculations, CoSE also adopts the grid variant heuristic of

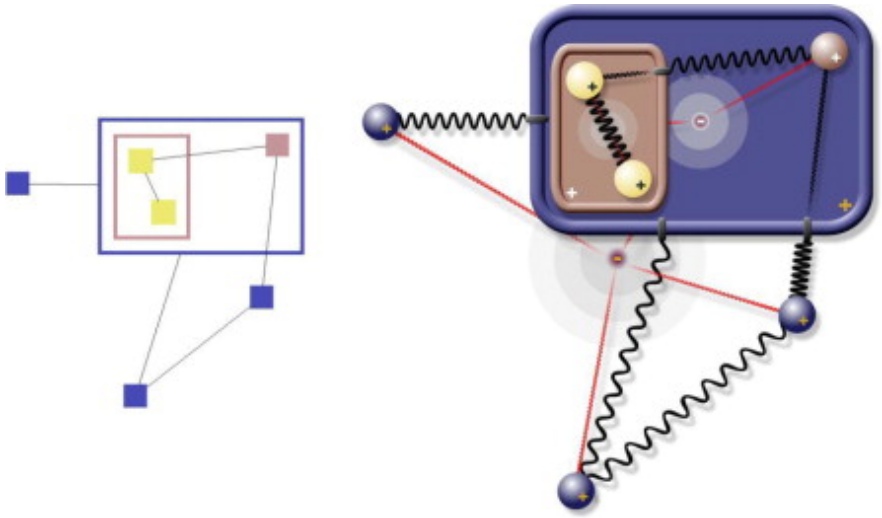


Figure 2.4: A sample compound graph (left) and its corresponding physical model used by CoSE (right) where the nodes are replaced with charged particles and the edges are replaced with springs while the compound nodes are like elastic carts that carry the nodes inside [8]

Fruchterman and Reingold to reduce its repulsion force calculation time. For the sake of simplicity and efficiency in the calculations, repulsion forces are calculated only between node pairs that are in the same graph. Additionally, CoSE introduces the gravitational forces, which are relatively weaker when compared to the repulsion and spring forces, to keep disconnected components in a graph together where such components are pulled towards the barycenter of the corresponding graph. Overall, CoSE has run time complexity of $\mathcal{O}(|V| + |E|)$ per iteration.

2.2.2 Spectral Layout Algorithms

The spectral layout approach is a class of graph layout algorithms that is based on the spectral decomposition (also called eigendecomposition) of graph-related matrices such as adjacency, Laplacian and graph-theoretical distance matrices [21]. The approach aims to utilize eigenvectors and eigenvalues computed from these matrices to find a positioning for the nodes.

These layout algorithms have two main advantages in terms of graph layout [22]. First, they provide analytical and exact solutions while almost all other formulations result in an NP-hard problem, which can only be solved with approximation. The second advantage is the computation speed of these approaches. Despite these advantages, spectral approaches are not commonly used in the graph layout, because they are devoid of aesthetic concerns. For example, these approaches cannot differentiate two nodes that have the same graph-theoretical distance to all other nodes and position them in the same location. Moreover, these algorithms consider nodes as points and do not support non-uniform node dimensions, which are widely used in real-life drawings and therefore yield node-node overlaps in that case.

In the scope of this thesis, we will take a close look at one of the well-known spectral approaches which is Classical Multidimensional Scaling (CMDS) [9].

2.2.2.1 Classical Multidimensional Scaling (CMDS)

The first CMDS algorithm was developed by Torgerson [23]; thus, it is also known as *Torgerson scaling*. The basic idea of CMDS is to find the coordinates that explain distances between objects [9]. More technically, given a distance matrix D of the graph, which is constructed from the graph-theoretical distances between all pairs of nodes, CMDS aims to find a mapping of the nodes in a d-dimensional space by preserving the graph-theoretical distances between nodes as much as possible. Assume X is the matrix of the final coordinates of the nodes in d dimensions. The scalar product matrix, $B = XX^T$, can be calculated by double-centering the squared distance matrix:

$$B = -\frac{1}{2}JLJ \tag{2.1}$$

where J is the centering matrix and $L = D^2$. Then, the coordinate matrix X can be obtained from the d largest positive eigenvalues λ_+ and their corresponding eigenvectors Q_+ of matrix B:

$$X = Q_+\sqrt{\lambda_+} \tag{2.2}$$

The calculation of λ_+ and Q_+ from B can be performed by a method called *power iteration* which is used to calculate dominant eigenvalues and eigenvectors of a matrix.

The computation of the distance matrix D and hence L takes $O(|V||E|)$ time by using a Breadth-First-Search (BFS). It takes $\Theta(|V|^2)$ time for the construction of B and another $O(|V|^2)$ time for the power iteration method which makes the overall run time of the operation quadratic. There exist a few studies [24][21][25] that aim to reduce this time complexity to linear time. Among them, Civril et al. [24] suggest a sampling-based approximation approach for the computation of L. Instead of running BFS from each node to compute the distances between every pair of nodes, they select a constant c number of *sample* nodes and run BFS only from those. By squaring each distance value computed, they obtain an $n \times c$ matrix C that contains the squared distances between each sample node and all other nodes. Then, they approximate the L matrix by multiplying three smaller matrices

$$L \approx C \Phi^+ C^T \tag{2.3}$$

where Φ is the $c \times c$ intersection matrix of C and C^T, and Φ^+ is the pseudo-inverse of Φ. With this approximation, they reduce the computation of the final coordinates X to linear time in the number of nodes and edges.

2.3 Layout Constraints

Layout constraints are generally divided into two categories, *soft* and *hard* constraints [26]. Soft constraints are also known as *optimization* constraints and they aim to optimize metrics from generally accepted aesthetic criteria such as the number of node-node overlaps and average edge length. Optimization of each such metric is an NP-hard problem [14]; hence it is not expected for layout algorithms to satisfy these constraints, instead they use some heuristics to keep these metrics in acceptable values. Soft constraints are generally used while evaluating the visual quality of a layout.

Hard constraints, on the other hand, are given as user input and expected to be satisfied by the layout algorithms. These constraints include but are not limited to placement constraints defined on nodes such as assigning a node to a fixed position or requiring alignment or relative placement of a group of nodes, styling constraints such as drawing edges only in polylines or in orthogonal form and port constraints such as restricting edges to be connected to certain ports.

In the scope of this thesis, we use the number of node-node overlaps, edge-edge crossings and node-edge crossings, average edge length and total area as the soft constraints to evaluate the quality of the layout algorithms. We also focus on three hard constraints related to node placement; fixed node, alignment and relative placement constraints whose details are given in Section 4. We will call hard constraints simply constraints in the rest of this thesis.

2.4 Orthogonal Procrustes Problem

The Procrustes problem is a matrix approximation problem in linear algebra that aims to find a transformation matrix to map a source configuration of a set of points into a target configuration as closely as possible. Unsurprisingly, it gets its name from a bandit in Greek mythology who tries to fit his victims into a bed either scratching their limbs or cutting them off. In the orthogonal version, transformation is restricted to only rotations and reflections (Chapter 20 of [9]).

The orthogonal Procrustes problem can be solved as follows. Let A and B be $n \times 2$ matrices keeping the centralized coordinates (in x and y axes) in the target and source configuration of the n points, respectively. Also, let $P\Sigma Q^T$ be the singular value decomposition (SVD) of $A^T B$. Then, the orthogonal transformation matrix that maps source configuration into the target one can be calculated with $T = QP^T$.

For example, in Figure 2.5 we see two rectangles A and B with the following

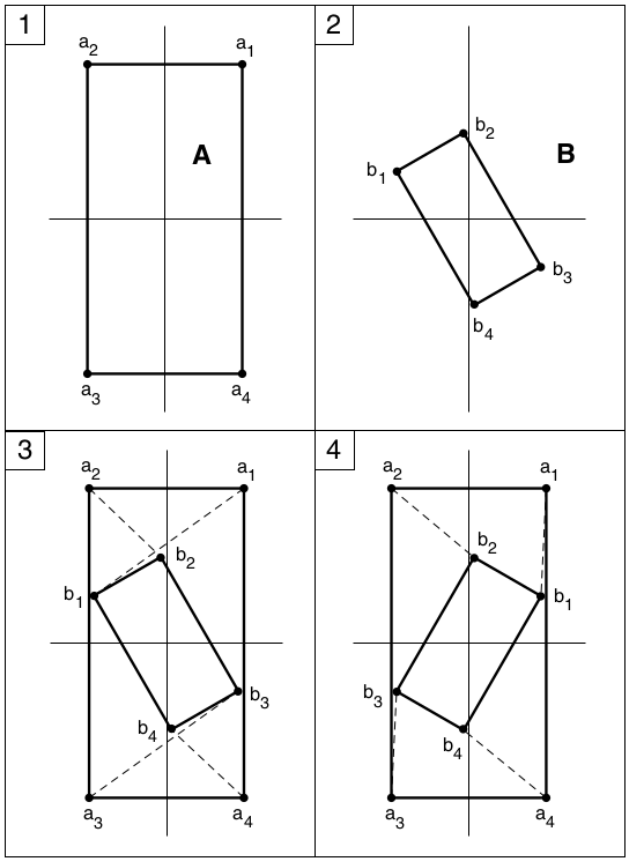


Figure 2.5: Some steps used by an orthogonal transformation to fit rectangle B to rectangle A [9]

configurations:

$$A = \begin{bmatrix} 1 & 2 \\ -1 & 2 \\ -1 & -2 \\ 1 & -2 \end{bmatrix} \quad B = \begin{bmatrix} -1.866 & 1.232 \\ -0.134 & 2.232 \\ 1.866 & 1.232 \\ 0.134 & -2.232 \end{bmatrix}$$

To map rectangle B into rectangle A as closely as possible by using only rotations and reflections, we first calculate $A^T B$.

$$A^T B = \begin{bmatrix} 1 & -1 & -1 & 1 \\ 2 & 2 & -2 & -2 \end{bmatrix} \begin{bmatrix} -1.866 & 1.232 \\ -0.134 & 2.232 \\ 1.866 & 1.232 \\ 0.134 & -2.232 \end{bmatrix} = \begin{bmatrix} -3.464 & -2 \\ -8 & 13.856 \end{bmatrix}$$

Then SVD of A^TB is

$$P\Sigma Q^T = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 4 & 0 \\ 0 & 16 \end{bmatrix} \begin{bmatrix} -0.866 & -0.5 \\ -0.5 & 0.866 \end{bmatrix}$$

Finally, we can obtain T by

$$T = QP^T = \begin{bmatrix} -0.866 & -0.5 \\ -0.5 & 0.866 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} -0.866 & -0.5 \\ -0.5 & 0.866 \end{bmatrix}$$

T is the transformation matrix that maps B into A, but what T does to B? From Figure 2.5, we can see that it first reflects B on the vertical axis and then rotates it 30° counterclockwise. We can also see that result by multiplying corresponding reflection and rotation matrices:

$$T = UR = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 0.866 & 0.5 \\ -0.5 & 0.866 \end{bmatrix} = \begin{bmatrix} -0.866 & -0.5 \\ -0.5 & 0.866 \end{bmatrix}$$

where U is the reflection matrix for a reflection on y-axis and R is the rotation matrix for a rotation by 30° counterclockwise.

Chapter 3

Related Work

Force-directed layout approaches are intuitive and easy to understand and program [7]. Therefore, a lot of studies have been done in this field since the studies of Eades [18], Kamada and Kawai [19] and Fruchterman and Reingold [20], whose details are given in Section 2.2.1, both to improve their performance and to adopt and use them for field-specific purposes. Some methods used in performance improvement include but are not limited to CPU/GPU acceleration/parallelization [27][28][29], which mainly aims for the force calculations to be done in parallel, and applying multilevel scaling approach [30][31][32] where the graph is first recursively coarsened by grouping its nodes into clusters until obtaining a trivial one and then the coarsest graph is extended again by optimizing the layout each time until ending with the original graph. Force-directed layouts are also adopted to be used in many applications such as visualizing clustered graphs [33] and biological pathways [34]. More studies on force-directed layout and their details can be found in Chapter 12 of [35] and a recent survey [36].

On the other hand, the number of studies on the layout of compound graphs is limited probably because of their distinctive and relatively complex structures, which require handling various levels of nesting, inter-graph edges and so on. These studies generally suffer from varying weaknesses in quality together with poor run time complexity. Sugiyama and Misue [37], Sander [38] and Eades et

al. [39] study the layout of directed hierarchical graphs with compound structures, in which the hierarchy between nodes is enforced via edge directions. These studies, however, do not perform well when applied to undirected graphs. Bertault and Miller [40] and Didimo and Montecchiani [41] aim specifically for undirected graphs and use a top-down or bottom-up approach on the inclusion tree of compound graphs. The drawback of these approaches is that they do not handle inter-graph edges properly, yielding long and poorly routed such edges. The rest of the studies on compound graph layout has some weaknesses such as supporting only one level of nesting [42][43] and not allowing edges to be connected directly to compound nodes [44], significantly limiting the type of compound graphs.

The CoSE algorithm [8] whose details are given in Section 2.2.1.1, however, provides full support for compound graphs by handling compound structures properly, allowing an arbitrary number of nesting levels and avoiding node-node overlaps etc. However, it offers a mediocre performance in terms of speed with a run time complexity $\mathcal{O}(k \cdot (|V| + |E|))$ by using the grid variant method in [20], where k is the number of iterations estimated to be $\mathcal{O}(|V|)$, which can be considered slow for especially medium size graphs that are commonly used in interactive visualization tools.

There has been also limited work done on spectral layout algorithms because of their limitation on producing aesthetically pleasing layouts for real-life graphs. Even though they offer a fast approach to graph layout, they cannot handle non-uniform node dimensions which yields node-node overlaps and makes them not applicable for compound graphs. Recall that the spectral methods construct the layout using the eigenvectors and eigenvalues of the certain matrices associated with the graph. One of the first studies on the spectral layout is by Hall [45] who uses eigenvectors of the Laplacian matrix. Similarly, Koren et al. [46] and Koren [22] offer algorithms that use eigendecomposition of the Laplacian matrix. Harel and Koren [47] first lay out the graph in a high-dimensional environment by using graph-theoretical distances and then project the high-dimensional drawing into 2-dimensions by applying principal component analysis (PCA). Besides these, one of the common techniques used in spectral graph drawing is Classical Multidimensional Scaling (CMDS) whose details are given in Section 2.2.2.1.

While the first algorithm proposed by Torgerson [23] has a quadratic time complexity, [24], [21] and [25] propose approximations that run in linear time. From those, we use [24] by Civril et al. in our study because it runs slightly faster and produces more quality layouts according to our observations.

While many layout algorithms including CoSE aim to achieve good layouts in terms of one or more of the soft constraints by using heuristic approaches, there are also studies that explicitly focus on the optimization of such constraints [48][49][50] by either modifying the force model or defining cost functions in the stress majorization method based on such constraints. On the other hand, the first attempt on satisfying hard constraints (simply *constraints*) was by Böhringer and Paulisch [51] who try to enforce some placement constraints on nodes by modifying the layered drawing algorithm of Sugiyama et al. [52]. He and Marriot [53] achieve separation constraints by extending Kamada and Kawai stress model [19] and using an active-set method to solve constrained optimization problems; however, their algorithm scale only to small graphs due to inefficient solvers. Studies by Ryall et al. [54], Wang & Miyamoto [42] and Didimo et al. [55] use varying heuristics such as adding dummy nodes and defining extra springs between nodes to modify and manipulate force-directed model for constraint support.

CoLa (Constraint-based Layout) is the most well-known constrained layout algorithm in the literature which is a cumulative result of a few studies [56][57][58]. It uses an iterative gradient-projection algorithm that utilizes either stress majorization or force-directed model. In each iteration, each node is first positioned based on a calculated steepest descent vector and then the nodes with constraints are projected to the appropriate positions that satisfy constraints. Both movement operations aim to decrease either stress or the energy of the system. CoLa is able to support a wide range of constraints at the cost of quadratic run time complexity. Moreover, CoLa utilizes constraints to support compound graphs and avoid node-node overlaps. They achieve this by defining relative placement constraints between all node pairs and between the boundaries of the compound nodes and simple nodes which yields in a quadratic number of constraints. This

approach further increases its already high run time complexity and makes it unsuitable to be applied to medium-sized graphs. A recent study by Wang et al. [48], on the other hand, proposes an approach that improves the stress majorization method by reformulating its stress function to support more constraint types (including some soft constraints) than CoLa supports. They present a GPU-based implementation for their algorithm that outperforms CoLa while their CPU-based implementation generates a comparable performance.

All in all, when we evaluated the studies done so far on compound graphs and constrained layout, we can infer that there is no layout algorithm that both runs fast enough to be applied on small to medium-sized graphs and supports non-uniform node dimensions, compound structures and placement constraints commonly used in real-life applications. Here, we present such an algorithm named fCoSE to fill this void.

Chapter 4

Algorithm

In this chapter, we present our fCoSE layout algorithm in detail by first introducing the constraint types it supports, then explaining the main phases of the algorithm in depth and lastly discussing its run time complexity.

fCoSE supports three types of layout constraints that are commonly used in real-life graphs:

- **Fixed node constraint:** In this constraint type, the user may provide desired (aka *anchor*) positions for a set of nodes, which we call *fixed* nodes, as input to the algorithm. This is especially useful if the user has previously appointed positions for some nodes but wants other nodes to be positioned by the layout algorithm appropriately and freely. We denote a node a with a fixed node constraint at anchor position (x, y) as "$a\dagger[x, y]$" and the set of fixed node constraints as C^f. The algorithm is expected to produce a layout with the fixed node a positioned *exactly* at (x, y).
- **Alignment constraint:** The user sometimes may want to align a set of nodes vertically or horizontally for practical reasons such as to show that they are on the same level or to keep them in an order to better represent a sequence. Hence, the algorithm aims to align the centers of two or more nodes vertically or horizontally with this constraint type. We denote the

vertical alignment of nodes a, b, c as "$a \mid b \mid c$", horizontal alignment of them as "$a - b - c$" and the set of alignment constraints as C^a. The user is allowed to define more than one constraint in each direction. Moreover, a node can be a part of two constraints defined in different directions, but not in the same direction. In such cases where a node is part of two constraints defined in the same direction, those constraints should be combined into one constraint (e.g., "$a-b-c$" as opposed to "$a-b$" and "$b-c$" as two separate constraints). We should note that both of these relations are transitive and reflexive.

- **Relative placement constraint:** This constraint type allows the user to position a node relative to another one by a certain minimum distance between them in either vertical or horizontal direction. This constraint type is useful to form an ordering between nodes or in cases where the relative positioning of nodes makes sense. If node a will be positioned to the left of (above) node b by at least $x > 0$ units, we denote it as "$a <[x]\ b$" ("$a\ \wedge[x]\ b$"). If x value is not specified, then the algorithm assumes a minimum separation amount between nodes. We also denote the set of relative placement constraints as C^r. A node can get involved in multiple relative placement constraints in either direction. We should note that both of these relations are transitive and irreflexive.

We should also note that these constraints can be specified only on simple nodes and each simple node can be part of multiple constraint types. The algorithm also expects that the user does not specify conflicting constraints such as $a < b$ and $b < a$.

The fCoSE algorithm running on a compound graph $G = (V, E, F)$ with a user-specified constraint set $C = C^f \cup C^a \cup C^r$ consists of three main phases (Figure 4.1 and 4.2). In the first phase, a *draft* layout is obtained by first converting a possibly disconnected compound graph into a connected simple graph and then applying a spectral layout algorithm [24] on it.

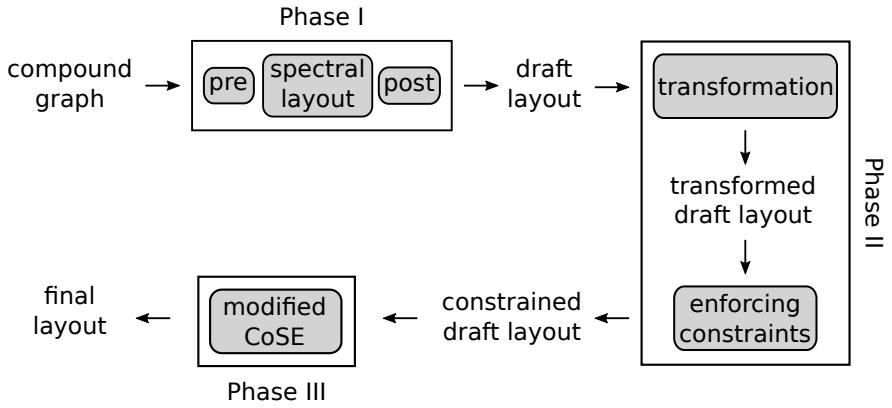


Figure 4.1: Algorithm overview of fCoSE. In Phase I, a spectral layout algorithm is applied on the input compound graph to generate a draft layout. Phase II enforces the user-specified placement constraints on this draft layout after a transformation step. Finally, a final layout is obtained as a result of Phase III by polishing the constrained draft layout via a modified CoSE algorithm.

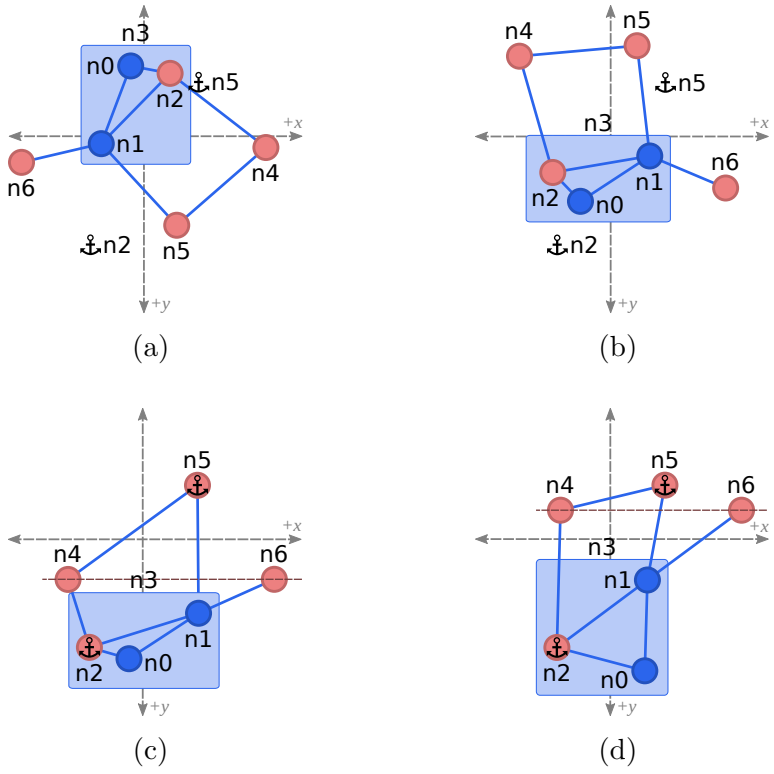


Figure 4.2: The process of applying fCoSE on a sample compound graph with two fixed node constraints ($n2\dagger[-50, 100]$, $n5\dagger[50, -50]$) and one alignment constraint ($n4 - n6$). (a) After generating a draft layout in Phase I, (b) a transformation is first applied based on the fixed node constraints (transformed draft layout) and then (c) all constraints are enforced (constrained draft layout) in Phase II. (d) The final layout is obtained by applying a modified version of CoSE in Phase III.

Algorithm 1 The fCoSE Algorithm

function RunLayout(G, C^f, C^a, C^r)
  ApplySpectral(G)
  if $|C^f| > 1$ **then** ▷ use fixed nodes
    $xformMatrix \leftarrow$ CalcXformFixed(G, C^f)
    ApplyXform($G, xformMatrix$)
  else if $|C^f| \leq 1$ **and** $|C^a| > 0$ **then** ▷ use alignment
    $xformMatrix \leftarrow$ CalcXformAlignment(G, C^a)
    ApplyXform($G, xformMatrix$)
    if $|C^r| > 0$ **then**
      ApplyMajorityReflection(G, C^r)
  else if $|C^f| \leq 1$ **and** $|C^a| = 0$ **and** $|C^r| > 0$ **then** ▷ use relative placement
    construct dags D^h and D^v from C^r
    $D \leftarrow D^h \cup D^v$
    $C_i = (V_i, E_i) \leftarrow$ largest component in D
    if $|V_i| < |V(D)/2|$ **then**
      ApplyMajorityReflection(G, C^r)
    else
      $xformMatrix \leftarrow$ CalcXformRelative(G, C_i)
      ApplyXform($G, xformMatrix$)
  if $|C^f| > 0$ **then**
    EnforceConstraintsFixed(G, C^f)
  if $|C^a| > 0$ **then**
    EnforceConstraintsAlignment(G, C^f, C^a)
  if $|C^r| > 0$ **then**
    EnforceConstraintsRelative(G, C^f, C^a, C^r)
  $totIter \leftarrow 0$
  while $totIter < maxIter$ **or** !Converged() **do**
    $totIter \leftarrow totIter + 1$
    UpdateBounds() ▷ resize compounds
    CalcForces()
    CalcDisplacements()
    if $|C^f \cup C^a \cup C^r| > 0$ **then**
      AdjustDisplacements()
    MoveNodes()

The second phase first transforms the draft layout so that its orientation better complies with the user-specified constraint set (*transformed* draft layout) and then enforces the constraints in order to obtain a *constrained* draft layout. In the last phase of the algorithm, a modified version of CoSE [8] is applied to polish the constrained draft layout by taking node dimensions into account and eliminating the node-node overlaps disregarded so far while maintaining the enforced constraints. As a result, the algorithm achieves a *final* layout where compound structures are respected and user-specified constraints are satisfied. Algorithm 1 summarizes the overall structure of the algorithm.

4.1 Phase I: Obtaining Draft Layout

Spectral layout algorithms are based on the adjacency or the graph-theoretical distance of the nodes to each other which prevents them to be applied directly to disconnected or compound graphs. To solve this problem, we propose the following heuristic. Assuming we have a disconnected compound graph as the input, we first apply a preprocessing step to connect disconnected components in the graph together and convert it into a simple graph. After performing the spectral layout algorithm on the connected simple graph, we finally convert the graph into a disconnected compound graph again via a postprocessing step and obtain a draft layout.

4.1.1 Preprocessing Step

We convert a disconnected graph into a connected one by using "dummy nodes" that tie its components together. To achieve this, we first perform a Breadth-First-Search (BFS) algorithm in the root graph to identify the disconnected components and connect them to a dummy node. Two points to note here are: First, the BFS algorithm we use is a specialized one that considers the compound node structure where upon reaching a particular node, if it is a compound node, we also visit all nodes in its nested child graph. Similarly, if it is a child node, we also

visit all its parent/sibling nodes. For example, in Figure 4.3a, although the compound node $c1$ is not directly adjacent with $n2$ or compound node $c0$, we consider them as connected since a child node $n3$ of $c1$ is adjacent with $n2$ and $c1$ itself is adjacent with a child node $n1$ of $c0$. Therefore, upon reaching $c1$, we should also be visiting $n2$ and $c0$ on our traversal together with some other nodes. Secondly, we select the node with the *minimum degree* from each component to connect it to a dedicated dummy node to keep the node degrees as homogeneous as possible (Figure 4.3b). After we handle the root graph by considering these points, we need to apply the same procedure for each child graph inside the compound nodes because the graph might become disconnected once we remove compound structures as described below (Figure 4.3c).

We convert a compound graph into a simple one by assigning the mission of each compound node to a simple node inside that compound node. For this purpose, we first select a simple node with minimum degree (again to keep node degrees homogeneous after conversion) inside a compound node, connect the incident edges of the compound node to the selected node and remove the compound node *temporarily* (Figure 4.3d). As a result of this process, we obtain a connected simple graph which is ready to apply a spectral layout algorithm on.

4.1.2 Applying Spectral Layout Algorithm

Having a connected and simple graph, we now apply a linear time CMDS algorithm proposed by Civril et al. [24] whose details are explained in Section 2.2.2.1. As noted before, as a spectral layout approach, this algorithm does not consider non-uniform node dimensions and places two nodes with the same graph-theoretical distance to all other nodes on top of each other potentially leading to many node-node overlaps. However, it helps to generate a quick draft layout that nicely represents the skeleton of the graph.

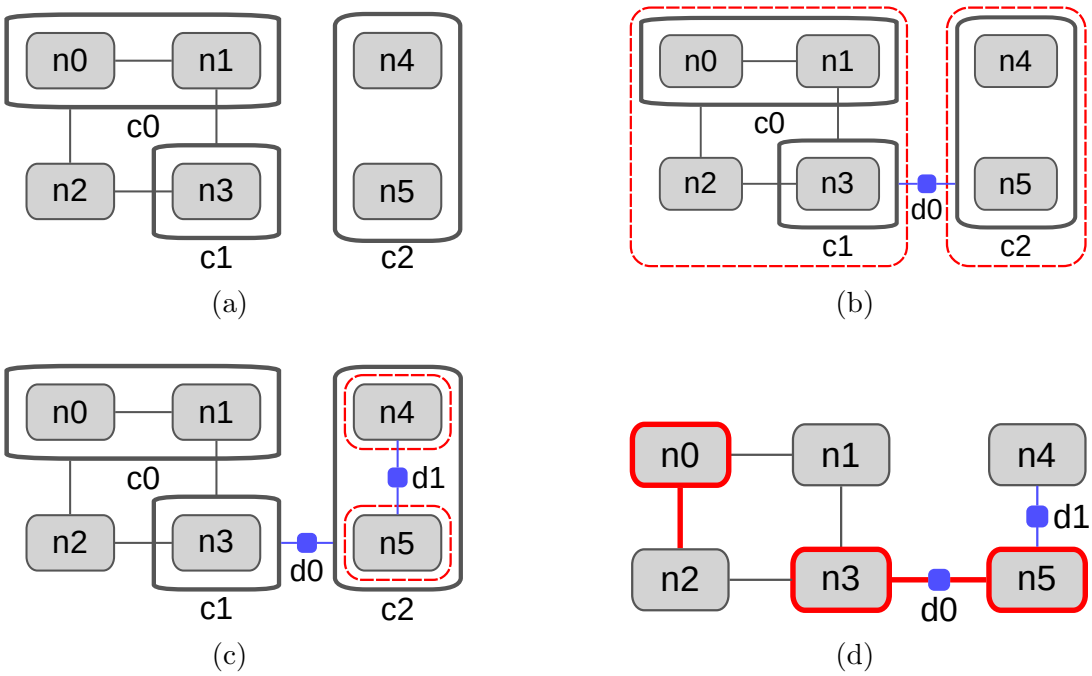


Figure 4.3: (a) Given a sample disconnected compound graph, (b) the preprocessing step first connects the disconnected components shown inside the red rounded rectangles in the root graph via dummy node $d0$. (c) Dummy node $d1$ then connects the components inside the child graph of $c2$. (d) As there are no remaining components to connect, the compound graph is finally converted to a simple graph by selecting a representative node for each compound node. Nodes $n0$, $n3$ and $n5$ with red borders are the representative simple nodes selected for compound nodes $c0$, $c1$ and $c2$, respectively and the edges in red indicate the edges previously connected to compound nodes.

4.1.3 Postprocessing Step

In the postprocessing step, we finalize the construction of a draft layout simply by bringing the temporarily removed compound nodes back by computing their positions and dimensions using the bounding boxes of their newly positioned child nodes and removing the dummy nodes that we inserted to connect disconnected components.

4.2 Phase II: Satisfying Constraints

In this phase, we satisfy the user-specified placement constraints on top of the draft layout that is the result of Phase I. To achieve this, we first apply a transformation on the draft layout by performing rotation and/or reflection to better align it with a subset of constrained nodes and obtain a transformed draft layout. We then process and enforce the constraints on the transformed draft layout in order, yielding a constrained draft layout where all constraints are satisfied.

4.2.1 Transformation of Draft Layout

The aim of the transformation step is to adjust the orientation of the graph in such a way that it becomes more compatible with the user-specified placement constraints. In this way, the movement of the constrained nodes during the next step where we process the constraints to enforce them is minimized and the structure of the draft layout is protected as much as possible. If we directly enforce the constraints to the draft layout, this may cause drastic changes in the node positions and emergence of the long edges and eventually reduce the quality of the constrained draft layout and hence the quality of the final layout. To provide a better insight, Figure 4.4 exemplifies the transformation step. Notice how directly enforcing fixed nodes may cause long edges (Figure 4.4b), whereas, we obtain a better quality constrained draft layout with the help of the transformation step (Figure 4.4c).

We achieve this transformation by using the solution to the famous orthogonal Procrustes problem, whose details are given in Section 2.4, that aims to map a source configuration to a target one by calculating an orthogonal transformation matrix. This solution exactly serves our purpose by restricting the transformation to only rotations and reflections, because we only want to change the orientation of the draft layout without distorting it.

To apply this solution in our case, we first decide the nodes that will be used

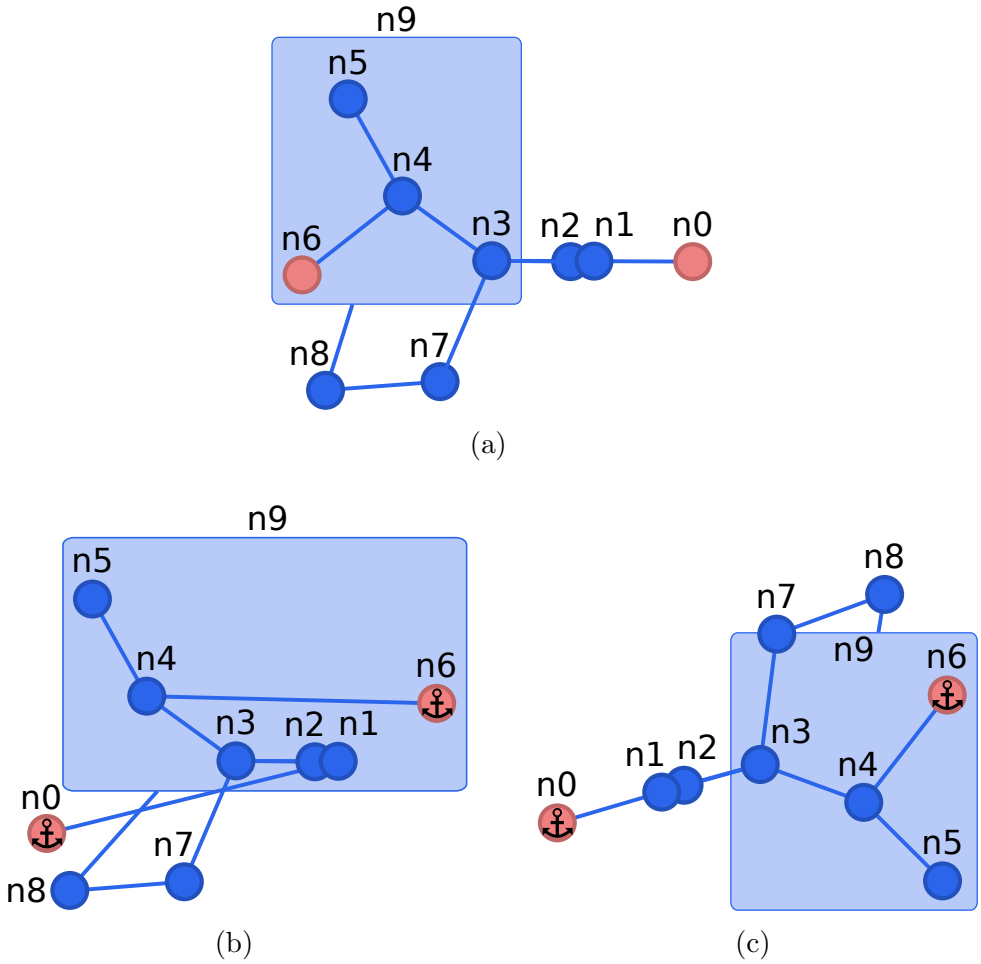
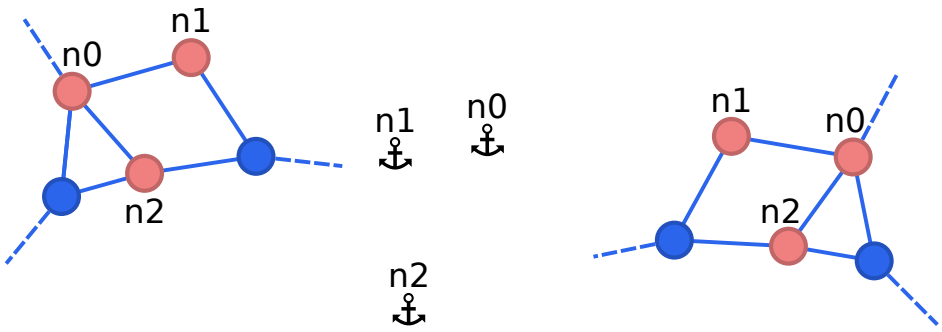


Figure 4.4: (a) A sample draft layout with two fixed node constraints $n0\dagger[-150, 50]$ and $n6\dagger[150, -50]$ (b) Constrained draft layout obtained by skipping the transformation step and (c) by applying a transformation step that rotates the draft layout by 163.5° clockwise

in the calculation of the transformation matrix together with their source and target configurations, and then we apply the transformation matrix to the whole graph. Because we aim to adjust the orientation of the draft layout according to the user-specified constraints, we can select a proper subset of the constrained nodes for this purpose. The source configuration of the selected nodes comes from the draft layout, while the construction of the target configuration is rather complicated and changes according to the chosen constraint type(s). The choice of the constraint type(s) and the actual set of nodes, together with the construction of the target configuration are done as follows.



(a) (b)

Figure 4.5: (a) Part of a draft layout along with the target configuration formed by anchor positions on the fixed nodes $n0$, $n1$ and $n2$ (b) The corresponding transformed draft layout which first reflects the draft layout on the y-axis and then rotates it by 6° counterclockwise

If there is more than one fixed node constraint ($|C^f| > 1$), we select the fixed nodes for the transformation, as the fixed node constraints are the most strict ones in terms of positioning. The target configuration is then formed directly from the positions specified by the user for these nodes (anchor positions). Figure 4.5 illustrates a sample scenario with three fixed nodes. The positions of these nodes in the draft layout form the source configuration, while the anchors show the anchor positions that form the target configuration (Figure 4.5a). Please see that the transformed draft layout obtained by applying the transformation matrix formed from these configurations is more compatible with the anchor positions (Figure 4.5b).

If the number of fixed node constraints is insufficient to define a target configuration ($|C^f| \leq 1$), and there exist any alignment constraints ($|C^a| > 0$), then we use all nodes involved in this constraint type for the transformation. In this case, the target configuration is formed by aligning the nodes in each alignment constraint on their average position in the respective direction. Figure 4.6a shows a draft layout with two alignment constraints (one in each direction). In Figure 4.6b, nodes in each alignment constraint are moved to their average positions to form the target configuration and the resulting transformed draft layout is shown in Figure 4.6c. Here, we can also see how the transformation reduces the total amount of node movement required to enforce the alignment constraints in

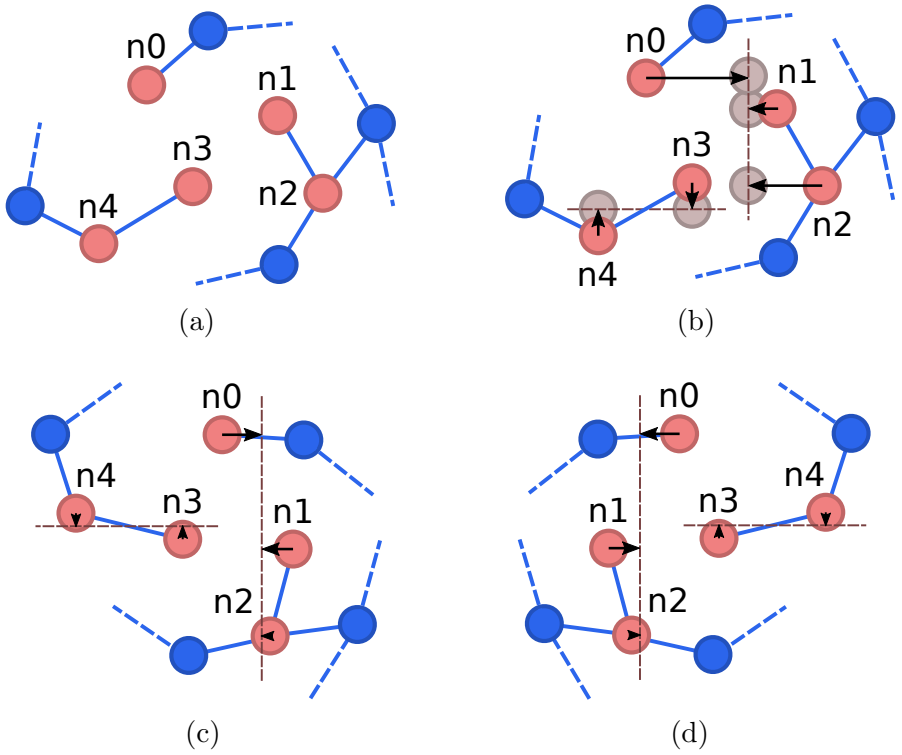


Figure 4.6: (a) Part of a draft layout with alignment constraints $n0 \mid n1 \mid n2$ and $n3 - n4$ (b) Black arrows show the required movement for the constrained nodes to be aligned in the corresponding average coordinate to form the target configuration. (c) The transformation step based on the alignment constraints rotates the draft layout by 44.5° clockwise. (d) Additional process of relative placement constraints $n1 < n3$, $n2 < n4$, $n0 \wedge n1$ and $n2 \wedge n4$ on the transformed draft layout in (c) reflects the graph on y-axis.

the next step by simply comparing the total lengths of dark line segments with arrows in Figure 4.6b (if we enforce without transformation) and Figure 4.6c (with transformation).

At this point, after we transformed the draft layout based on the alignment constraints, we check whether we can make it more compatible with the relative placement constraints as well. If there are any relative placement constraints ($|C^r| > 0$), we additionally apply a majority-based reflection on the graph by taking these constraints into account, if required, as follows. We evaluate the relative placement constraints defined along the x-axis (y-axis) one by one, and if the two involved nodes are not on the correct side of each other as required by the

constraint, we consider that this constraint is violated. After the evaluation of all constraints defined along the x-axis (y-axis), if the majority of these constraints are violated, we reflect the graph on y-axis (x-axis). To illustrate this with an example, assume that the graph in Figure 4.6a has also the following relative placement constraints: $\{n1 < n3,\ n2 < n4,\ n0 \wedge n1,\ n2 \wedge n4\}$. If we process each constraint based on the positions in the current transformed draft layout (Figure 4.6c) as explained, we see that both constraints defined along the x-axis are violated, while one of the two constraints is violated on y-axis. In this case, we reflect the graph only on the y-axis (Figure 4.6d). With this additional action, the orientation of the graph becomes more compatible with the relative placement constraints, reducing the number of violated constraints to one as opposed to three, while the effect of the earlier transformation based on the alignment constraints is also protected as this action involves only reflections.

If the number of both fixed node and alignment constraints is not sufficient to apply a transformation ($|C^f| \leq 1$ and $|C^a| = 0$), we base the transformation only on the relative placement constraints, if they exist ($|C^r| > 0$). To achieve this, we first form two dependency dags $D^h = (V^h, E^h)$ and $D^v = (V^v, E^v)$, one for each direction, by using the relative placement constraints. Here,

$$V^h = \{v \mid (u < v) \in C^r \vee (v < u) \in C^r\} \text{ and}$$

$$E^h = \{e = (u, v) \ \wedge \ e.w = x \mid (u <_{[x]} v) \in C^r\},$$

where $e.w$ denotes the weight of the edge e. D^v can be formed in the same way. Please notice here that $E^h \cap E^v = \emptyset$ but $V^h \cap V^v$ can be nonempty. The union of these two dags then constructs the directed dependency graph $D = D^h \cup D^v$, which may not be a dag itself.

We then have two choices for the transformation. Assume that $C_i = (V_i, E_i)$ is the largest (weakly connected) component of D. If $|V_i| < |V(D)|/2$; in other words, the largest component is not big enough to direct a transformation, we apply a majority-based reflection on the draft layout as explained before. Otherwise, we use the nodes in the largest component C_i to configure the target configuration as follows. Suppose $V_i = V_i^h \cup V_i^v$, where $V_i^h \subseteq D^h$ and $V_i^v \subseteq D^v$ correspond to those nodes involved in horizontal and vertical relative placement

constraints in the component, respectively. Let $D_i^h = D^h[V_i^h]$ and $D_i^v = D^v[V_i^v]$, both of which are dags as they are subgraphs of dags. For each of these dags, we first define the nodes with no incoming edges as the source nodes. We then calculate the longest distances from these source nodes to all other nodes using a topological order based computation [59]. We finally obtain the target configuration by first aligning the source nodes in their average x (y) coordinate and then placing the rest of the nodes in the V_i^h (V_i^v) to the appropriate x (y) coordinates by using the previously calculated longest distance of each node from the source nodes in the x-axis (y-axis). Figure 4.7 shows an example of this approach where the largest component is used to obtain a target configuration for the transformation. We can see that the transformation changes the orientation of the graph in a way that all constrained nodes are on the correct side of each other as required by the constraints (Figure 4.7d), whereas none of them are on the correct side initially (Figure 4.7a).

4.2.2 Enforcing Constraints

After we adjust the orientation of the graph based on the user-specified placement constraints, we now enforce these constraints on the transformed draft layout in the following order: fixed node, alignment and relative placement. As a result, we obtain a constrained draft layout in which all constraints are satisfied and ready to be processed by Phase III.

4.2.2.1 Fixed Node Constraints

The positions of the fixed nodes in the transformed draft layout are highly likely to be different than the user-specified anchor positions. Therefore, we enforce the fixed node constraints by simply moving each fixed node to the corresponding anchor position. However, this movement may affect the overall structure of the graph drastically by causing long edges or leaving a part of the graph away. To avoid this, we also need to move the rest of the graph towards the anchor positions

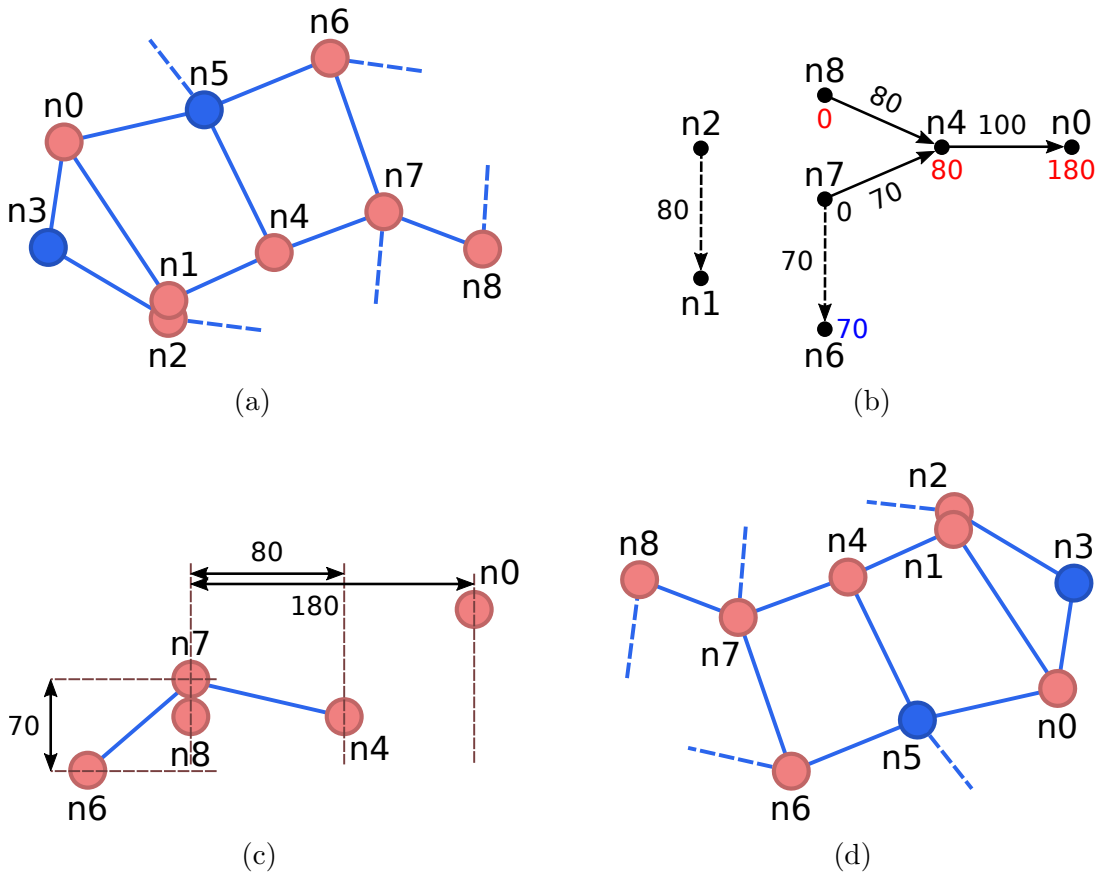


Figure 4.7: (a) Part of a draft layout for a graph with relative placement constraints $\{n2 \wedge[80]\ n1,\ n7 \wedge[70]\ n6,\ n4 <[100]\ n0,\ n7 <[70]\ n4,\ n8 <[80]\ n4\}$ (b) Dependency graph D formed by the nodes involved in these constraints. The dashed edges represent the constraints defined in the vertical direction while the solid ones are for those in the horizontal direction. The value on the edge represents its weight and the value near each node represents its longest distance from the source nodes. The component on the right is large enough to be used to construct the target configuration. (c) The target configuration constructed by appropriately positioning the nodes in the largest component according to the calculated longest distances (d) Transformed draft layout obtained by a transformation that rotates the draft layout by 180°)

by taking the displacement amount of the fixed nodes into account. We apply this displacement process as follows.

Let $V^f = \{v \mid v\dagger(x, y) \in C^f\} = \{v_0, v_1, \ldots, v_k\}$. Also let $(x_i,\ y_i)$ be the coordinates of the fixed node v_i in the transformed draft layout and (x'_i, y'_i) be the coordinates of the fixed node v_i required by the fixed node constraint (Figure 4.8a). Each fixed node v_i is moved by $(x'_i - x_i)$ and $(y'_i - y_i)$ along the x and y axes, respectively, to enforce the constraint (Figure 4.8b). The average displacement amounts of fixed nodes are calculated in each direction to decide the displacement amounts, δx and δy, for the rest of the graph:

$$\delta x = \frac{\sum_{i=0}^{k}(x'_i - x_i)}{k} \quad \text{and} \quad \delta y = \frac{\sum_{i=0}^{k}(y'_i - y_i)}{k}.$$

The rest of the graph is then moved by $(\delta x, \delta y)$ along x and y axes, respectively (Figure 4.8c).

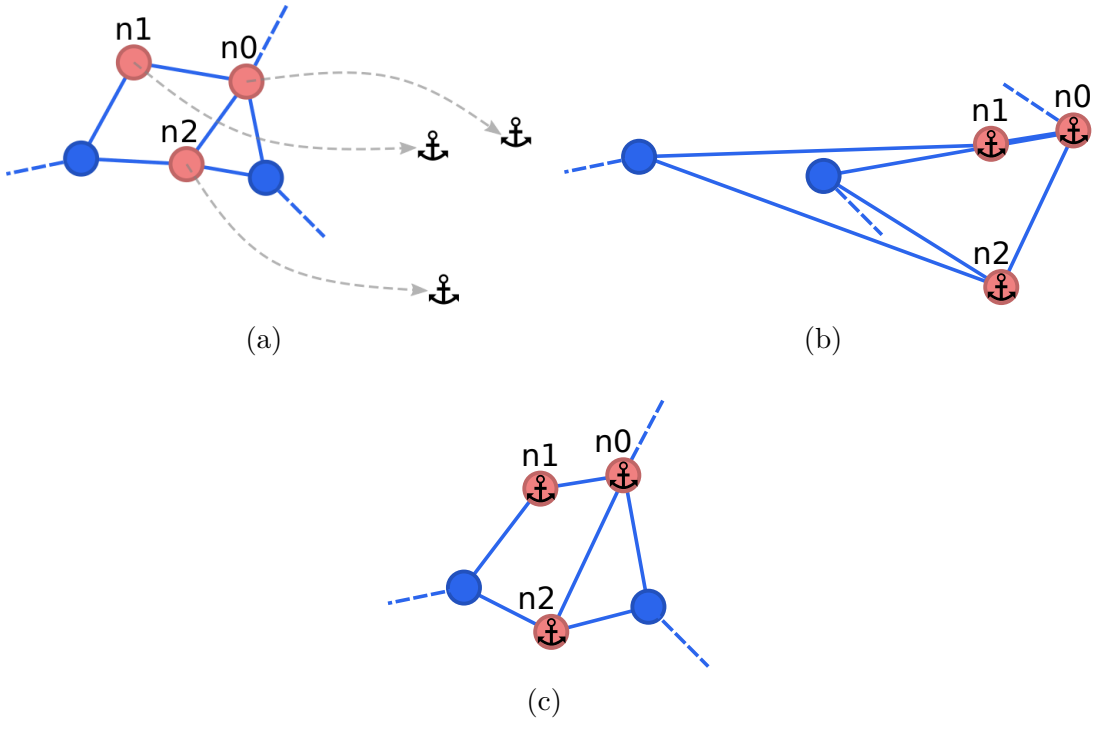
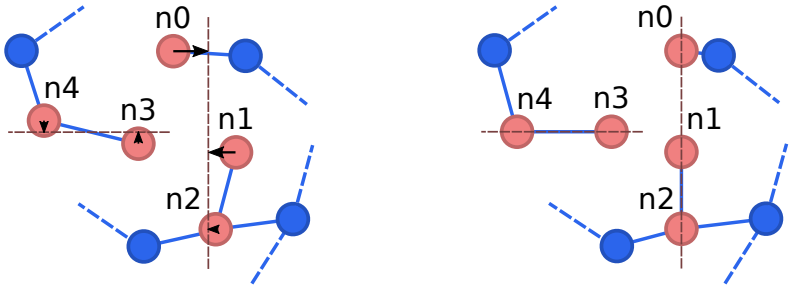


Figure 4.8: (a) Transformed draft layout (same as the one in Figure 4.5b) along with the anchor positions of fixed nodes (b) First, the fixed nodes are moved to anchor positions. (c) Then, the rest of the graph is moved based on the displacement of the fixed nodes.

4.2.2.2 Alignment Constraints

We enforce an alignment constraint by simply aligning the nodes involved in the constraint to their average x (y) coordinate for vertical (horizontal) alignment (Figure 4.9). However, if a node in the alignment constraint also has a fixed node constraint, then other nodes in the constraint should comply with the fixed node. Hence, we use the corresponding coordinate of the fixed node as the alignment coordinate. We should, however, note that if there is more than one fixed node in an alignment constraint, then their corresponding coordinates that will be used for the alignment should be the same; otherwise, the constraints will conflict with each other.



(a) (b)

Figure 4.9: (a) Transformed draft layout with two alignment constraints (same as the one in Figure 4.6c) where the arrows show the actions required to enforce these constraints (b) Constrained draft layout in which the alignment constraints are satisfied

4.2.2.3 Relative Placement Constraints

To enforce the relative placement constraints, we again construct dependency dags, D^h and D^v, as in the transformation step. However, this time the construction and processing of these dags are a bit more sophisticated because we also need to consider possible fixed node and alignment constraints. For example, during the construction of D^h (D^v) we represent the nodes involved in each vertical (horizontal) alignment constraint with a *meta* node (i.e., a single merged node). Moreover, if any of these nodes has a fixed node constraint, then we also consider the meta node as a fixed node.

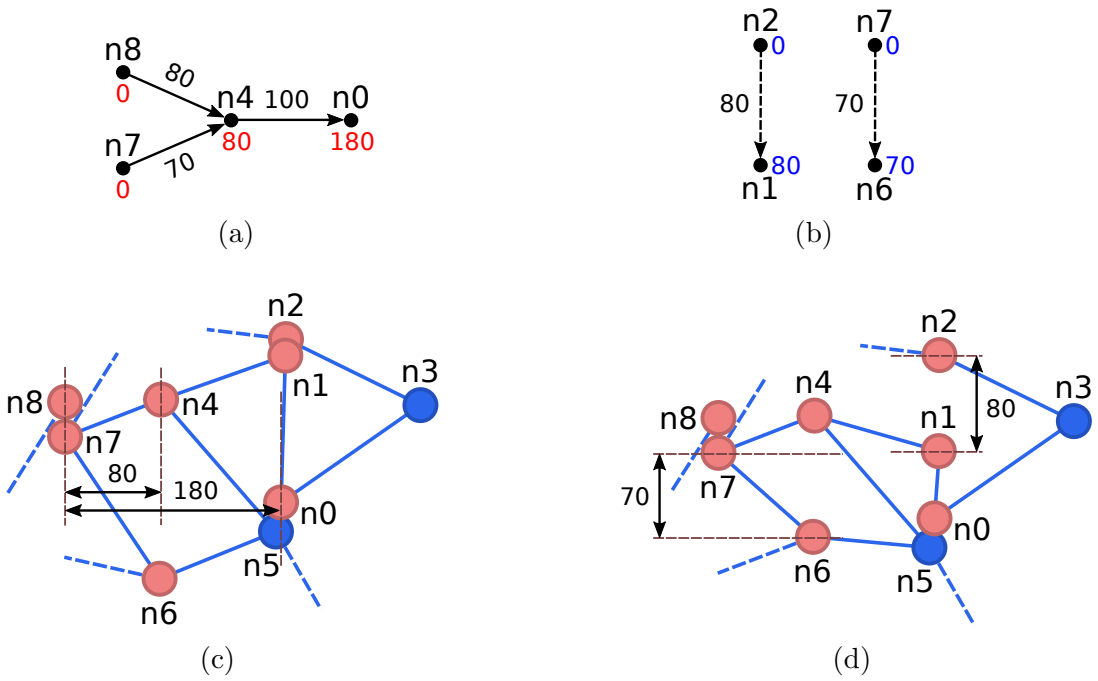


Figure 4.10: Dependency dags (a) D^h and (b) D^v constructed from the relative placement constraints defined on the graph in Figure 4.7 (c) Constraints defined in the x-axis are processed by using D^h. The source nodes $n7$ and $n8$ are initially aligned in their average x coordinate and then $n4$ and $n0$ are placed at the calculated longest distances (80 and 180 units, respectively) from the source nodes. (d) Constraints defined in the y-axis are processed similarly by using the components in D^h. $n1$ and $n6$ are placed to the calculated longest distances (80 and 70 units, respectively) from their corresponding source nodes $n2$ and $n7$.

The dags are then processed in order as follows. For each component of D^h, we first relocate the source nodes to their average x coordinate. Then for the other nodes, we calculate the longest distances of these nodes to the source nodes along the x-axis by using their topological ordering as in Section 4.2.1 and place each node to the appropriate x coordinate. However, if we encounter a fixed node during this placement, we adjust the x coordinates of the predecessors and successors of this node accordingly, to satisfy the minimum separation amount specified by the constraint. After all components of D^h are processed, we continue with the components of D^v by processing them similarly. In this way, all relative placement constraints are enforced. Figure 4.10 shows an example that consists of only relative placement constraints while Algorithm 2 presents all details of the process.

Algorithm 2 Enforcing Relative Placement Constraints

function EnforceConstraints(G, C^f, C^a, C^r)
  for each $dir \in \{h, v\}$ **do**
    $fixedNodes \leftarrow$ nodes in C^f
    $metaToOrgMap \leftarrow \{\}$ ▷ bidirectional map btw meta and original nodes in alignment constraints
    $M \leftarrow \{m_i \mid c_i \in C^a \wedge c_i.dir \neq dir\}$ ▷ a meta node for each alignm. set defined in opposite direction
    $M^f \leftarrow \{m_i \in M \ \wedge \ \exists (x \in c_i.nodes \ \wedge \ x \in fixedNodes)\}$
    $fixedNodes \leftarrow fixedNodes \cup M^f$
    for each $m_i \in M^f$ **do** ▷ set meta node positions based on average position of nodes represented
      $m_i.currPos(dir) \leftarrow$ AveragePos($c_i.nodes, dir$)
    for each $m_i \in M$ **do**
      $metaToOrgMap$.add($m_i, c_i.nodes$)
    $D^{dir} \leftarrow$ CalcDAG($C^r, dir, M, metaToOrgMap$) ▷ use meta nodes here
    EnforceAux($D^{dir}, fixedNodes, M, dir$)

function EnforceAux($D^{dir}, fixedNodes, M, dir$)
  for each component C in D^{dir} **do**
    AlignIndegreeZeroVertices(C, dir) ▷ align zero indegree vertices of C in current direction
    for each node v in C **do**
      $v.predList \leftarrow \{v\}$
      if $v.indegree(dir) = 0$ **then**
        $queue$.enqueue(v)
        $v.newPos(dir) \leftarrow v.currPos(dir)$
      else
        $v.newPos(dir) \leftarrow -\infty$
    while ! $queue$.empty() **do**
      $u \leftarrow queue$.dequeue()
      for each neigbor v of u where $e = (u, v)$ **do**
        $pos \leftarrow u.newPos(dir) + e.weight$
        if $v.newPos(dir) < pos$ **then** ▷ constraint violated
          if $v \in fixedNodes$ **then**
            $v.newPos(dir) \leftarrow v.currPos(dir)$
            if $v.newPos(dir) < pos$ **then** ▷ still violated
              $discr \leftarrow pos - v.newPos(dir)$
              for each node $w \in u.predList$ **do**
                $v.newPos(dir) \leftarrow w.newPos(dir) - discr$
          else
            $v.newPos(dir) \leftarrow pos$
        $v.indegree(dir) \leftarrow v.indegree(dir) - 1$
        if $v.indegree(dir) = 0$ **then**
          $queue$.enqueue(v)
        $v.predList \leftarrow v.predList \cup \{u\}$
    for each node u in C **do**
      if $u = m_i \in M$ **then**
        for each node $v \in c_i.nodes$ **do**
          $v.currPos(dir) \leftarrow m_i.newPos(dir)$
      else
        $u.currPos(dir) \leftarrow u.newPos(dir)$

4.3 Phase III: Polishing Phase

In the last phase, we apply a modified version of the CoSE algorithm on the constrained draft layout incrementally (starting from the current node positions) to remove any node-node overlaps, respect the compound structures, and refine the layout by minimizing the stress of the physical model while maintaining the satisfied constraints.

The CoSE algorithm does not take the user-specified placement constraints into account. In each iteration, the algorithm first calculates the displacement amounts for all nodes based on the applied forces (repulsion, spring and gravitational) on the nodes and then moves them according to these displacement amounts. However, the positioning of the nodes based on this movement strategy may violate the already established constraints. Hence, we add an intermediate step between these two operations to maintain the satisfied constraints during iterations. This intermediate step adjusts (i.e., limits) the calculated displacement amounts of the constrained nodes so that none of the constraints are violated at the end of movements. The details of the intermediate step are as follows:

- First, displacement amounts of fixed nodes are adjusted to 0 along x and y axes.
- Secondly, we adjust the displacement amounts of the nodes involved in a vertical (horizontal) alignment constraint to their average displacement value in x (y) direction. However, if any of these nodes is a fixed node, then the displacement values of all nodes in the corresponding direction are set to 0.
- Lastly, for a node that is part of a relative placement constraint, we adjust its displacement amount in the corresponding direction so that all other constraints in which this node is involved will not be violated. In other words, if such a node is also a fixed node or involved in an alignment constraint, its displacement amount is first updated in previous stages. Then we check all relative placement constraints in which this node is involved

one by one and continue to update the node's displacement amount to the maximum value that will not violate any of these constraints.

As a result of the intermediate step, each constrained node can have a displacement amount that is less than or equal to the calculated displacement amount before. This step does not change the displacement amounts of the unconstrained nodes. The logic of this intermediate step is similar to the one used in [42]. They behave to the nodes in alignment and relative placement constraints like there exists a "rigid stick" between these nodes and move them together by the same amount. We behave similarly to the nodes in alignment constraints, while the nodes in the relative placement constraints can have different displacement amounts giving them more freedom. With this modified movement strategy, enforced constraints are not violated during the iterations of the CoSE algorithm and hence they stay satisfied in the final layout.

4.4 Time Complexity

We now present an in-depth time complexity analysis of the fCoSE algorithm. Phase I is expected to run in $\mathcal{O}(n+m)$ time for a compound graph $G = (V, E, F)$, where $|V| = n$ and $|E| = m$. The preprocessing step applied to convert a disconnected compound graph into a connected simple graph requires a constant number of BFS operations to find the disconnected components and the nodes with the minimum degree, assuming there is a constant number of levels in the inclusion tree as in most real-life graphs. The CMDS algorithm proposed by Civril et al. [24] also works in linear time with respect to the number of nodes and edges as explained in Section 2.2.2.1. The postprocessing step, lastly, requires a one-time traversal of all simple nodes in the graph to calculate the new positions and dimensions of the compound nodes, taking $\mathcal{O}(n)$ time as well.

Similarly, Phase II is expected to run in $\mathcal{O}(n + m)$ time. The most costly operations in this phase are the construction and processing of dependency graphs used when we base the transformation on relative placement constraints and

during the enforcement of the same type of constraints. These operations require a constant number of BFS traversals to find the disconnected components and solution to the longest path problem in a dag. Transformation based on the fixed node or alignment constraints and the enforcement of these types of constraints are relatively simple operations which can be done in $\mathcal{O}(n)$. The computation of the transformation matrix which requires the multiplication of matrices with dimensions $2 \times n$ and $n \times 2$ together with an SVD operation on a 2×2 matrix can also be done in linear time in the number of nodes.

As we apply the modified version of CoSE algorithm incrementally in Phase III, starting with a low cooling factor thanks to the previously formed draft layout, the number of iterations it requires reduces significantly. Each iteration of the original CoSE algorithm takes $\mathcal{O}(n+m)$ time. The operations applied to maintain the fixed node and alignment constraints in the intermediate step we introduce do not affect the asymptotic complexity of an iteration. The only negative effect can emerge when each node is involved in $\mathcal{O}(n)$ relative placement constraints. However, because the support of fCoSE for compound nodes and prevention of node-node overlaps are not based on the constraints as in CoLa that may require a quadratic number of relative placement constraints in the number of nodes (on top of those defined by the user), we expect each node to have at most constant number of relative placement constraints, a reasonable assumption for user-specified constraints, this keeps the complexity of an iteration to $\mathcal{O}(n + m)$.

As a result, while the first two phases of the fCoSE are expected to work in $\mathcal{O}(n + m)$ time, the overall run time of the algorithm depends on the number of iterations in Phase III where each iteration is also expected to run in $\mathcal{O}(n + m)$ time like CoSE. We expect, however, much fewer iterations when compared to CoSE as our algorithm starts from a draft layout instead of starting from scratch with totally random positions.

Chapter 5

Evaluation

We evaluated the quality and performance of fCoSE by comparing it with CoLa on a dataset that we constructed from real-life and randomly generated compound graphs. We chose CoLa for comparison because it is the most similar algorithm to fCoSE with its support for non-uniform node dimensions, compound structure with more than one level of nesting and similar constraint types. The most common way to evaluate a graph layout in terms of quality is to focus on metrics related to the generally accepted aesthetic criteria. Hence, we compared values for average edge length, edge crossings, node-node overlaps, node-edge overlaps and total area metrics. We also made measurements on the execution duration of both algorithms to evaluate their run time performances. This evaluation process was repeated on experiment setups constructed with varying ratios of different constraint types. In addition, we compared the run time performances of fCoSE and its predecessor CoSE on constraint-free graphs.

5.1 Experiment Setup

We implemented fCoSE in JavaScript programming language as an extension to Cytoscape.js [60], a graph visualization and analysis library. Other layout

algorithms that we used for comparison, CoLa and CoSE, are also available as Cytoscape.js extensions. Hence, we used these three extension libraries and an ordinary computer with Intel i7-4790 3.60GHz x 4 CPU and 16GB RAM to run our experiments.

5.2 Dataset

We aim fCoSE to work in small to medium-sized graphs up to several thousand nodes [61][62] which is the common scale for many interactive applications designed for real-life graphs. The visualization and analysis of larger graphs are much harder to handle and require complexity management techniques [16] to be used to reduce the graph size and simplify working with them. To this end, we used both real-life graphs and two randomly generated compound graph datasets with 10 to 5000 nodes to evaluate the quality and performance of fCoSE and compare it with other algorithms, CoLa and CoSE.

The real-life graphs we used can be seen in Figure 5.1 through Figure 5.5. Because there is no readily available compound graph dataset proposed in the literature, we generated two new ones by using simple graphs from two different sources. One of these sources is the Rome graph dataset [63] which is one of the benchmark datasets used frequently in graph visualization with biconnected, undirected, and 4-planar graphs and the other one is the Network Repository website [64] that contains benchmark datasets. The generation of the compound graph datasets was performed as follows. We first randomly selected 81 graphs from the Rome dataset (60 of them are small-sized with $10 - 200$ nodes and 21 of them are medium-sized with $250 - 5000$ nodes) with $d(G) \leq 3$ and 10 denser graphs (5 of them are small-sized with $20 - 209$ nodes and 5 of them are medium-sized with $530 - 4245$ nodes) from the Network Repository with $4.3 \leq d(G) \leq 7$. This selection process was done to reduce the computation time of the tests by eliminating some of the graphs that are many of the same size. We then applied Markov Clustering Algorithm [65] on each selected graph to compute the clusters within. For half of these clusters selected randomly, we created a compound node

for each cluster and moved the nodes in the clusters into their corresponding compound nodes. We repeated the same procedure inside the newly created compound nodes two more times so that we can generate compound graphs with an inclusion tree depth up to 3. Please note that a generated compound graph in this manner can have an inclusion tree depth less than 3 when there are not enough number of nodes to construct more than one cluster in any branch. This process, however, creates inter-graph edges whose both ends are simple nodes. Hence, we lastly changed the source and target nodes of some inter-graph edges randomly to one of their noncommon ancestors (necessarily compound nodes) in the inclusion tree so that we obtain connections to some compound nodes as well.

After we finished generating random compound graphs, we added random constraints to each graph this time. For this purpose, we added fixed node, alignment, relative placement and hybrid (which includes all constraint types) constraints to the 25%, 50%, 75% and 100% of all nodes for various test cases (the only exception to this process is to constrain 100% of the nodes with fixed node constraints which does not require a layout to be applied). Constraints were added as follows.

For adding fixed node constraints, we first generated a draft layout of the graph by using only Phase I of the fCoSE algorithm. Then we randomly selected a specified portion of the simple nodes and assigned an anchor position to each one which is a random position near (up to ± 30 units in both x and y coordinates) the one calculated in the draft layout. The reason why we use a draft layout to generate anchor positions is that we want them to be in line with the overall structure of the graph, whereas a totally random generation of those positions causes a tangled layout with unnecessarily long edges. We added alignment constraints by simply matching each node from the randomly selected portion of all simple nodes with its random adjacent neighbor either in the vertical or horizontal direction. Relative placement constraints are also added in the same way by using a default minimum gap value as the separation amount. To constrain a graph with hybrid constraints, we added an equal amount of each constraint type using the approaches mentioned above (i.e., to constrain 75% of the nodes, we added one type of constraint for every 25%). It should be noted that we paid attention

not to create conflicting constraints during this process. For instance, we do not allow both nodes in an alignment constraint to have a fixed node constraint as well. Moreover, if the pair of nodes in a relative placement constraint also has an alignment constraint, we assign the same direction to both constraints.

Figure 5.6 through Figure 5.8 show sample random constrained graphs generated in this manner.

5.3 Results and Discussion

The results of the experiments conducted on both real-life graphs and randomly generated datasets show the superiority of fCoSE over CoLa in terms of both run time performance and visual quality. A comparison of the two algorithms in some real-life graphs, each with a domain-specific constraint set, such as the dependency graph, the underwater wireless sensor network and the call graph can be seen in Figure 5.1 through Figure 5.5. In each of these examples, fCoSE runs faster than CoLa and achieves better values in terms of the quality metrics in general.

The experiments on the randomly generated graphs were performed by repeating each test 5 times with a new constraint set in each run and averaging the results. Here we should note that we can compare fCoSE and CoLa in only small-sized graphs because of the high computational cost of CoLa that prevents it to scale to larger graphs.

We first present the results for the compound graph dataset generated from the Rome graphs (see Figures 5.6 through 5.8 for some examples). As expected, fCoSE is much faster than CoLa in all constraint types in terms of *run time* performance (Figure 5.10). If we look at the quality metrics, fCoSE generates up to 27% shorter *average edge lengths* on the graphs with fixed node and hybrid constraints, while the results are comparable for the other constraint types (Figure 5.11). fCoSE also yields 37 to 74% fewer *edge crossings* and 50 to 81%

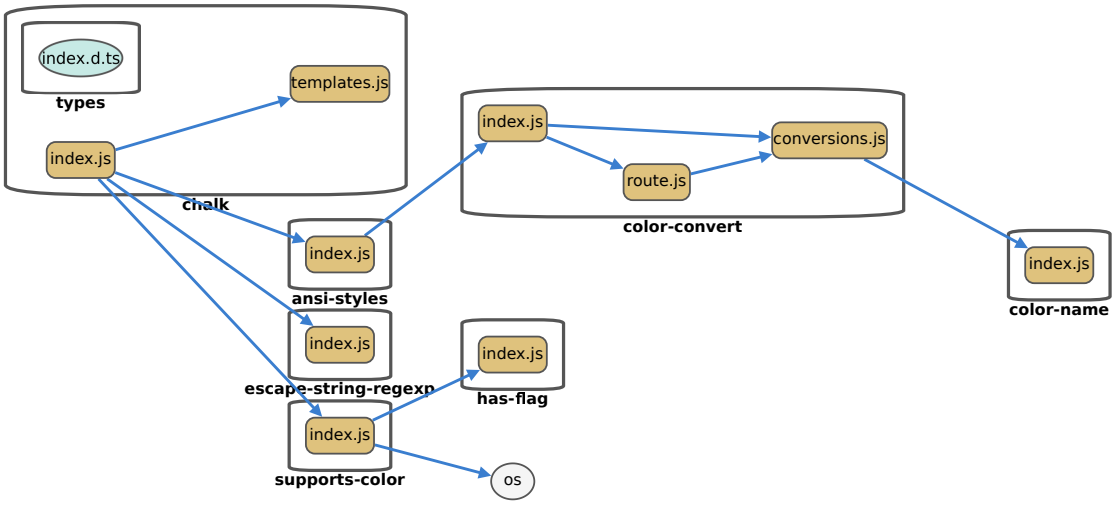
fewer *node-edge overlaps* in all constraint types (Figure 5.12 and Figure 5.14). In terms of the number of *node-node overlaps* and *total area*, both algorithms present comparable results for fixed node and hybrid constraints. For the other constraint types, however, CoLa generates less number of *node-node overlaps* and much smaller *total area* when compared to fCoSE (Figure 5.13 and Figure 5.15). Our observation from these results is that CoLa tends to generate more compact layouts at the expense of poor quality in terms of other aesthetic metrics.

We also performed experiments on the medium-sized graphs that compare fCoSE and its base method, CoSE, to see the effect of generating a quick draft layout on the run time performance. Results show that fCoSE runs up to 2 times as fast as CoSE in constraint-free graphs (Figure 5.16). In addition, we observed that the run time of fCoSE on constrained graphs is also better than the run time of CoSE on constraint-free graphs, although fCoSE loses extra time to satisfy constraints. Our another observation is that the alignment and relative placement constraints cause a slight increase in the run time of fCoSE while the other constraint types decrease it, probably with the effect of fixed node constraints which yields faster convergence. Moreover, Figure 5.9 shows the positive effect of generating a draft layout in Phase I of fCoSE on the layout quality, as well as the run time, instead of directly applying CoSE algorithm from scratch.

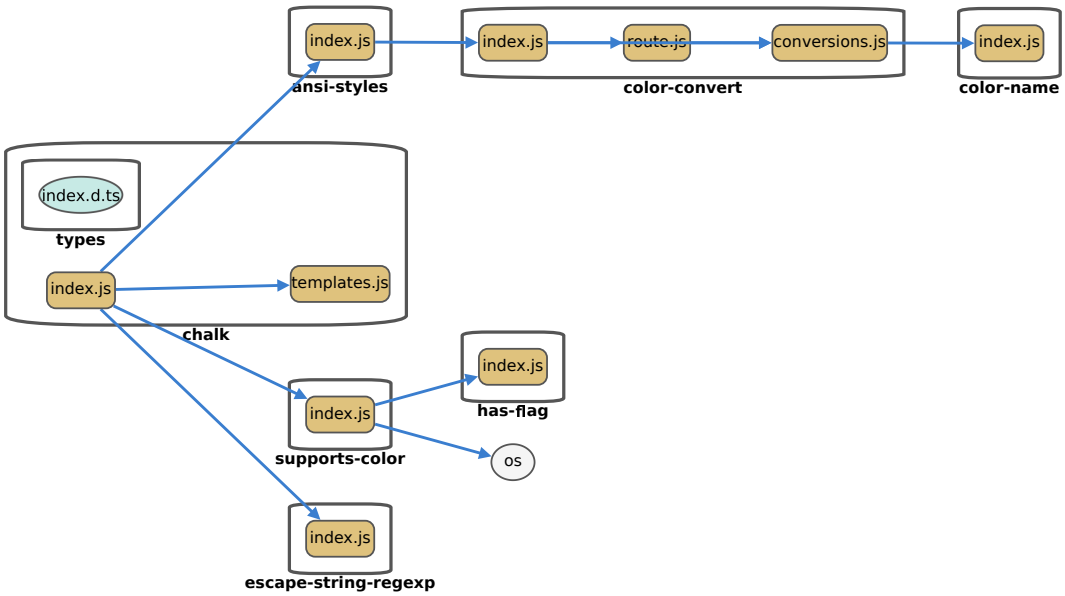
The results for the quality performance of fCoSE on medium-sized graphs in terms of *average edge length*, *number of edge crossings*, *number of node-node overlaps*, *number of node-edge overlaps* and *total area* are presented in Figure 5.17 through Figure 5.21, respectively. An important point to notice here is that while the ratio of alignment and relative placement constraints increases, the quality of the layout deteriorates in terms of these metrics, because it becomes harder to satisfy these types of constraints. On the other hand, the ratio of fixed node constraints has a negligible effect on these metrics.

Lastly, we present the results for the denser dataset generated from the graphs on the Network Repository website. For this dataset, we performed experiments by constraining the graphs only with hybrid constraints (in increasing ratios) and evaluated middle-sized graphs with only fCoSE as before. Table 5.1 shows

that the behavior of the two algorithms is similar, in terms of both run time and quality metrics, when compared with the results from the other datasets we analyzed above. However, the problem with these graphs is that they are prone to turn into "hairballs" as their densities increase, making the analysis difficult and obsolete.

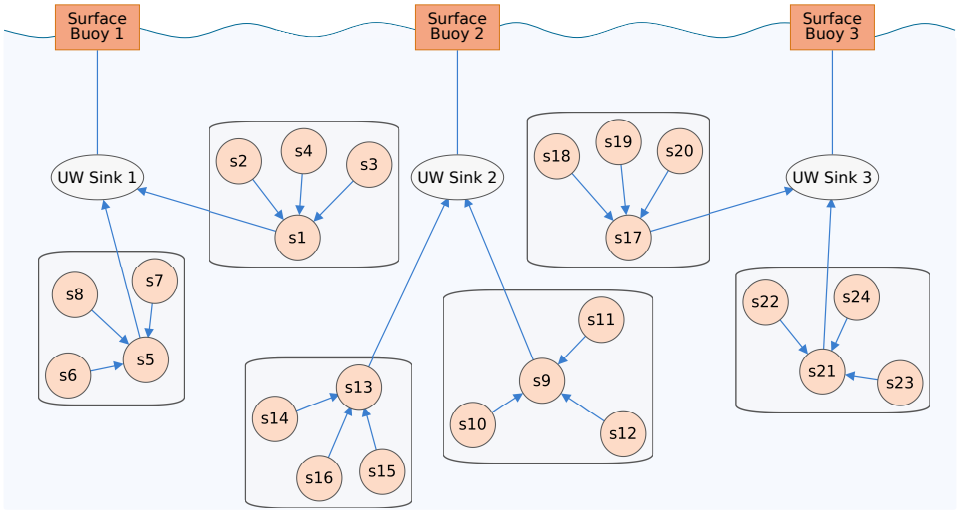


(a)

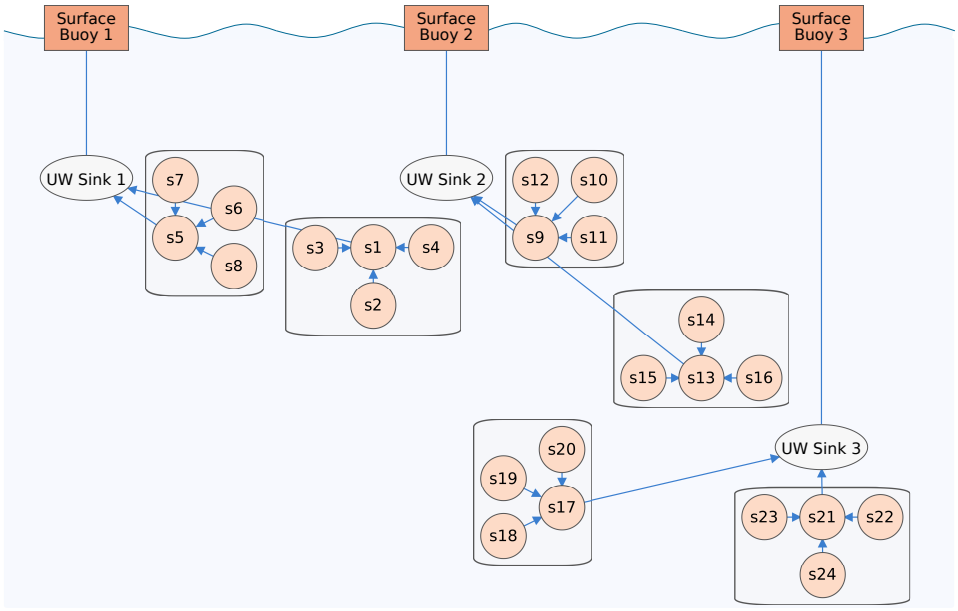


(b)

Figure 5.1: A sample dependency graph of a JavaScript project [4] with $|V| = 20$, $|E| = 11$ and $d(G) = 1.1$ where the nodes are constrained with relative placement constraints in such a way that each target node will be on the right of the source node while alignment constraints are also introduced for the nodes in the same level to be aligned vertically. Sample layouts generated by (a) fCoSE and (b) CoLa have the following performance and quality metrics (fCoSE - CoLa): run time: 11.79 ms - 167.14 ms, average edge length: 122.19 - 113.23, number of edge crossings: 0 - 2, number of node-node overlaps: 0 - 0, number of node-edge overlaps: 0 - 1, and total area: 267314 - 303002 square units.

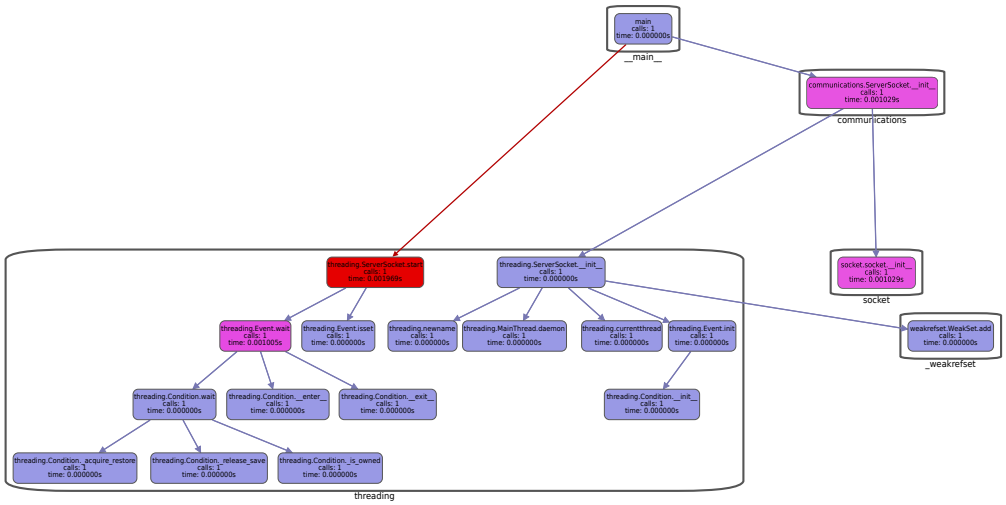


(a)

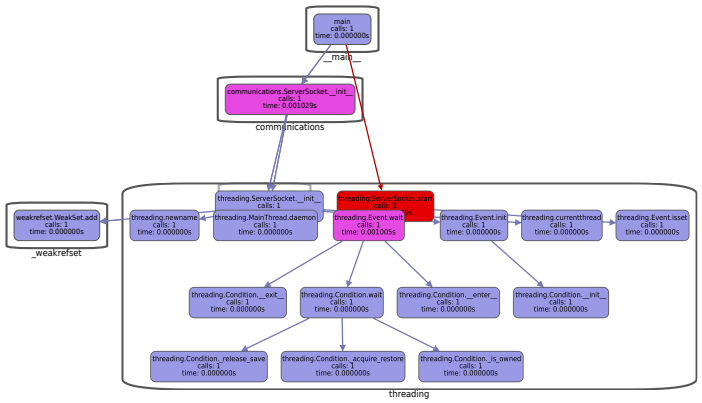


(b)

Figure 5.2: An underwater wireless sensor network with $|V| = 36$, $|E| = 27$ and $d(G) = 1.5$ where the surface buoys have fixed node constraints to keep them at anchor positions on the surface of the water. Underwater sinks are constrained to be in alignment with the buoys vertically. Sensor nodes in clusters have no constraints on their positioning with the only exception that cluster heads s1, s5, s9, s13, s17 should stay below the underwater sinks. Sample layouts generated by (a) fCoSE and (b) CoLa have the following performance and quality metrics (fCoSE - CoLa): run time: 12.51 ms - 369.02 ms, average edge length: 127.25 - 104, number of edge crossings: 0 - 0, number of node-node overlaps: 0 - 0, number of node-edge overlaps: 0 - 4, and total area: 1250680 - 1447749 square units.

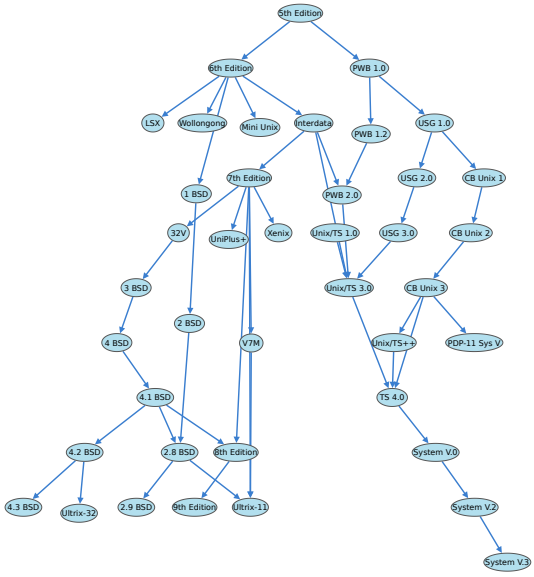


(a)

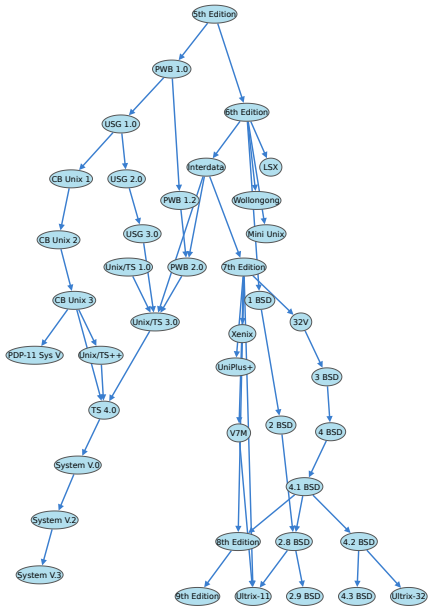


(b)

Figure 5.3: A sample call graph of a Python application [10] with $|V| = 24$, $|E| = 18$ and $d(G) = 1.5$ where the nodes are constrained with relative placement constraints in such a way that each target node will be below of the source node while alignment constraints are also introduced for the nodes in the same level to be aligned horizontally. Sample layouts generated by (a) fCoSE and (b) CoLa have the following performance and quality metrics (fCoSE - CoLa): run time: 19.314 ms - 174.75 ms, average edge length: 169.57 - 125.32, number of edge crossings: 0 - 2, number of node-node overlaps: 0 - 8, number of node-edge overlaps: 0 - 14, and total area: 1191812 - 653876 square units.

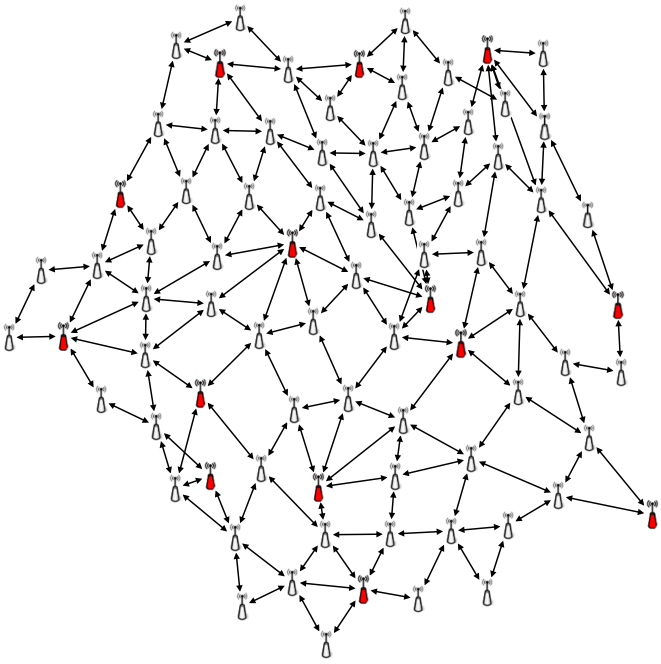


(a)

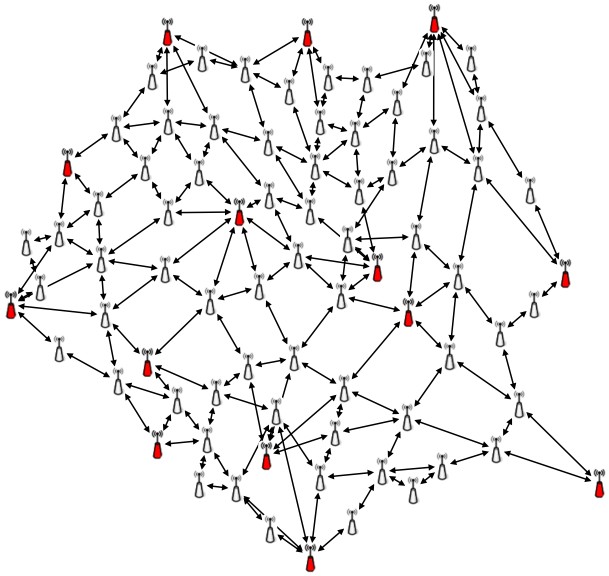


(b)

Figure 5.4: Unix family tree [11] with $|V| = 41$, $|E| = 49$ and $d(G) = 2.38$ where the nodes are constrained with relative placement constraints in such a way that each target node will be below of the source node. Sample layouts generated by (a) fCoSE and (b) CoLa have the following performance and quality metrics (fCoSE - CoLa): run time: 15.25 ms - 317.78 ms, average edge length: 99.37 - 101.93, number of edge crossings: 3 - 4, number of node-node overlaps: 0 - 0, number of node-edge overlaps: 9 - 19, and total area: 579961 - 492325 square units.



(a)



(b)

Figure 5.5: A sample wireless sensor network [5] with $|V| = 79$, $|E| = 156$ and $d(G) = 3.94$ where the nodes with fixed node constraints are shown in red. Sample layouts generated by (a) fCoSE and (b) CoLa have the following performance and quality metrics (fCoSE - CoLa): run time: 33.95 ms - 614.79 ms, average edge length: 65.42 - 58.36, number of edge crossings: 5 - 12, number of node-node overlaps: 0 - 0, number of node-edge overlaps: 2 - 9, and total area: 782448 - 623572 square units.

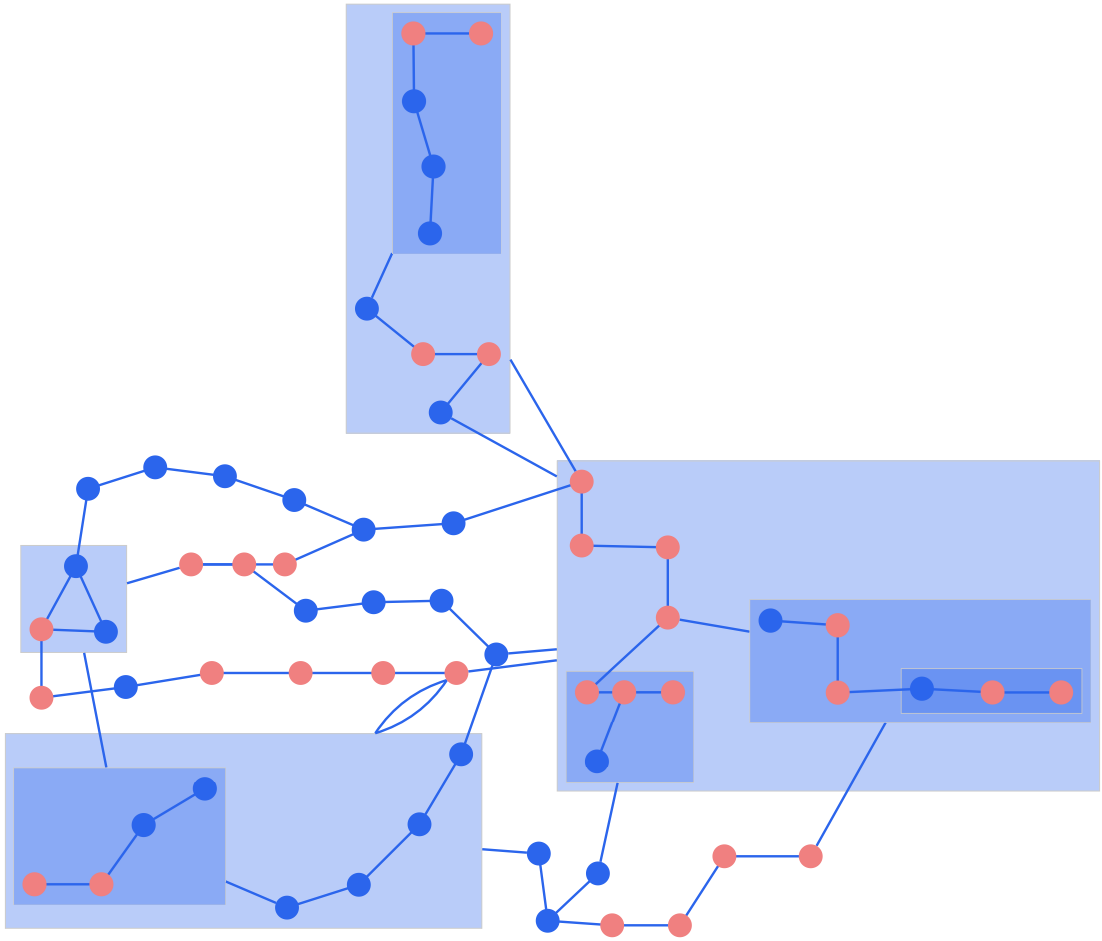


Figure 5.6: A sample layout of a *small-sized* graph from our dataset generated using Rome graphs ($|V| = 69$, $|E| = 69$, $d(G) = 2$). 50% of the simple nodes have alignment constraints shown in red. The layout is generated in 29.5 ms and has the following quality metrics: average edge length: 97.7, number of edge crossings: 2, number of node-node overlaps: 0, number of node-edge overlaps: 1, and total area: 1620261 square units.

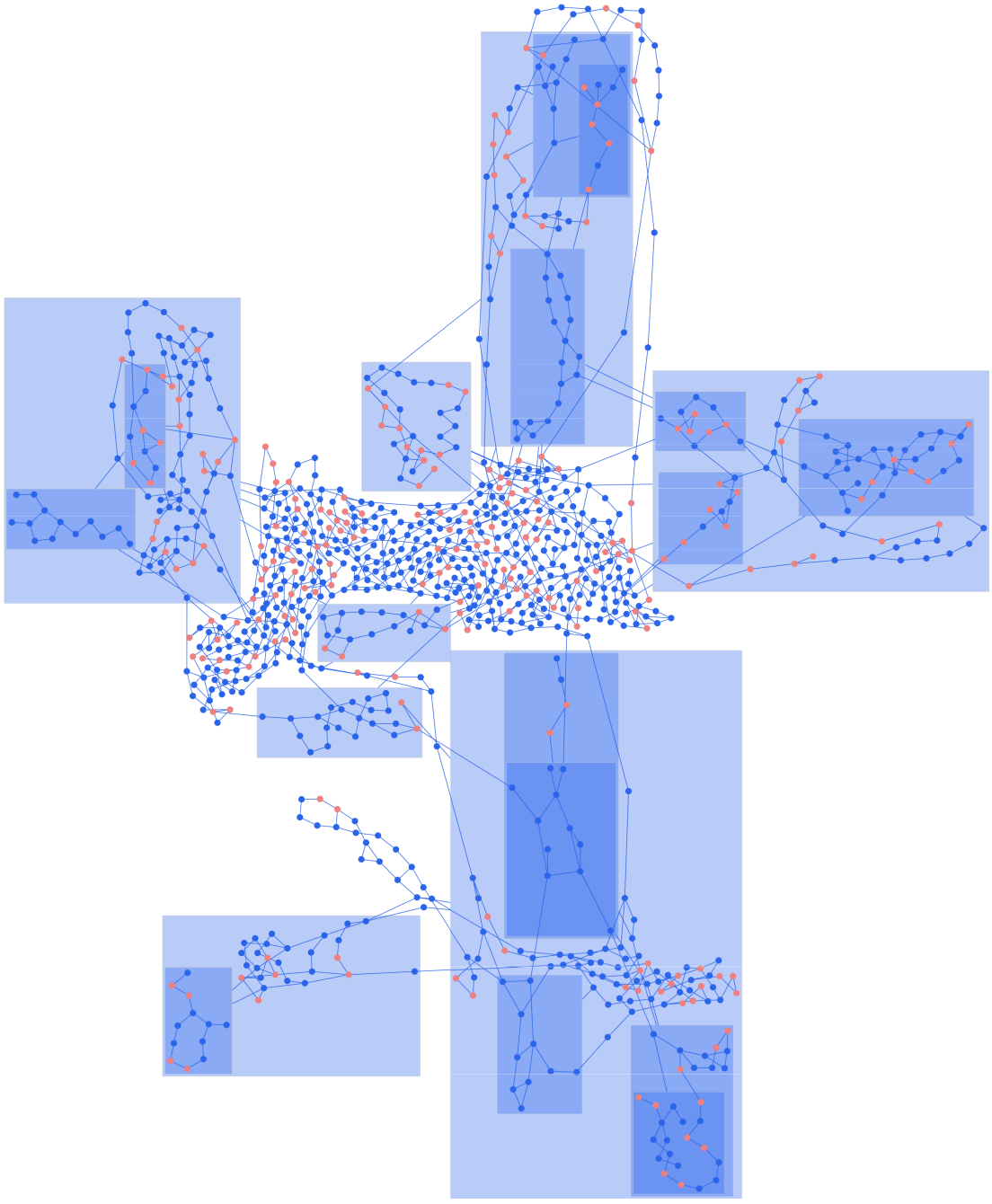


Figure 5.7: A sample layout of a *medium-sized* graph from our dataset generated using Rome graphs ($|V| = 1022$, $|E| = 1228$, $d(G) = 2.4$). 25% of the simple nodes have alignment constraints shown in red. The layout is generated in 1031.6 ms and has the following quality metrics: average edge length: 130.2, number of edge crossings: 880, number of node-node overlaps: 29, number of node-edge overlaps: 513, and total area: 26225364 square units.

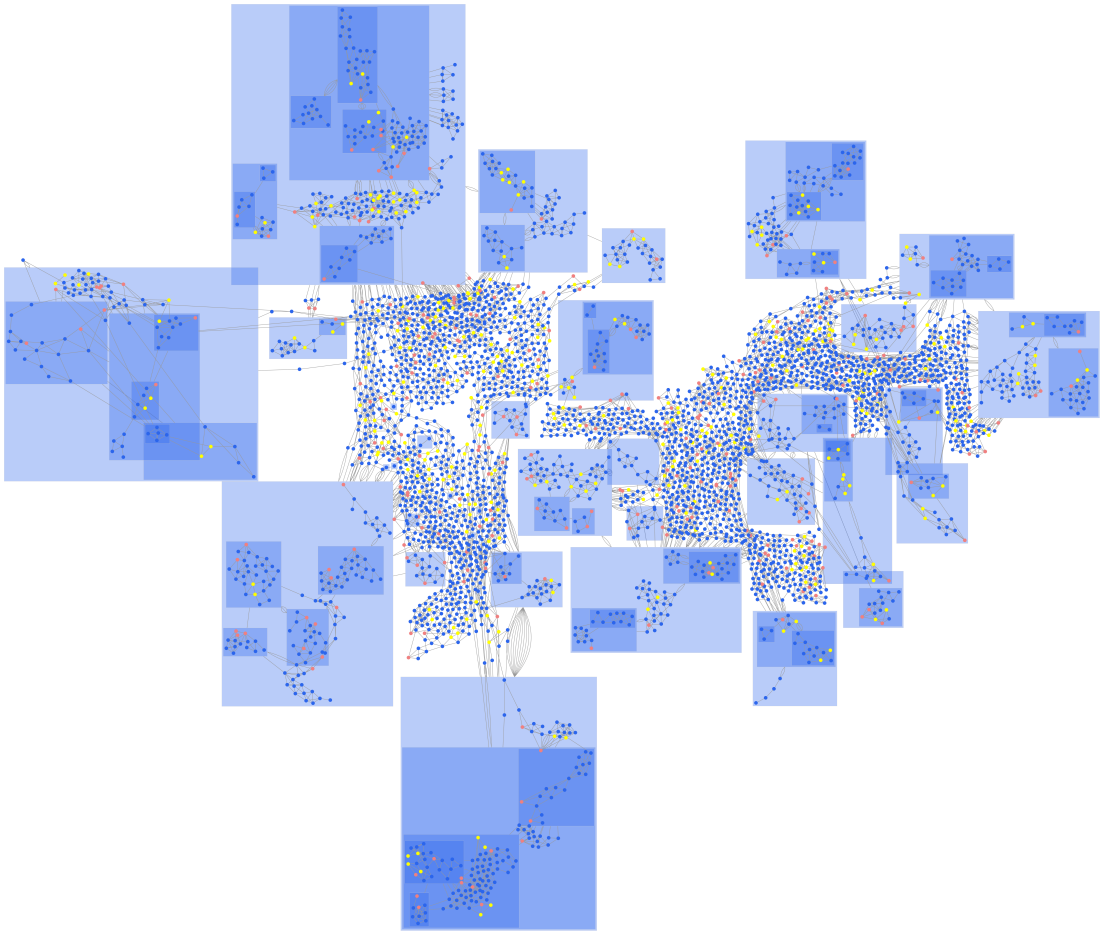


Figure 5.8: A sample layout of a *medium-sized* graph from our dataset generated using Rome graphs ($|V| = 4245$,—E—=10319, $d(G) = 4.86$). 25% of the simple nodes have hybrid constraints where the nodes with fixed node, alignment and relative placement constraints are shown in red, green and yellow, respectively. The layout is generated in 7189.8 ms and has the following quality metrics: average edge length: 85.7, number of edge crossings: 16755, number of node-node overlaps: 785, number of node-edge overlaps: 9627, and total area: 70108183 square units.

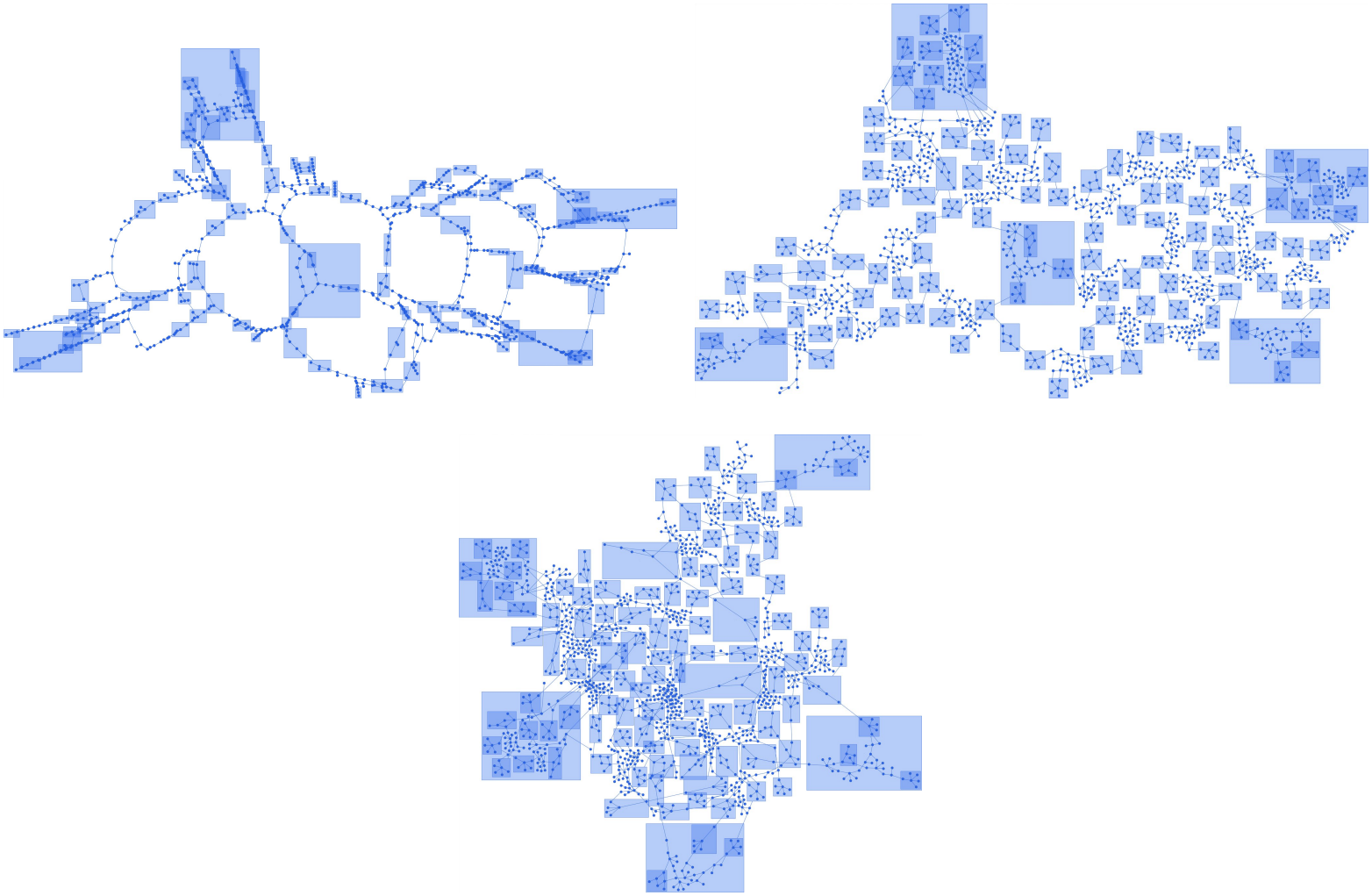


Figure 5.9: Comparison between fCoSE and CoSE algorithms on a randomly generated compound graph ($|V| = 1525$, $|E| = 1527$, $d(G) = 2$). Draft layout generated after Phase I of fCoSE, run time: 123.4 ms (top-left). Final layout after Phase III is applied directly on top of the draft layout for polishing, run time 175.9 ms (top-right). Same graph laid out with CoSE, run time: 579.4 ms (bottom). Notice how the construction of a draft layout as a first step of fCoSE beautifies the layout and makes it run faster (299.3 ms in total vs 579.4 ms) when compared to directly applying CoSE algorithm.

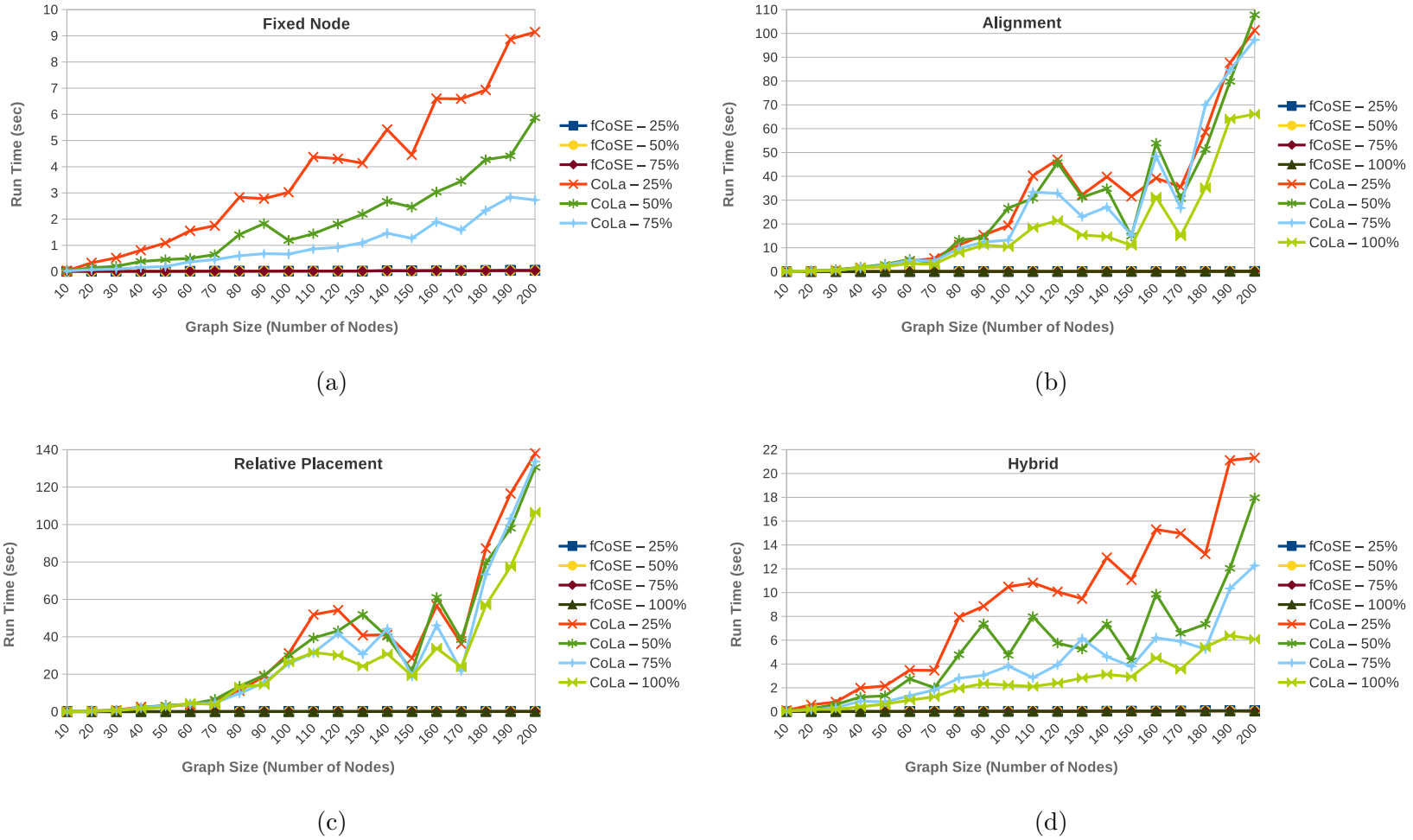


Figure 5.10: fCoSE vs CoLa *run time* comparison in small-sized graphs constrained with different ratios of (a) fixed node, (b) alignment, (c) relative placement and (d) hybrid constraints

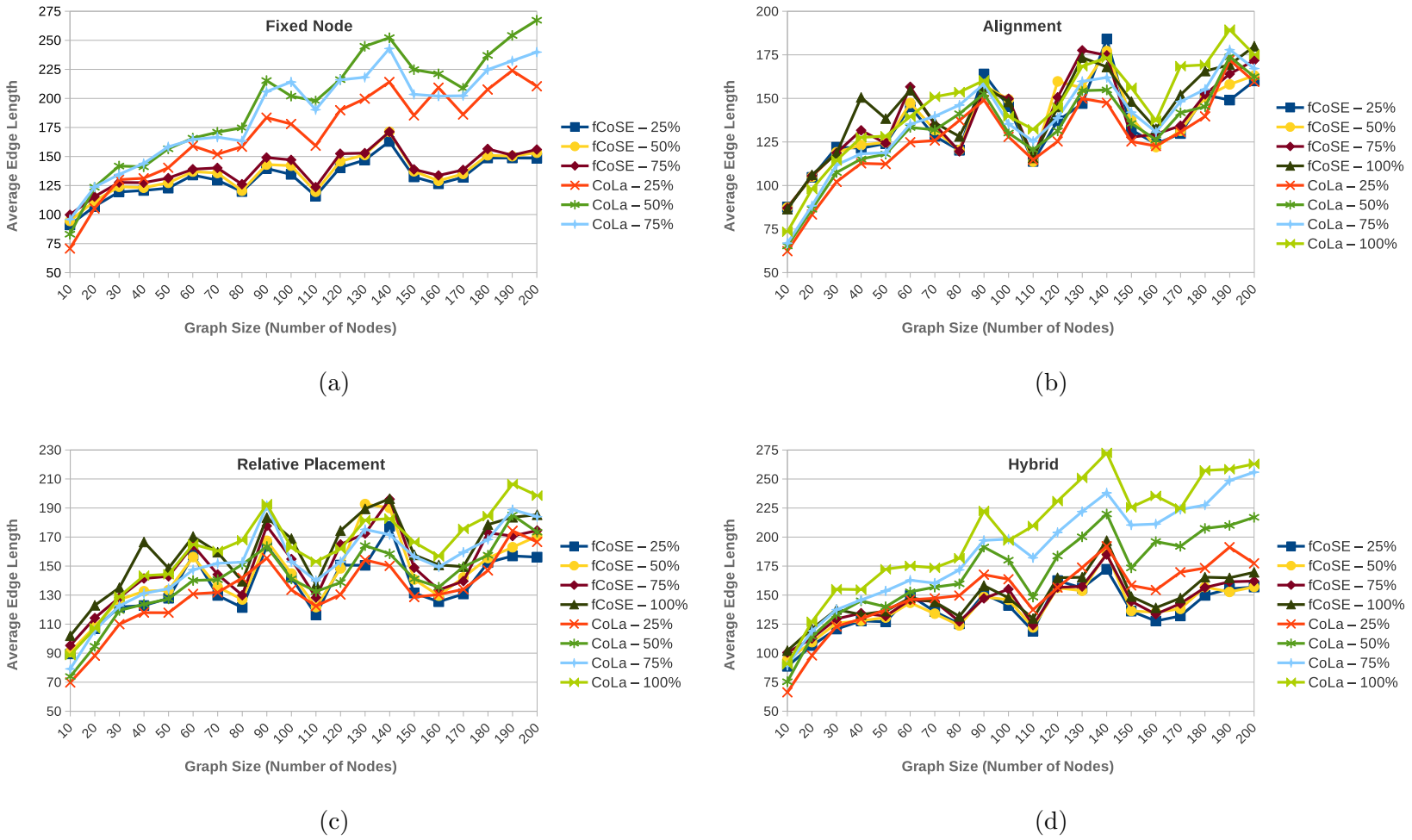


Figure 5.11: fCoSE vs CoLa *average edge length* comparison in small-sized graphs constrained with different ratios of (a) fixed node, (b) alignment, (c) relative placement and (d) hybrid constraints

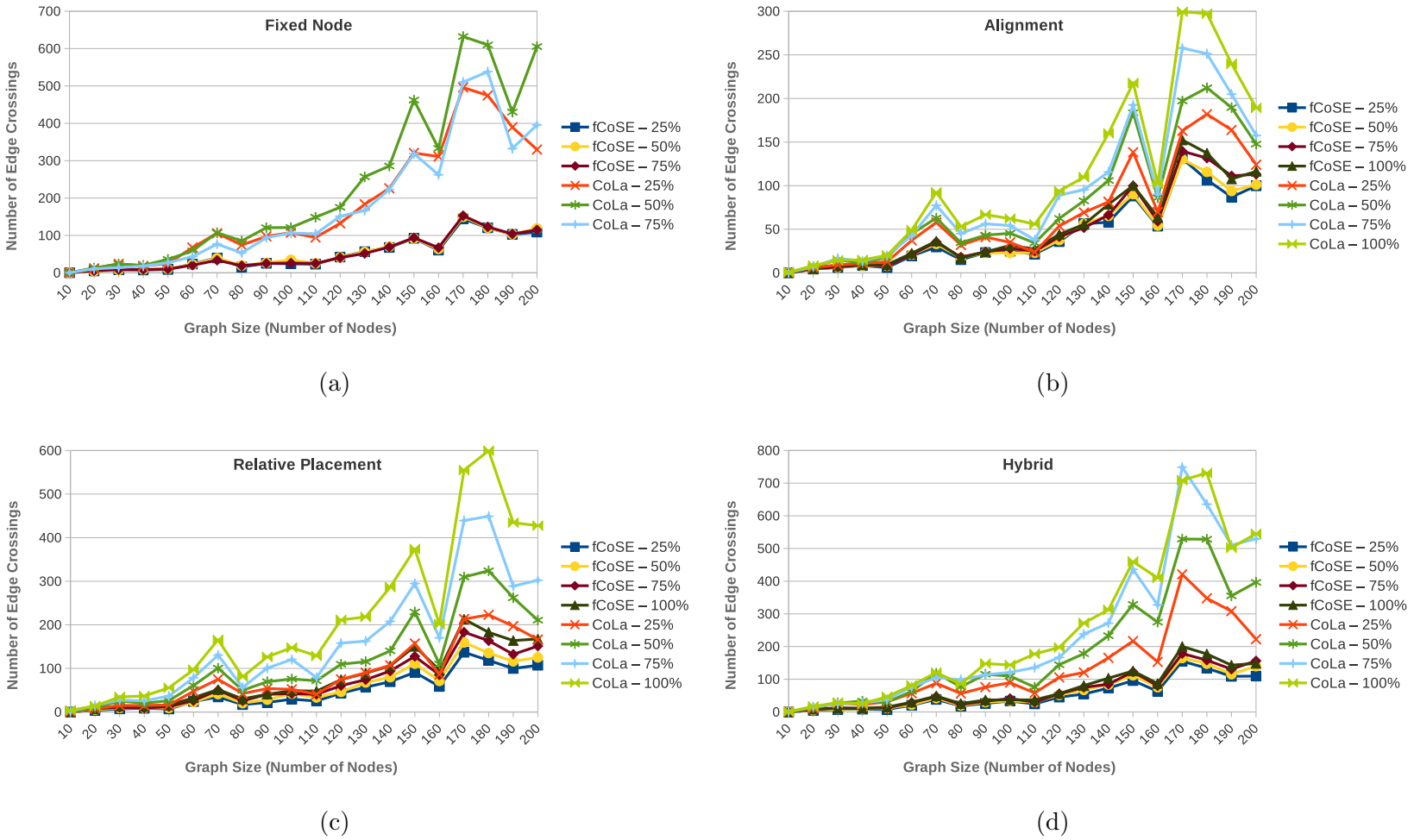


Figure 5.12: fCoSE vs CoLa *edge crossings* comparison in small-sized graphs constrained with different ratios of (a) fixed node, (b) alignment, (c) relative placement and (d) hybrid constraints

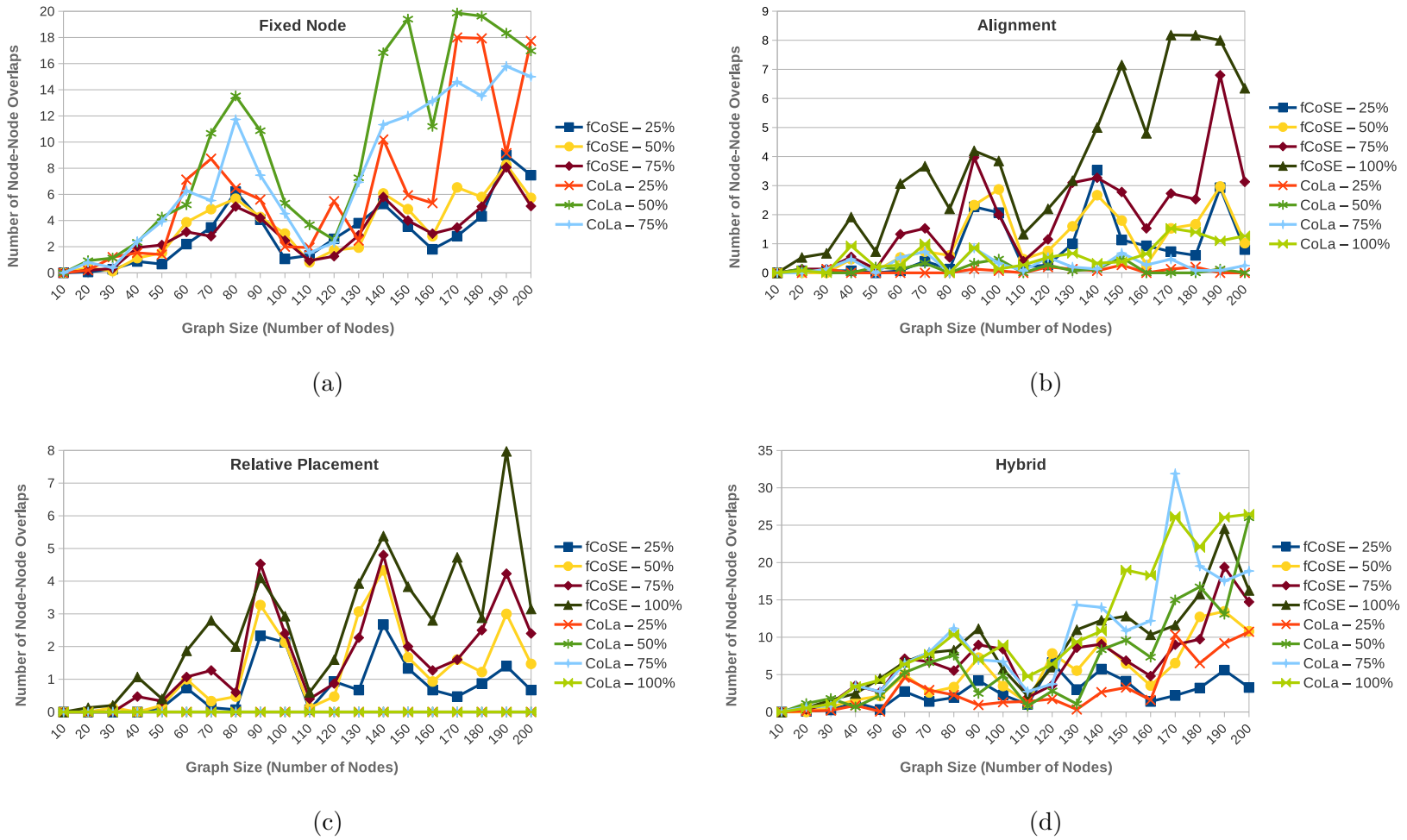


Figure 5.13: fCoSE vs CoLa *node-node overlaps* comparison in small-sized graphs constrained with different ratios of (a) fixed node, (b) alignment, (c) relative placement and (d) hybrid constraints

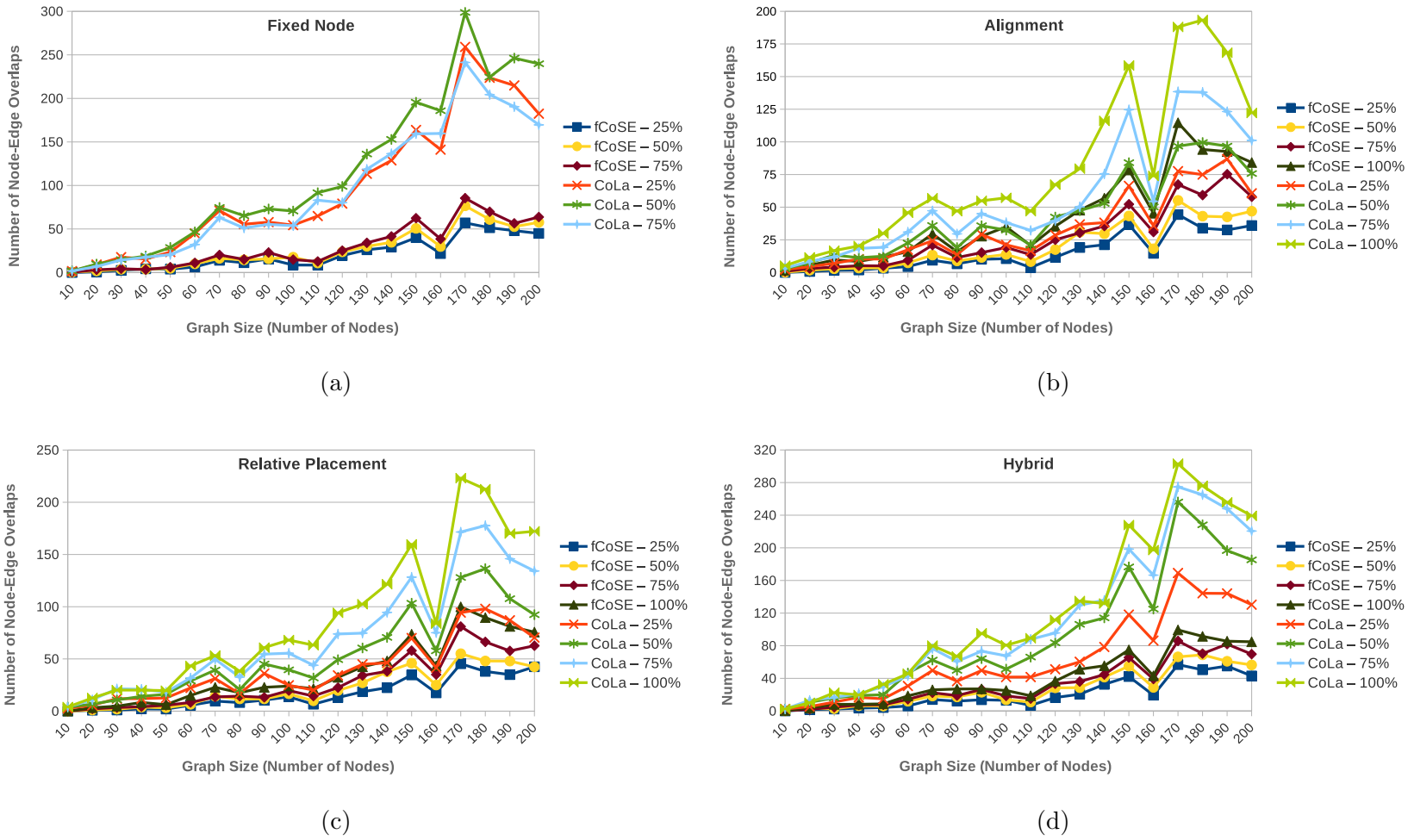


Figure 5.14: fCoSE vs CoLa *node-edge overlaps* comparison in small-sized graphs constrained with different ratios of (a) fixed node, (b) alignment, (c) relative placement and (d) hybrid constraints

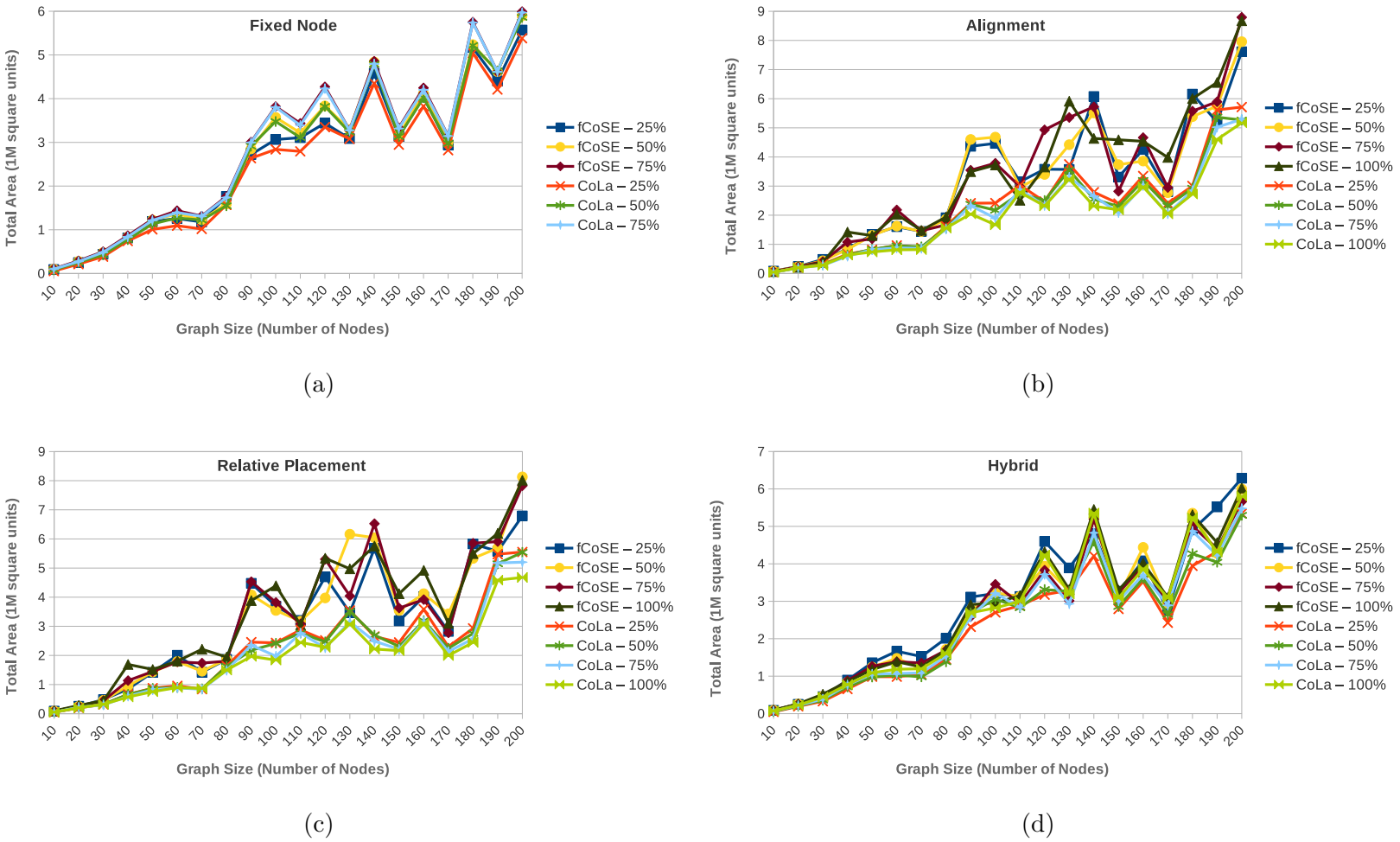


Figure 5.15: fCoSE vs CoLa *total area* (in 10^6 square units) comparison in small-sized graphs constrained with different ratios of (a) fixed node, (b) alignment, (c) relative placement and (d) hybrid constraints

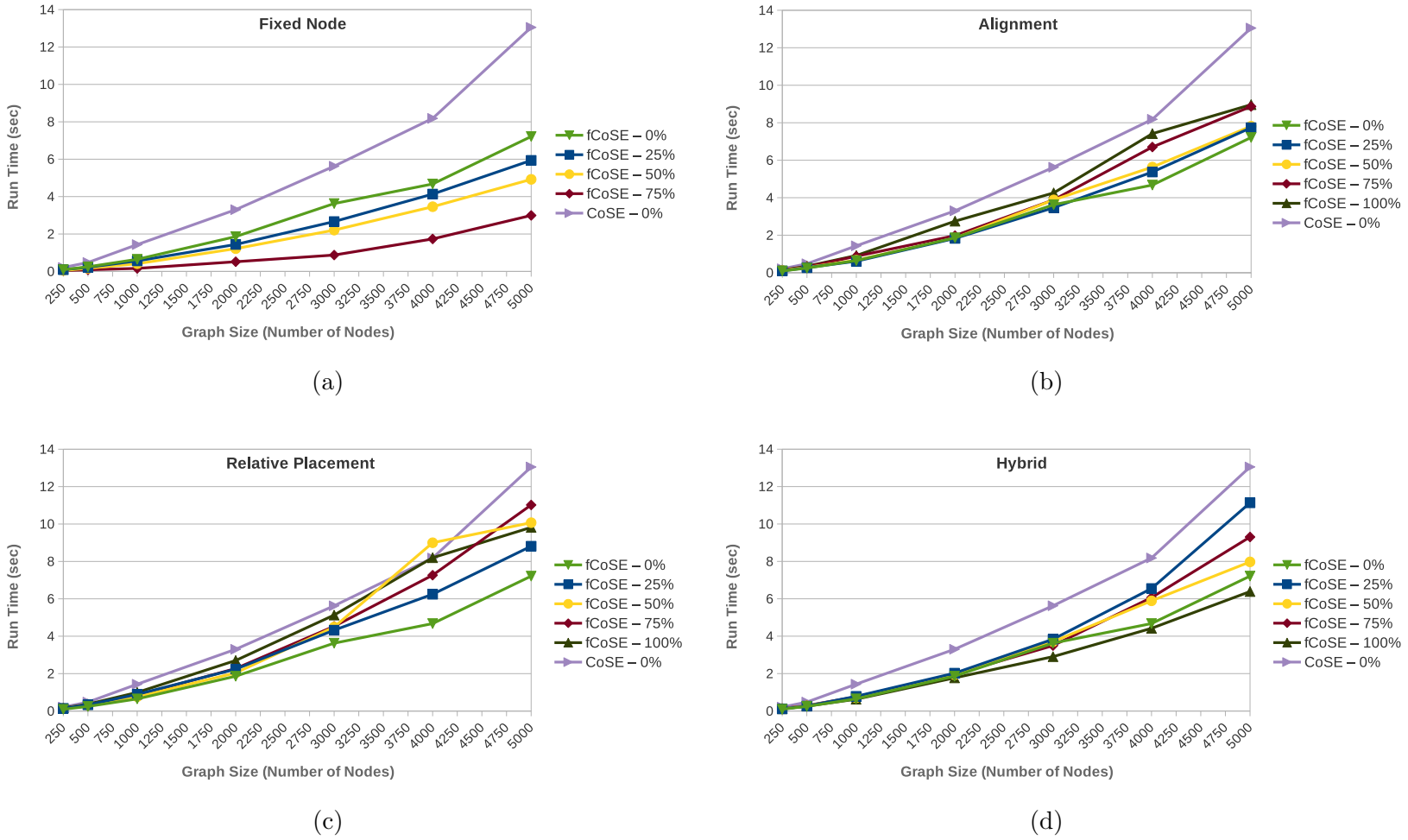


Figure 5.16: fCoSE vs CoSE *run time* comparison in medium-sized graphs constrained with different ratios of (a) fixed node, (b) alignment, (c) relative placement and (d) hybrid constraints

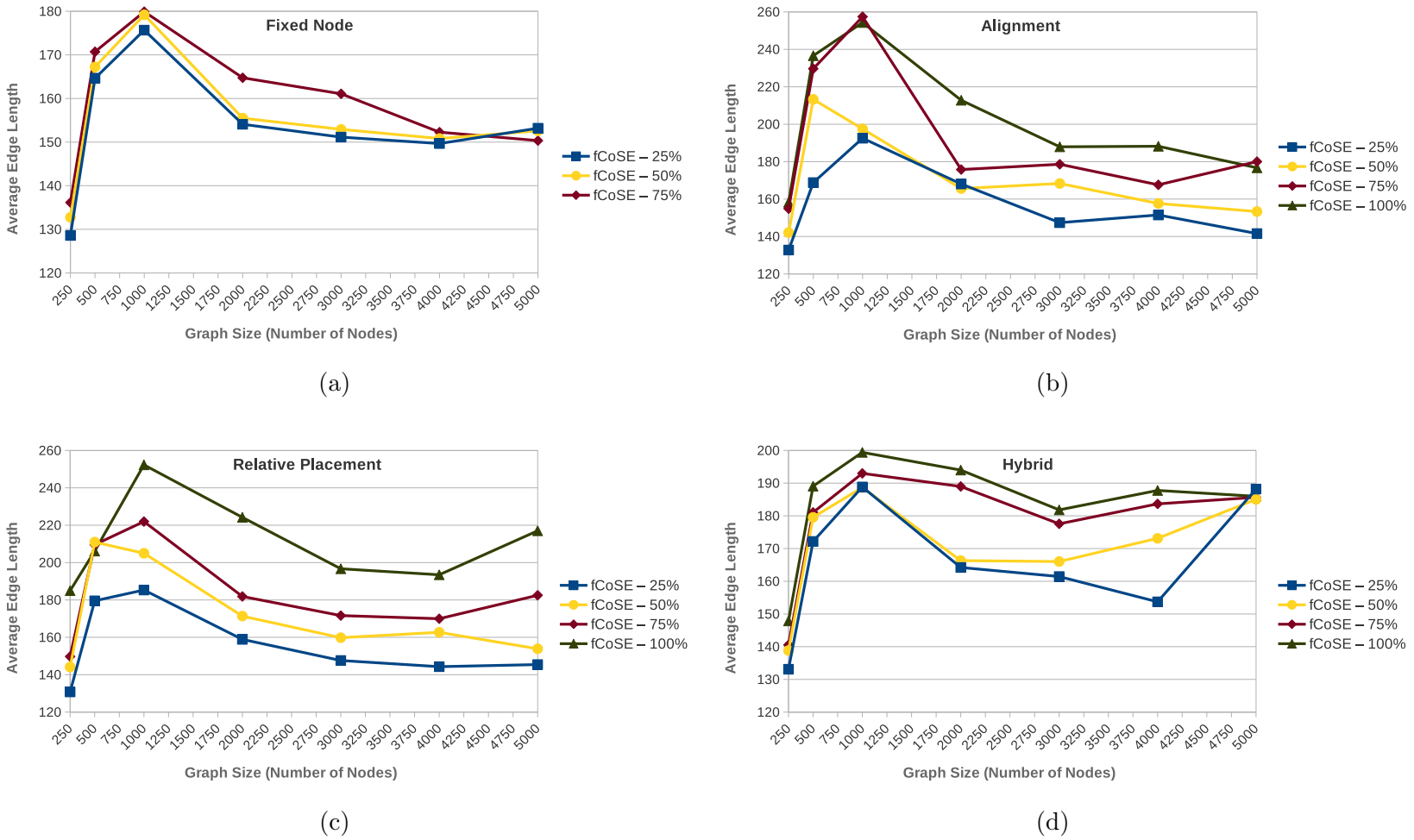


Figure 5.17: fCoSE results for *average edge length* metric in medium-sized graphs constrained with different ratios of (a) fixed node, (b) alignment, (c) relative placement and (d) hybrid constraints

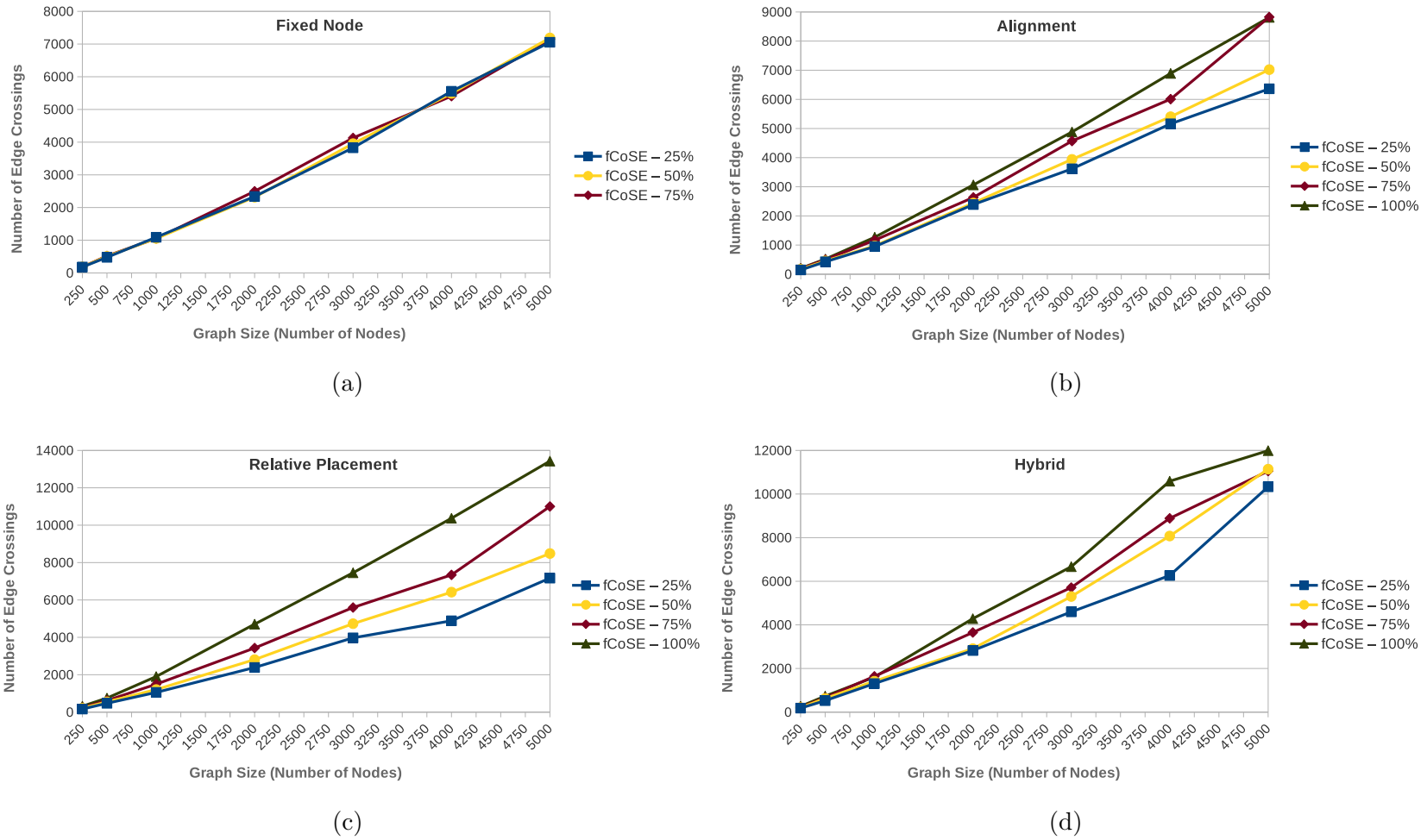


Figure 5.18: fCoSE results for *edge crossings* metric in medium-sized graphs constrained with different ratios of (a) fixed node, (b) alignment, (c) relative placement and (d) hybrid constraints

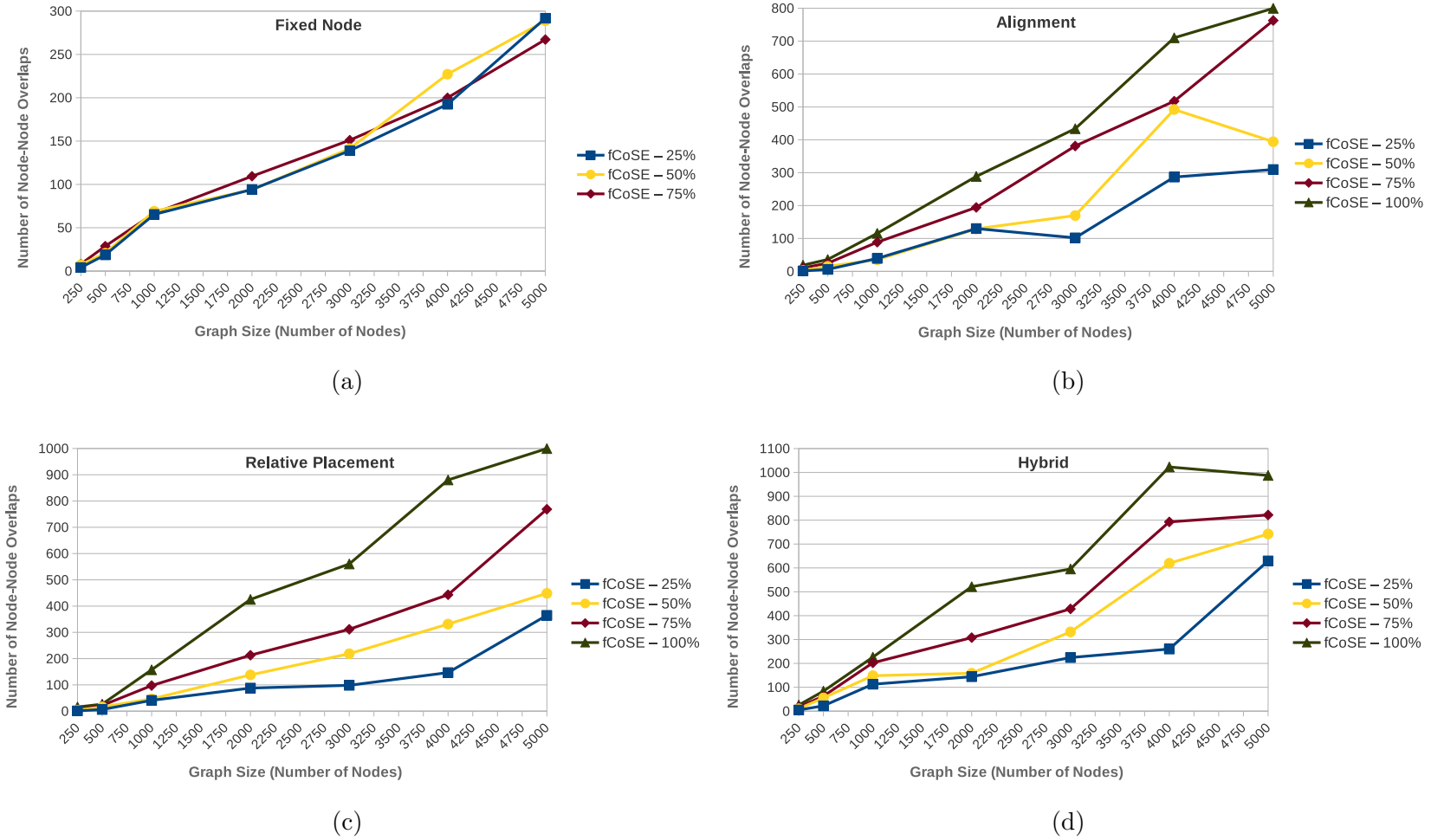


Figure 5.19: fCoSE results for *node-node overlaps* metric in medium-sized graphs constrained with different ratios of (a) fixed node, (b) alignment, (c) relative placement and (d) hybrid constraints

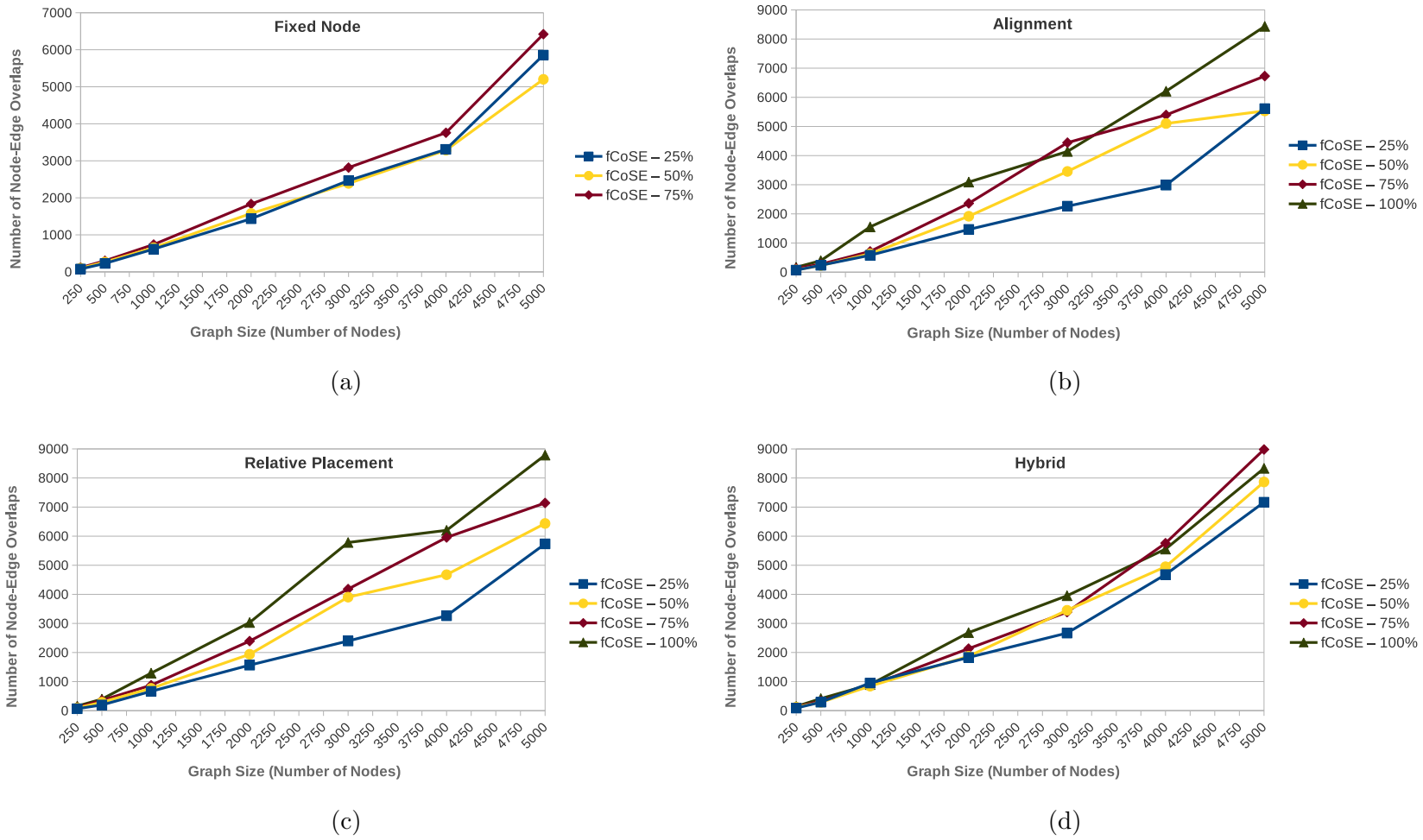


Figure 5.20: fCoSE results for *node-edge overlaps* metric in medium-sized graphs constrained with different ratios of (a) fixed node, (b) alignment, (c) relative placement and (d) hybrid constraints

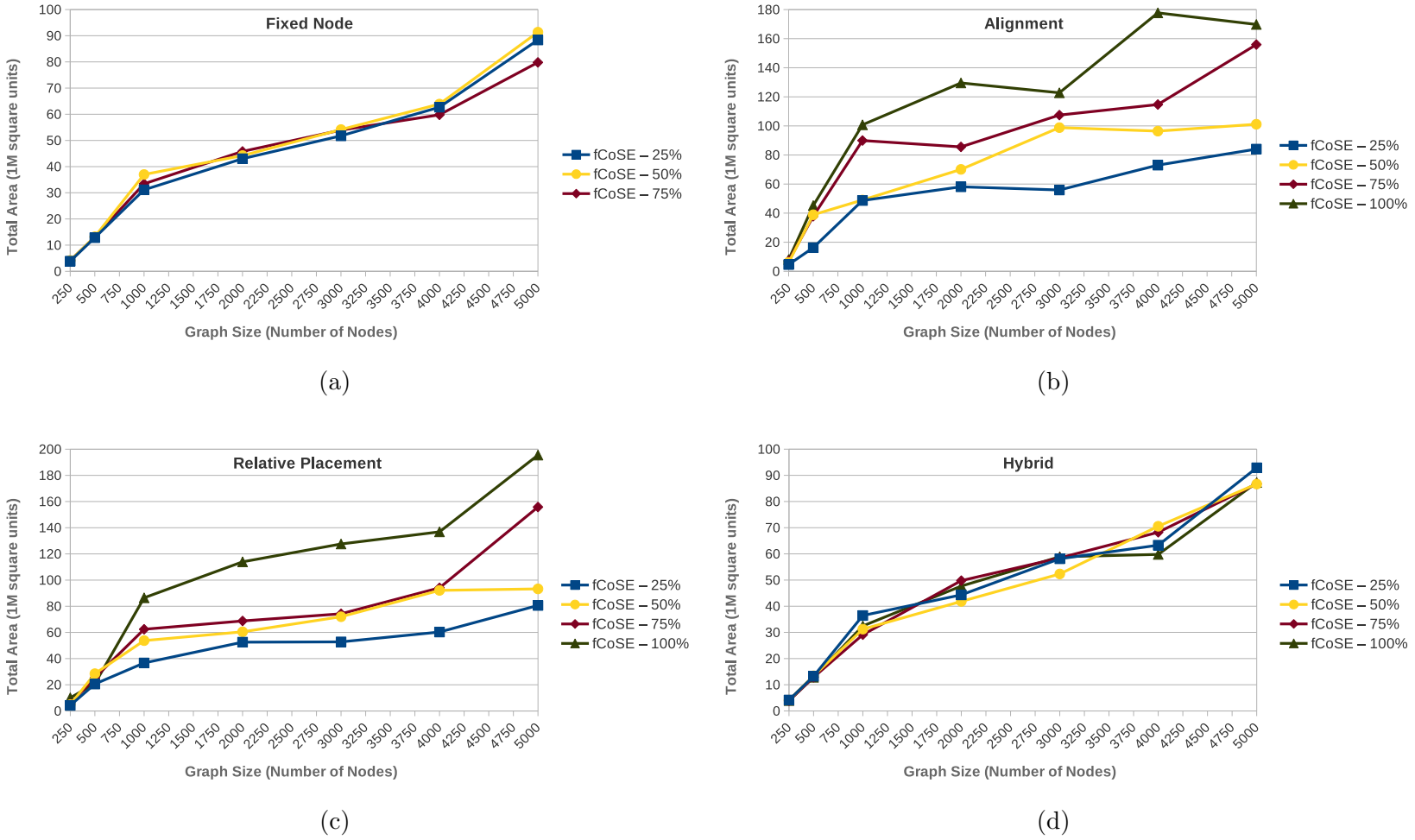


Figure 5.21: fCoSE results for *total area* (in 10^6 square units) metric in medium-sized graphs constrained with different ratios of (a) fixed node, (b) alignment, (c) relative placement and (d) hybrid constraints

Table 5.1: Evaluation with Denser Graphs

Hybrid Constraints 25%

graph name	$\|V\|$	$\|E\|$	$d(G)$	run time (sec)		avg. edge length		edge crossing		node-edge overlap		node-node overlap		total area (x1M)	
				fCoSE	CoLa	fCoSE	CoLa	fCoSE	CoLa	fCoSE	CoLa	fCoSE	CoLa	fCoSE	CoLa
aves-weaver-social-06	20	64	6.4	**0.04**	0.54	**56**	68	**57**	103	**15**	33	1	**0**	0.15	**0.11**
mammalia-dolphin-social	67	159	4.7	**0.06**	1.53	**80**	141	**209**	409	**80**	156	7	**2**	**0.65**	0.86
SW-100-6-0d2-trial1	106	300	5.7	**0.09**	1.76	**87**	151	**510**	1032	**231**	319	16	**3**	**0.92**	1.11
mammalia-voles-plj-trapping-25	166	497	6.0	**0.15**	21.57	**70**	126	**733**	1944	**220**	533	**9**	23	2.46	**2.07**
DD68_subgraph	209	454	4.3	**0.12**	27.87	**82**	158	**245**	1228	**119**	492	**5**	16	**4.03**	4.09
bio-desieasome	530	1188	4.5	**0.54**	-	**80**	-	**2444**	-	**948**	-	**45**	-	**8.00**	-
DD687	741	2599	7.0	**1.05**	-	**109**	-	**11790**	-	**4640**	-	**345**	-	**10.88**	-
DD242	1302	3305	5.1	**1.36**	-	**89**	-	**4509**	-	**2413**	-	**166**	-	**26.84**	-
lshp3446	3523	10215	5.8	**4.03**	-	**109**	-	**20156**	-	**8767**	-	**538**	-	**57.17**	-
DD6	4245	10319	4.9	**6.62**	-	**83**	-	**18214**	-	**10836**	-	**996**	-	**60.87**	-

Hybrid Constraints 50%

graph name	$\|V\|$	$\|E\|$	$d(G)$	run time (sec)		avg. edge length		edge crossing		node-edge overlap		node-node overlap		total area (x1M)	
				fCoSE	CoLa	fCoSE	CoLa	fCoSE	CoLa	fCoSE	CoLa	fCoSE	CoLa	fCoSE	CoLa
aves-weaver-social-06	20	64	6.4	**0.03**	0.37	**60**	96	**60**	123	**17**	39	1	**0**	**0.13**	0.18
mammalia-dolphin-social	67	159	4.7	**0.06**	1.24	**82**	136	**211**	468	**80**	172	**3**	6	**0.70**	0.73
SW-100-6-0d2-trial1	106	300	5.7	**0.09**	1.18	**86**	184	**516**	1470	**234**	445	21	**17**	**0.90**	1.2
mammalia-voles-plj-trapping-25	166	497	6.0	**0.13**	6.48	**72**	186	**773**	3179	**265**	787	**19**	58	**2.03**	2.51
DD68_subgraph	209	454	4.3	**0.11**	11.74	**85**	203	**272**	1738	**138**	550	**10**	20	4.29	**4.28**
bio-desieasome	530	1188	4.5	**0.58**	-	**82**	-	**2679**	-	**1051**	-	**58**	-	**6.64**	-
DD687	741	2599	7.0	**0.96**	-	**122**	-	**12752**	-	**4275**	-	**309**	-	**9.49**	-
DD242	1302	3305	5.1	**1.30**	-	**86**	-	**4330**	-	**2097**	-	**119**	-	**26.08**	-
lshp3446	3523	10215	5.8	**3.8**	-	**108**	-	**21797**	-	**9516**	-	**693**	-	**50.57**	-
DD6	4245	10319	4.9	**6.41**	-	**90**	-	**22144**	-	**12629**	-	**1402**	-	**65.06**	-

Hybrid Constraints 75%

graph name	$\|V\|$	$\|E\|$	$d(G)$	run time (sec)		avg. edge length		edge crossing		node-edge overlap		node-node overlap		total area (x1M)	
				fCoSE	CoLa	fCoSE	CoLa	fCoSE	CoLa	fCoSE	CoLa	fCoSE	CoLa	fCoSE	CoLa
aves-weaver-social-06	20	64	6.4	**0.03**	0.22	**60**	96	**63**	116	**22**	40	1	**0**	**0.12**	0.16
mammalia-dolphin-social	67	159	4.7	**0.06**	0.97	**89**	184	**250**	633	**105**	204	**7**	15	**0.73**	0.90
SW-100-6-0d2-trial1	106	300	5.7	**0.09**	0.87	**89**	226	**618**	1595	**268**	452	**22**	28	**0.83**	1.24
mammalia-voles-plj-trapping-25	166	497	6.0	**0.15**	11.77	**77**	185	**926**	3175	**309**	728	**22**	47	**2.11**	2.4
DD68_subgraph	209	454	4.3	**0.14**	10.26	**89**	220	**290**	2227	**181**	670	**20**	49	**4.08**	4.20
bio-desieasome	530	1188	4.5	**0.54**	-	**91**	-	**3268**	-	**1273**	-	**79**	-	**7.07**	-
DD687	741	2599	7.0	**0.98**	-	**143**	-	**15258**	-	**4997**	-	**416**	-	**16.28**	-
DD242	1302	3305	5.1	**1.31**	-	**96**	-	**5532**	-	**2785**	-	**263**	-	**25.48**	-
lshp3446	3523	10215	5.8	**3.53**	-	**120**	-	**26511**	-	**10294**	-	**890**	-	**51.28**	-
DD6	4245	10319	4.9	**6.10**	-	**99**	-	**26560**	-	**13745**	-	**1658**	-	**60.22**	-

Hybrid Constraints 100%

graph name	$\|V\|$	$\|E\|$	$d(G)$	run time (sec)		avg. edge length		edge crossing		node-edge overlap		node-node overlap		total area (x1M)	
				fCoSE	CoLa	fCoSE	CoLa	fCoSE	CoLa	fCoSE	CoLa	fCoSE	CoLa	fCoSE	CoLa
aves-weaver-social-06	20	64	6.4	**0.03**	0.12	**68**	123	**57**	150	**30**	56	3	**2**	**0.13**	0.17
mammalia-dolphin-social	67	159	4.7	**0.07**	0.84	**92**	189	**229**	477	**111**	179	15	**11**	**0.69**	0.87
SW-100-6-0d2-trial1	106	300	5.7	**0.10**	0.98	**98**	184	**739**	1555	**358**	480	40	**35**	**0.95**	1.22
mammalia-voles-plj-trapping-25	166	497	6.0	**0.13**	3.33	**83**	185	**960**	2933	**327**	685	**24**	27	**2.22**	2.47
DD68_subgraph	209	454	4.3	**0.13**	5.04	**101**	271	**359**	2870	**219**	754	**27**	53	**4.04**	4.12
bio-desieasome	530	1188	4.5	**0.55**	-	**103**	-	**4045**	-	**1492**	-	**118**	-	**6.79**	-
DD687	741	2599	7.0	**1.05**	-	**126**	-	**15312**	-	**5050**	-	**368**	-	**10.57**	-
DD242	1302	3305	5.1	**1.30**	-	**98**	-	**5889**	-	**2944**	-	**281**	-	**25.02**	-
lshp3446	3523	10215	5.8	**3.68**	-	**123**	-	**28044**	-	**10742**	-	**992**	-	**49.71**	-
DD6	4245	10319	4.9	**6.18**	-	**99**	-	**26084**	-	**13472**	-	**1479**	-	**60.87**	-

Graphs with $|V| > 500$ are evaluated with only fCoSE since Cola does not scale well to larger graphs.

5.4 Extensibility and Limitations

While fCoSE tries its best to support some soft constraints such as avoiding node-node overlaps or keeping child nodes inside the boundaries of their corresponding parent nodes, it has full support for the fixed node, alignment and relative placement constraints as mentioned in detail previously. The supported constraint types can be easily increased either by using the current constraint set or by adding support for new constraint types. For example, an orthogonal placement of the nodes can be satisfied by specifying a proper combination of alignment and relative placement constraints. A new constraint type can also be easily integrated as long as it is enforced in Phase II of the algorithm since maintaining it in the last phase is straightforward. For instance, the user can define an arbitrary region in the drawing canvas so that the layout of the graph should stay within the boundaries of that region. In that case, nodes are first enforced to be inside that region in Phase II and then they are allowed to move in a such way that they will not cross the boundaries of the region in the last iterative phase. Another possible improvement for the algorithm would be to allow separating nodes with an *exact* amount as opposed to a minimal value while defining relative placement constraints. Achieving it requires slight changes in Phase II where we place nodes to satisfy the separation amount and then we can treat the pair as a block during the movements in the last phase.

Since fCoSE is a combination of spectral and force-directed approaches, it suffers from the limitations of both, while the latter one is more dominant as it is applied last. Such limitations include not dealing with the edge-edge crossings or not trying to use the drawing area efficiently. On top of these inherited limitations, the algorithm's effort to support user-specified placement constraints makes obtaining a good layout more difficult, which is already an NP-hard problem [14]. An important limitation of fCoSE is to support user-defined constraints only on simple nodes, not on compound nodes. Adding constraint support for compound nodes is a challenging issue because a constraint in the positioning of a compound node may lead to violation of already satisfied constraints on its children nodes and vice versa.

Chapter 6

Conclusion and Future Work

In this thesis, we have presented a new, fast compound graph layout algorithm named fCoSE that also supports commonly used user-specified placement constraints in real-life graphs. fCoSE gets its speed from spectral layout approaches while presenting aesthetic results with the help of force-directed algorithms. It first produces a draft layout with the help of a spectral approach, then enforces placement constraints by using some heuristic methods and finally polishes the layout via a force-directed compound graph layout modified to maintain enforced constraints. Our experiments show that fCoSE is suitable to be used in interactive graph visualization and analysis applications that support small to medium-sized graphs, with its superiority over its competitors in both run time performance and commonly accepted aesthetic criteria.

As the future work, additional constraint types can be added such as containment constraint where some nodes are restricted to a specific region or a circle constraint where some nodes are positioned in such a way that they will form a circle. Furthermore, fCoSE currently allows constraints to be defined only on simple nodes. Adding constraint support for compound nodes can be considered in the future as such a feature will be useful in many application areas including visualization of biological networks. Moreover, fCoSE simply selects the simple node with the minimum degree while connecting disconnected components and

selecting representative nodes for compound nodes in the preprocessing step. A more deliberate approach can be used for these selection processes for a possible improvement in the quality of the draft layout. Lastly, a possible improvement in the quality of the layout with alignment constraints can be achieved by allowing aligned nodes to change the order in the polishing phase with swaps to further relax the underlying system.

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

Randomly generated compound graph dataset and test scripts can be found on *test-assets* branch of the same repository: https://github.com/iVis-at-Bilkent/cytoscape.js-fcose/tree/test-assets.

A demo page is availabe to try out the layout interactively: https://ivis-at-bilkent.github.io/cytoscape.js-fcose/demo/demo-constraint.html.

A series of videos showing fCoSE in action is available as a playlist in YouTube: https://youtube.com/playlist?list=PLJA9by$_{c}rwfaCK6gkAMIQ8OPp-k2NOi2H$.

Bibliography

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

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

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

[4] Dependency cruiser, Github repository, "Chalk dependency graph." `https://github.com/sverweij/dependency-cruiser/blob/develop/doc/real-world-samples/chalk.png`. (accessed: July 8, 2022).

[5] H. Kaur and A. P. Singh, "WSN localization in 3-d environments to minimize the localization errors," *International Journal of Science and Research (IJSR)*, vol. 4, no. 2, pp. 1362 – 1364, 2015.

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

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

[8] U. Dogrusoz, E. Giral, A. Cetintas, A. Civril, and E. Demir, "A layout algorithm for undirected compound graphs," *Information Sciences*, vol. 179, pp. 980–994, 2009.

[9] I. Borg and P. J. F. Groenen, *Modern Multidimensional Scaling: Theory and Applications*. Springer Science Business Media, 2005.

[10] "r/learnpython - Having lagging issues as well as some very odd behavior regarding a GUI, Threading, and Sockets." `https://www.reddit.com/r/learnpython/comments/cgtc4f/having_lagging_issues_as_well_as_some_very_odd/`. (accessed: Jan. 19, 2021).

[11] Graphviz - Graph Visualization Software, "Unix family 'tree'." `http://www.graphviz.org/Gallery/directed/unix.html`. (accessed: Jan. 13, 2021).

[12] E. Bobek and B. Tversky, "Creating visual explanations improves learning," *Cognitive research: principles and implications*, vol. 1, no. 1, pp. 1–14, 2016.

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

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

[15] U. Dogrusoz and B. Genc, "A multi-graph approach to complexity management in interactive graph visualization," *Computers & Graphics*, vol. 30, no. 1, p. 86–97, 2006.

[16] U. Dogrusoz, A. Karacelik, I. Safarli, H. Balci, L. Dervishi, and M. C. Siper, "Efficient methods and readily customizable libraries for managing complexity of large networks," *PLoS ONE*, vol. 13, no. 5, p. e0197238, 2018.

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

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

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

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

[21] U. Brandes and C. Pich, "Eigensolver methods for progressive multidimensional scaling of large data," in *Int. Symp. Graph Drawing*, pp. 42–53, Springer, 2006.

[22] Y. Koren, "On spectral graph drawing," in *International Computing and Combinatorics Conference*, pp. 496–508, Springer, 2003.

[23] W. S. Torgerson, "Multidimensional scaling: I. theory and method," *Psychometrika*, vol. 17, no. 4, pp. 401–419, 1952.

[24] A. Civril, M. Magdon-Ismail, and E. Bocek-Rivele, "Ssde: Fast graph drawing using sampled spectral distance embedding," in *Int. Symp. Graph Drawing*, pp. 30–41, Springer, 2006.

[25] V. De Silva and J. B. Tenenbaum, "Global versus local methods in nonlinear dimensionality reduction," *Advances in neural information processing systems*, pp. 721–728, 2003.

[26] M. Fink, *Crossings, Curves, and Constraints in Graph Drawing*. Würzburg University Press, 2014.

[27] A. Tikhonova and K.-L. Ma, "A scalable parallel force-directed graph layout algorithm," in *Proceedings of the 8th Eurographics conference on Parallel Graphics and Visualization*, pp. 25–32, 2008.

[28] V. Uher, P. Gajdo, and V. Snael, "The visualization of large graphs accelerated by the parallel nearest neighbors algorithm," in *2016 IEEE Second International Conference on Multimedia Big Data (BigMM)*, pp. 9–16, IEEE, 2016.

[29] P. Gajdoš, T. Ježowicz, V. Uher, and P. Dohnálek, "A parallel fruchterman–reingold algorithm optimized for fast visualization of large graphs and swarms of data," *Swarm and evolutionary computation*, vol. 26, pp. 56–63, 2016.

[30] C. Walshaw, "A multilevel algorithm for force-directed graph drawing," in *International Symposium on Graph Drawing*, pp. 171–182, Springer, 2000.

[31] D. Harel and Y. Koren, "A fast multi-scale method for drawing large graphs," in *International symposium on graph drawing*, pp. 183–196, Springer, 2000.

[32] F. G. Toosi and N. S. Nikolov, "Vertex-neighboring multilevel force-directed graph drawing," in *2016 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pp. 002996–003001, IEEE, 2016.

[33] 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.

[34] B. Genc and U. Dogrusoz, "An algorithm for automated layout of process description maps drawn in sbgn," *Bioinformatics*, vol. 32, no. 1, pp. 77–84, 2016.

[35] R. Tamassia, *Handbook of graph drawing and visualization*. CRC press, 2013.

[36] S.-H. Cheong and Y.-W. Si, "Force-directed algorithms for schematic drawings and placement: A survey," *Information Visualization*, vol. 19, no. 1, pp. 65–91, 2020.

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

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

[39] P. Eades, Q. Feng, X. Lin, and H. Nagamochi, "Straight-line drawing algorithms for hierarchical graphs and clustered graphs," *Algorithmica*, vol. 44, no. 1, pp. 1–32, 2006.

[40] F. Bertault and M. Miller, "An algorithm for drawing compound graphs," in *Int. Symp. Graph Drawing*, pp. 197–204, Springer, 1999.

[41] W. Didimo and F. Montecchiani, "Fast layout computation of clustered networks: Algorithmic advances and experimental analysis," *Information Sciences*, vol. 260, pp. 185–199, 2014.

[42] X. Wang and I. Miyamoto, "Generating customized layouts," in *Int. Symp. Graph Drawing*, pp. 504–515, Springer, 1995.

[43] Y. Frishman and A. Tal, "Dynamic drawing of clustered graphs," in *IEEE Symp. Information Visualization*, pp. 191–198, IEEE, 2004.

[44] P. Eades and M. L. Huang, "Navigating clustered graphs using force-directed methods," in *Graph Algorithms And Applications 2*, pp. 191–215, World Scientific, 2004.

[45] K. M. Hall, "An r-dimensional quadratic placement algorithm," *Management science*, vol. 17, no. 3, pp. 219–229, 1970.

[46] Y. Koren, L. Carmel, and D. Harel, "Ace: A fast multiscale eigenvectors computation for drawing huge graphs," in *IEEE Symposium on Information Visualization, 2002. INFOVIS 2002.*, pp. 137–144, IEEE, 2002.

[47] D. Harel and Y. Koren, "Graph drawing by high-dimensional embedding," in *International symposium on graph drawing*, pp. 207–219, Springer, 2002.

[48] Y. Wang, Y. Wang, Y. Sun, L. Zhu, K. Lu, C.-W. Fu, M. Sedlmair, O. Deussen, and B. Chen, "Revisiting stress majorization as a unified framework for interactive constrained graph visualization," *IEEE Trans. Vis. Comput. Graph.*, vol. 24, no. 1, pp. 489–499, 2017.

[49] S. Devkota, R. Ahmed, F. De Luca, K. E. Isaacs, and S. Kobourov, "Stress-Plus-X (SPX) Graph Layout," in *Int. Symp. on Graph Drawing and Network Visualization*, pp. 291–304, Springer, 2019.

[50] R. Ahmed, F. De Luca, S. Devkota, S. Kobourov, and M. Li, "Graph drawing via gradient descent, $(GD)^2$," in *Int. Symp. on Graph Drawing and Network Visualization*, pp. 3–17, Springer, 2020.

[51] K.-F. Böhringer and F. N. Paulisch, "Using constraints to achieve stability in automatic graph layout algorithms," in *Proc. SIGCHI Conf. Human Factors in Computing Systems*, pp. 43–51, 1990.

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

[53] W. He and K. Marriott, "Constrained graph layout," *Constraints*, vol. 3, no. 4, pp. 289–314, 1998.

[54] K. Ryall, J. Marks, and S. Shieber, "An interactive constraint-based system for drawing graphs," in *Proc. 10th Annual ACM Symp. User Interface Soft. and Technol.*, pp. 97–104, 1997.

[55] W. Didimo, G. Liotta, and F. Montecchiani, "Network visualization for financial crime detection," *Journal of Visual Languages & Computing*, vol. 25, no. 4, pp. 433–451, 2014.

[56] T. Dwyer, Y. Koren, and K. Marriott, "IPSep-CoLa: An incremental procedure for separation constraint layout of graphs," *IEEE Trans. Vis. Comput. Graph.*, vol. 12, no. 5, pp. 821–828, 2006.

[57] T. Dwyer, K. Marriott, and M. Wybrow, "Topology preserving constrained graph layout," in *Int. Symp. Graph Drawing*, pp. 230–241, Springer, 2008.

[58] T. Dwyer, "Scalable, versatile and simple constrained graph layout," in *Computer Graphics Forum*, vol. 28, pp. 991–998, Wiley Online Library, 2009.

[59] R. Sedgewick and K. Wayne, *Algorithms*, pp. 661–667. Addison-wesley professional, fourth ed., 2011.

[60] 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.

[61] C. Ware and R. Bobrow, "Supporting visual queries on medium-sized node–link diagrams," *Information Visualization*, vol. 4, no. 1, pp. 49–58, 2005.

[62] M. Ventresca and D. Aleman, "Efficiently identifying critical nodes in large complex networks," *Computational Social Networks*, vol. 2, no. 1, p. 6, 2015.

[63] 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.

[64] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

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