fCoSE: a fast compound graph layout algorithm with constraint support

Hasan Balci, and Ugur Dogrusoz, Senior Member, IEEE

Abstract—Visual analysis of relational information is vital in most real-life analytics applications. Automatic layout is a key requirement for effective visual display of such information. This paper introduces a new layout algorithm named fCoSE for compound graphs showing varying levels of groupings or abstractions with support for user-specified placement constraints. fCoSE builds on a previous compound spring embedder layout algorithm and makes use of the spectral graph drawing technique for producing a quick draft layout, followed by phases where constraints are enforced and compound structures are properly shown while polishing the layout with respect to commonly accepted graph layout criteria. Experimental evaluation verifies that fCoSE produces quality layouts and is fast enough for interactive applications with small to medium-sized graphs by combining the speed of spectral graph drawing technique with the quality of force-directed layout algorithms while satisfying specified constraints and properly displaying compound structures. An implementation of fCoSE along with documentation and a demo page is freely available on [GitHub].

Index Terms—Information visualization, graph layout, visual analytics, compound graphs, constrained layout, spectral graph drawing.

1 INTRODUCTION

Nowadays, data is being accumulated at an extraordinary speed. Analyzing such data, including relational ones, is an important prerequisite to making informed decisions in all kinds of businesses. Making use of visualization makes analysis easier for human beings as it brings out broad relationships, patterns, and emerging trends, providing deeper insight.

A commonly used visual representation of relational data is graphs or networks. When visualizing relational information via graphs, a good layout of objects and links is crucial since a poor one will confuse the user and a typical user will spend up to 25% of their time on manual layout adjustments [1]. Hence, a good automatic layout operation is an indispensable part of graph visualization-based analysis software.

Normally, a layout algorithm is completely free in placing nodes and routing edges to optimize metrics such as number of edge-edge crossings, total area, and maximal display of symmetries. However, oftentimes, an application will have some domain-specific constraints on the placement of individual nodes or require alignment or relative placement of a group of nodes (Fig. 1). Such constraints are input to the particular layout algorithm along with the topology of the graph.

The notion of compound graphs [4] has been in wide use to represent complex relationships or varying levels of abstractions in data. Even though the automatic layout of simple graphs without constraints is a well-studied problem [5], work on compound graphs or graphs with specified constraints has been very limited [4] [6] [7].

This paper introduces fCoSE (a fast Compound Spring Embedder), a new automatic layout algorithm for compound graphs with support for a fairly rich set of constraint types. fCoSE combines the best of two worlds: speed of spectral drawing techniques [8] and quality of force-directed layout [9] algorithms, while properly addressing input constraints, exhibiting compound structures, and accounting for non-uniform node dimensions.

Experimental results comply with the theoretical analysis of the run time efficiency of fCoSE, achieving an average of 2x speedup over CoSE [4] and even more over CoLa [6], fast enough for small to medium-sized graphs for use in interactive graph visualization components. fCoSE meets the expectations in terms of quality metrics as well, comparing fairly well with those of previous algorithms.

2 BASICS

A graph (aka network) is a commonly used representation for a discrete set of objects, called nodes, related to each other.
through links, called edges. Basics of graphs can be found in the supplementary material.

A compound graph $G = (V, E, F)$ consists of a set of nodes $V$, a set of (adjacency) edges $E$, and a set of inclusion edges $F$ [10]. The inclusion graph $T = (V, F)$ is a rooted tree, defined on nodes $V$ and inclusion edges $F$, where no adjacency edge is allowed to connect a node to one of its descendants or ancestors (Fig. 2). For $(u, v) \in F$, we say $u$ is a parent node of $v$, and $v$ is in the child graph or subgraph nested within compound node $u$. A dedicated subgraph that contains all nodes $u$ such that $(u, v) \in F$ and $u$ is not in any child graph is called the root graph. We call adjacency edges $(u, v) \in E$ intra-graph when both $u$ and $v$ are in the root graph or the same child graph, inter-graph when end nodes belong to different subgraphs. Compound graphs are used in representing varying levels of groupings or abstractions in data with a particular use in complexity management through expand-collapse operations [11]. Notice that a compound graph differs from its variants like hierarchically nested or ancestors (Fig. 2). For nodes $u$, $v$ $(u, v) \in F$ and $u$ is not in any child graph is called the root graph. We call adjacency edges $(u, v) \in E$ intra-graph when both $u$ and $v$ are in the root graph or the same child graph, inter-graph when end nodes belong to different subgraphs. Compound graphs are used in representing varying levels of groupings or abstractions in data with a particular use in complexity management through expand-collapse operations [11]. Notice that a compound graph differs from its variants like hierarchically nested or ancestors (Fig. 2).

Fig. 2. An example compound graph $G = (V = \{a, b, d, e, c_1, c_2\}, E = \{(a, b), (b, d), (d, e)\}, F = \{(c_1, a), (c_1, b), (c_1, d), (c_2, e), (c_1, c_2)\})$, containing two compound nodes $c_1$ and $c_2$ and a single inter-graph edge $(d, e)$ and its inclusion tree

A drawing of a graph is a function that maps each node to a distinct point in 2D space and each edge to a Jordan arc in the same space, with endpoints corresponding to the locations of the respective end nodes of the edge [5]. Within the scope of this work, we assume nodes to have a rectangular geometry. It is widely accepted that a good layout algorithm takes possibly non-uniform dimensions of the nodes into account in producing a layout, where node-node overlaps are avoided, edge-edge crossings and the total area is minimized, and edge lengths are as uniform as possible. It is eminent that a child node is drawn within its parent compound node’s bounds. When a user is relatively happy with a previously established layout of the graph but wants to “tidy it up” for any incremental changes, an incremental layout is applied with the purpose to maintain the user’s “mental map”.

Perhaps the most popular approach to automatic graph layout is the so-called force-directed approach [12], where the main idea is to simulate a system under the laws of physics. The CoSE algorithm [4] extends the force-directed model of Fruchterman & Reingold [13] to support compound nodes. Spectral layout algorithms are linear algebraic approaches to graph layout that are based on spectral decomposition of the graph-related matrices [8]. These algorithms generally use Laplacian or graph-theoretical distance matrices [14]. Of particular interest is the Classical Multidimensional Scaling (CMDS) method [15]. Work by Civril et al. [16] suggest a sampling-based approximation approach to reduce the time complexity to linear time in the number of nodes and edges. Another approach to fast graph layout is the so-called multilevel force-directed technique, where the graph is recursively clustered until a trivial one is obtained. Then, starting with the coarsest graph an incremental layout is calculated and extended to other graphs until the original graph is obtained [17]. Although multilevel algorithms are not as fast as the spectral layout algorithms, they produce better quality layouts. Further details of these graph layout approaches may be found in the supplementary material.

3 Related Work

There has been limited work done on compound graph layout with varying weakness accompanying poor run time complexity. Most such work [15] [16] focus on directed hierarchical graphs with compound structures, where edge directions enforce a hierarchy, generally failing to produce good quality layouts on undirected graphs. Work specifically done on undirected graphs [21] [22] use a top-down or bottom-up approach on the inclusion tree to lay out compound graphs yielding long inter-graph edges, while some others either support only one level of nesting [23] [24] or do not allow edges to directly connect to compound nodes [25].

The CoSE algorithm [4], however, provides full-support for compound structures, but with a mediocre run time performance. Even though its inherent cubic run time complexity $O(k \cdot |V|^2 + |E|)$ can be reduced to $O(k \cdot |V| + |E|)$ by using the grid variant method in [13], where $k$ is the number of iterations estimated to be $O(|V|)$, this is still not satisfactory for interactive use except for small graphs.

Spectral layout algorithms, on the other hand, are well known for their speed but do not support compound structures and often fail to produce refined drawings [6] [16]. For instance, two nodes with the same graph-theoretical distance to all remaining nodes will be positioned at the same location. Furthermore, these techniques do not support non-uniform node dimensions, which is widely used in real-life drawings, often yielding node-node overlaps.

Most layout algorithms including CoSE aim for good quality in one or more of the general graph drawing criteria (i.e., soft constraints) through different types of heuristics employed with varying success. Some recent studies such as [7] [26] [27] explicitly target and strive to optimize some of such criteria. The integration of user-specified hard constraints (or simply constraints) to automatic graph layout, where the constraints are expected to be fully satisfied unless conflicting, was first introduced in [28], where Böhlinger & Paulisch modified the layered drawing algorithm of Sugiyama et al. [29] to support some constraints on the positioning of the nodes. He & Marrriot [30] extend Kamada-Kawai stress model [31] to support separation constraints by using quadratic programming techniques; however, due to inefficient solvers, their algorithm does not scale well to larger graphs. Ryall et al. [32], Wang & Miyamoto [23] and Didimo et al. [33] modify the force-directed model to move nodes towards locations satisfying the constraints.

Maybe the most popular constrained layout algorithm in the literature is CoLa, a result of a series of studies [6] [34] [35]. The main idea is to use a gradient-projection algorithm, where nodes are first moved based on a descent vector, and
then constrained nodes are projected to satisfy constraints in each iteration of either stress majorization or force-directed model. Even though CoLa supports a wide range of constraints, it deeply suffers from a high computational cost. Besides, its support for compound structures along with an option to avoid node-node overlaps via constraints requires a quadratic number of constraints to be defined on the nodes, further increasing its computational cost, making it impractical for graphs with more than a few hundred nodes. A recent study by Wang et al. [7] reformulates the mathematical model of stress majorization to support a wider range of constraints than CoLa supports, obtaining a faster implementation by using the GPU. However, their algorithm has comparable efficiency with CoLa with a CPU implementation.

In summary, there is a need for a layout algorithm that supports compound graphs, non-uniform node dimensions, and user-specified constraints (on top of the soft ones addressed by fCoSE’s base method) simultaneously, running fast enough for small to medium-sized graphs.

### 4 Algorithm

Motivated by the lack of a fast constrained layout algorithm for compound graphs as set forth earlier, we now introduce fCoSE that supports three generic constraint types commonly used for the layout of real-life graphs:

- **Fixed node constraint**: The user may provide exact desired (aka anchor) positions for a set of nodes called fixed nodes. We denote a node \( a \) with a fixed node constraint at a location \((x, y)\) as “\(a\mid x, y\)”. The algorithm is to produce a layout with the fixed node \( a \) located exactly at \((x, y)\).

- **Alignment constraint**: This constraint aims to align the centers of two or more nodes vertically or horizontally. We denote nodes \( a, b, c \) aligned horizontally as “\(a \sim b \sim c\)”. Similarly, when the same nodes are vertically aligned, we use “\(a \mid b \mid c\).” There can be an arbitrary number of alignment constraints in each direction, and a node can be a part of both a vertical and a horizontal alignment constraint.

#### Algorithm 1 The fCoSE Algorithm

```plaintext
function RUNLAYOUT(G, C^f, C^n, C^r)

APPLYSPECTRAL(G)

if \(|C^f| > 1\) then \(\triangleright\) use fixed nodes

\(xformMatrix \leftarrow \text{CALCXFORMFIXED}(G, C^f)\)

APPLYFORM(G, xformMatrix)

else if \(|C^f| \leq 1\) and \(|C^n| > 0\) then \(\triangleright\) use alignment

\(xformMatrix \leftarrow \text{CALCXFORMALIGNMENT}(G, C^n)\)

APPLYFORM(G, xformMatrix)

if \(|C^r| > 0\) then

\(\text{APPLYMAJORITYREFLECTION}(G, C^r)\)

else if \(|C^f| \leq 1\) and \(|C^n| = 0\) and \(|C^r| > 0\) then \(\triangleright\) use relative placement

construct dags \(D^h\) and \(D^v\) from \(C^r\)

\(D \leftarrow D^h \cup D^v\)

\(C_i = (V_i, E_i) \leftarrow \text{largest component in } D\)

if \(|V_i| < |V(D)/2|\) then

\(\text{APPLYMAJORITYREFLECTION}(G, C^r)\)

else

\(xformMatrix \leftarrow \text{CALCXFORMRELATIVE}(G, C_i)\)

APPLYFORM(G, xformMatrix)

if \(|C^f| > 0\) then

\(\text{ENFORCECONSTRAINTSFIXED}(G, C^f)\)

if \(|C^n| > 0\) then

\(\text{ENFORCECONSTRAINTSALIGNMENT}(G, C^f, C^n)\)

if \(|C^r| > 0\) then

\(\text{ENFORCECONSTRAINTSRELATIVE}(G, C^f, C^n, C^r)\)

\(\text{totIter} \leftarrow 0\)

while \(\text{totIter} < \text{maxIter} \, \text{or} \, !\text{CONVERGED()}\) do

\(\text{totIter} \leftarrow \text{totIter} + 1\)

\(\text{UPDATEBOUNDS()}\)

\(\text{CALCForces()}\)

\(\text{CALCDisplacements()}\)

if \(|C^f| \cap C^n \cap C^r| > 0\) then

\(\text{ADJUSTDisplacements()}\)

\(\text{MOVEnodes()}\)
```

Fig. 3. Algorithm overview. Given a compound graph, a draft layout is obtained in Phase I with a spectral layout algorithm. Phase II then satisfies constraints on this draft layout. Finally, Phase III polishes the constrained draft layout with a modified CoSE algorithm to produce a final layout.

Fig. 4. A sample compound graph with constraints \(n2|[-50, 100], n5|[50, -50]\) and \(n4 - n6\) after (a) draft layout (b) transformed draft layout based on fixed node constraints (c) constrained draft layout (d) final layout (the constrained nodes are shown in red; anchors signify fixed node constraints).
Fig. 5. (a) A disconnected compound graph (b) Components (inside red rounded rectangles) in the root graph are tied via dummy node d1. (c) Components in the child graph of compound node c2 are tied via dummy node d0 as well, whereas child graphs of c0 and c1 are already connected. (d) The modified compound graph is converted into a simple one. Here, nodes with red borders (n0, n3 and n5) are selected simple nodes to represent their parent compound nodes c0, c1 and c2, respectively, and the red edges are the ones previously incident upon these parent nodes.

We assume, however, that when a node is involved in an alignment constraint in a certain direction, all its aligned nodes are gathered into and expressed as a single constraint (e.g., “a – b – c” as opposed to “a – b” and “b – c” as two separate constraints). Note that both of these relations are transitive and reflexive.

- Relative placement constraint: The user may constrain the position of a node relative to another node in either vertical or horizontal direction in the form of “node a will be to the left of (above) node b by at least x > 0 units” denoted as “a <x b” (“a ∧x b”). When x is not specified, we assume a default minimum separation amount between involved nodes. Note that both of these relations are transitive and reflexive. Clearly, the use of “right of” and “below” are redundant.

We assume the user does not specify any conflicting constraints (e.g., a < b and b < a). We also note that constraints can only include simple nodes and a node can get involved in more than one constraint of possibly different types.

The fCoSE algorithm running on a compound graph \( G = (V, E, F) \) with constraints \( C = C^f \cup C^n \cup C^r \) (a union of fixed node, alignment, and relative placement constraints, respectively) consists of three main phases (Fig. 3 and 4). In the first phase, we convert the possibly disconnected input compound graph into a connected simple one and apply a spectral layout algorithm [16] on it to obtain a draft layout. The second phase is aimed at enforcing user-specified placement constraints. Before that, however, we apply a transformation to make the drawing more compatible with the specified constraints, so the enforcement step will minimally disrupt the draft layout obtained. The last phase can be considered as a “polishing phase”, where we apply a modified version of the CoSE method [4] to respect nonuniform node dimensions and compound structures and eliminate node-node overlaps while preserving enforced constraints. The overall structure of the algorithm can be summarized as in Algorithm 1.

4.1 Phase I: Obtaining Draft Layout

As spectral layout algorithms are based on graph-theoretical distances of nodes, they cannot be directly applied to disconnected or compound graphs. Hence, we apply a preprocessing step to convert the input compound graph into a simple and connected graph. Then, a spectral layout algorithm is applied, followed by a postprocessing step that restores the topology of the original compound graph resulting in a draft layout.

4.1.1 Preprocessing Step

To handle disconnected graphs, we use “dummy nodes” to tie together components. In finding components of a compound graph, we use a specialized breadth-first search (BFS) which assumes that upon reaching a parent compound node, all nodes in its nested child graph are also reached via the traversal and vice versa. For example, in Fig. 5a, although c1 is not directly adjacent with n2 or c0, we consider them as “neighbors” and continue a traversal reaching c1, towards both n2 and c0 as well as some other nodes. Notice here that n2 is adjacent with a child node (n3) of c1, and c0 is a parent of a node (n1) that is adjacent with c1. To tie components of a disconnected graph, we select a node with a minimum degree from each component and connect it to a dedicated dummy node so as to keep the node degrees as homogeneous as possible (Fig. 5d). This is done not only for the root graph but also for each child graph as the graph might become disconnected once we remove compound structures as described below (Fig. 5c).

To be able to convert a compound graph into a simple one, for each compound node, we assign the mission of a compound node to a selected simple node inside that compound node and remove the compound node temporarily. This selected simple node represents the compound node and the intra-graph edges connected to the compound node are now connected to this representative node. Again, we choose a node with a minimum degree to keep the degrees of the nodes homogeneous after conversion (Fig. 5d). As a result, we have a connected, simple graph on which a spectral layout may be performed.

4.1.2 Applying Spectral Layout Algorithm

To obtain a draft layout from the simple and connected graph constructed earlier, we apply a linear time CMDS method [16] mentioned earlier.

4.1.3 Postprocessing Step

Construction of a decent draft layout in Phase I ends with a post-processing step. We remove any dummy nodes introduced earlier and position compound nodes based on their children so as to tightly contain them.

4.2 Phase II: Satisfying Constraints

In this phase, we start with a draft layout that is constraint-free and first apply a transformation on the draft layout by performing rotation and/or reflection. Here the goal
Fig. 6. (a) Draft layout with two fixed node constraints \( n0 \)\(|\{−150, 50\} \) and \( n6 \)\(|\{150, −50\} \) (b) Constrained draft layout calculated without a transformation step and (c) with a transformation step that rotates draft layout by \( 163.5^\circ \) clockwise

Fig. 7. (a) Draft layout together with target configuration formed by user-specified positions on the fixed nodes \( n0, n1 \) and \( n2 \) (b) Transformed (reflects draft layout on the \( y \)-axis and then rotates by \( 6^\circ \) counterclockwise) draft layout

is to better align the current layout with constraints, as enforcing constraints on a layout that is incompatible with our constraints could completely ruin the draft layout with respect to commonly accepted criteria such as minimal overlaps and edge-edge crossings. We then process the constraints and obtain a layout satisfying these constraints (i.e., a constrained draft layout).

4.2.1 Transformation of Draft Layout

The transformation step aims to adjust the orientation of the graph to be more compatible with the user-specified placement constraints. Directly enforcing the constraints on the draft layout may cause drastic changes in node positions, resulting in longer edges and more edge-edge crossing, and eventually reducing the overall quality of the final layout. With a transformation based on constraints, however, the movement of the constrained nodes in succeeding steps is minimized and the overall structure of the draft layout is protected as much as possible. Fig. 8 exemplifies the use of the transformation step. Notice how the transformation step helps in producing a more stable force-directed system, closer to convergence.

To calculate a suitable transformation, we make use of a solution to the famous orthogonal Procrustes problem (Chapter 20 of [15]), where the goal is to map a source configuration of a set of points to a target configuration. The solution used is a linear algebraic one, calculating an orthogonal matrix that most closely maps a source configuration to a target one by restricting the transformation to rotations and reflections. This type of transformation is exactly what we need since we only want to change the orientation of the layout while preserving its overall structure.

This is achieved 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 \( A^TB \). Then, the orthogonal transformation matrix can be calculated with \( T = QP^T \). For our purposes, we first need to decide on the nodes to use in the calculation of the transformation matrix before we can apply the resulting matrix to the whole graph. Remember that we would like to adjust the orientation of the graph according to the user-specified constraints. Here we assume that the constraints are not in utmost conflict with each other, and hence we may use a subset of the constraints as we see fit. Obviously, the source configuration of the selected nodes comes from the draft layout calculated by the spectral layout algorithm. However, the construction of the target configuration is rather involved and depends on the selected nodes of the chosen constraint type(s) as detailed in the rest of the section.

As fixed node constraints are the most strict of all, if \( |C| > 1 \), then we base the transformation on these nodes only. In this case, the target configuration is formed directly from the positions specified by the user for these fixed nodes. Fig. 7 illustrates a sample scenario with three fixed nodes. Note that the orientation of the drawing is now more compatible with the fixed node constraints.

If \( |C| \leq 1 \), insufficient to define a target configuration, and \( |C| > 0 \), then we use all nodes involved in alignment constraints. In this case, for each alignment constraint, the target configuration of all involved nodes is formed by taking their average position in the respective direction (remember that a node may be involved in at most one alignment constraint in each direction). Fig. 8 explains the use of two alignment constraints (one in each direction) for constructing a target configuration to be used for the transformation. Here, one can see that the transformation reduces the total amount of node movement required to enforce the alignment constraints in the next step, compared to the case where no transformation takes place (simply compare the total lengths of dark line segments with arrows in Fig. 8 and 9).

Now that the drawing was aligned with respect to the available alignment constraints, we might be able to make the drawing more compatible using the relative placement constraints as well. Hence, if \( |C'| > 0 \), we further apply a majority-based reflection on the graph based on the relative placement constraints. To do so, we evaluate the relative placement constraints defined along the \( x \)-axis (\( y \)-axis) and
This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TVCG.2021.3095303, IEEE Transactions on Visualization and Computer Graphics

4.2.2 Enforcing Constraints

Phase II is for enforcing the constraints on the draft layout constructed by Phase I. We process the constraints in the order of: fixed node, alignment and relative placement constraints with a goal to obtain a layout that satisfies all constraints. Then, the final phase (Phase III) will apply a force-directed incremental layout algorithm to improve/ refine the

Fig. 8. (a) A draft layout with constraints \( n_0 \mid n_1 \mid n_2 \) and \( n_3 \rightarrow n_4 \) (b) Amount of movement needed for the nodes in specified constraints to attain alignment and for determining the aligned target positions used in calculating the transformation matrix (c) Transformed (rotates draft layout by 45° clockwise) draft layout based on the alignment constraints (d) Transformed draft layout based on both alignment and relative placement constraints (\( n_1 \prec n_3, n_2 \prec n_4, n_0 \wedge n_1, \) and \( n_2 \wedge n_4 \))

Fig. 9. (a) A draft layout for a graph with constraints \( \{ n_2 \wedge [80], n_1, n_7 \wedge [70], n_6, n_4 \prec [100], n_0, n_7 < [70], n_4, n_8 \prec [80], n_4 \} \) (b) Dependency graph \( D \) formed by the nodes involved in relative placement constraints with the right one of the two components being larger. Solid edges show constraints in horizontal direction whereas dashed ones are for those in the vertical direction. The value on the edge shows the edge weight, while the value near a node shows its longest distance from a source node. (c) The target configuration formed by placing nodes by using the longest distance from their source node (d) Corresponding transformed (rotates draft layout by 180°) draft layout

if current positions of the involved nodes violate the majority of these constraints, we reflect the graph on the \( y \)-axis (\( x \)-axis).

As an example, assume that the graph in Fig. 8a also has the following relative placement constraints: \( \{ n_1 \prec n_3, n_2 \prec n_4, n_0 \wedge n_1, n_2 \wedge n_4 \} \). When the node positions on the current transformed layout (Fig. 8c) are inspected for the relative placement constraints given above, both constraints defined along the \( x \)-axis are violated. For constraints along the \( y \)-axis, one is satisfied while the other is not. Hence, we reflect the graph only on the \( y \)-axis (Fig. 8d), resulting in a draft layout that now violates only one of the relative placement constraints as opposed to the three we had before. Notice that this reflection step improves the orientation of the graph using the relative placement constraints while preserving the effect of the earlier transformation solely based on the alignment constraints.

If \( |C^f| \leq 1 \) and \( |C^n| = 0 \), we base the transformation only on the relative placement constraints. For this purpose, we form dependency dags \( D^h = (V^h, E^h) \) and \( D^v = (V^v, E^v) \), one for each direction. Here,

\[
V^h = \{ v \mid (u < v) \in C^r \lor (v < u) \in C^r \} \quad \text{and} \quad E^h = \{ e = (u, v) \wedge e.w = x \mid (u < [x] v) \in C^r \},
\]

where \( e.w \) represents the weight of the edge \( e \). \( D^v \) can be constructed similarly. The directed dependency graph, composed of both dags, not necessarily a dag itself, is defined as \( D = D^h \cup D^v \). Notice here that \( V^h \cap V^v \) is not necessarily empty but \( E^h \cap E^v = \emptyset \).

Assume \( C_i = (V_i, E_i) \) is the largest (weakly connected) component of \( D \). If \( |V_i| < |V(D)|/2 \), we simply apply a majority-based reflection on the draft layout as explained before to generate the transformed draft layout. Otherwise, when the largest component \( C_i \) is big enough, we base the transformation on vertices of \( V_i \) as follows. Suppose \( V_i = V_i^h \cup V_i^v \), where \( V_i^h \subseteq D^h \) and \( V_i^v \subseteq D^v \) respectively correspond to those vertices involved in horizontal and vertical relative placement constraints in the component. Let \( D_i^h = D_i^h[V_i^h] \) and \( D_i^v = D_i^v[V_i^v] \), both of which are dags as they are subgraphs of dags. For each of these dags, nodes with in-degree zero could be identified and taken as source nodes followed by a topological order based computation of longest distances from these sources. We then place the nodes in \( V_i^h (V_i^v) \) to appropriate \( x (y) \) coordinates taking the average \( x (y) \) coordinate of source nodes as a base and using the longest distance of each node from the source nodes in the \( x (y) \)-axis. As a result, we have a configuration based on the largest component of the dependency graph to be used as the target configuration for the transformation as exemplified in Fig. [9] Notice that the constrained node pairs are on the correct side of each other as dictated by the constraints after the transformation (Fig. [9d]), making jobs of later phases easier.
Relative placement constraints

For enforcing relative placement constraints, we form a directed dependency graph \( D = D^h \cup D^v \) from the node pairs involved in relative placement constraints, similar to the one formed during the transformation step (Fig. 9b). However, processing of the dependency dags \( D^h \) and \( D^v \) are done in order and the process is more sophisticated due to the fact that this step needs to keep fixed node and alignment constraints intact.

First, for each component of \( D^h \), similar to the part on relative placement constraints of Section 4.2.1 we use a topological ordering of the nodes involved and calculate longest distances along the \( x \)-axis to all nodes from a source node (one with no incoming edges). In case there is more than a single source node in the component, we normalize their starting position by first relocating all source nodes to their average \( x \) coordinate. Unlike the transformation step though, we also need to take nodes involved in fixed node and alignment constraints into account and make sure not to disrupt these constraints along the way. A fixed node visited during the longest distance calculation will not only affect its successors due to its forced location but it might also require its predecessors to be relocated to satisfy minimum specified separation. Furthermore, in case a node involved in a relative placement constraint is also part of an alignment constraint, we treat the nodes in the alignment constraint as a “block” (i.e., a single merged node) and place them together.

After we finish processing a component, we continue with other remaining components in the \( x \)-axis. Lastly, we process \( D^v \) similarly and complete enforcement of all constraints. Fig. 12 exemplifies enforcing relative placement constraints while Algorithm 2 presents details of the process.

4.3 Phase III: Polishing Phase

During the last phase, we apply an incremental and modified version of the CoSE algorithm on the constrained draft layout to refine the layout by minimizing the stress of the physical model. The main goals include the removal of any node-node overlaps and proper representation of the nesting relations while maintaining enforced constraints.

The original CoSE algorithm does not take constraints into account. In each iteration, the displacement of each node is calculated using various types of forces, and then each node is moved based on these amounts. However, as such movements might violate already established constraints, iCoSE adjusts (i.e., limits) the calculated displace-
ment amounts of the constrained nodes so that no constraint is violated by the movements as detailed below:

- First, for each node with a fixed node constraint, its displacement amount in both directions is reset (i.e., the node will not move).

- Then, for each group of nodes with a vertical (horizontal) alignment constraint, the displacement amount in $x$ ($y$) direction is adjusted to be the average value of displacement amounts in that direction. In case, at least one of these nodes has a fixed node constraint in the same direction, all displacement values in that direction are reset. This is similar to the work in [23], where they treat nodes of an alignment or relative placement constraints as if they are tied together through a "rigid stick" forcing them to move together.

- Finally, if a node is involved in a relative placement constraint, then its displacement amount in that direction is adjusted, and the node is only allowed to move up to a location, where it does not violate the constraint. Notice that if the node is also involved in a fixed node constraint in the same direction, its displacement will already be reset. If the node is also involved in an alignment constraint, its displacement amount will have been previously updated to keep it aligned with others. Here, we further adjust the displacement amount up to the point where it will not violate the relative placement constraints either.

Notice that this newly introduced intermediate step does not change the displacement amount of unconstrained nodes.

### 4.4 Time Complexity

Phase I is expected to run in $O(n + m)$ time, where $|V| = n$ and $|E| = m$. The preprocessing step of handling disconnected graphs and compound structures requires a number of BFS operations, where each node/edge is visited as many times as their depth in the inclusion graph. Assuming the inclusion graph has a height independent of the graph size, as expected with real-life graphs, this operation should work in $O(n + m)$ time. Civril et al.’s [16] spectral layout algorithm also works in linear time in $n$ and $m$. Finally, the postprocessing requires a one-time traversal of all simple nodes to calculate compound node positions and dimensions, working in $O(n)$ time as well.

Phase II is also expected to run in $O(n + m)$ time. The most costly operations in this phase are for the transformation of the draft layout and for enforcing relative placement constraints. These may require finding disconnected components and solving the longest path problem in a dag, which can all be handled with a few, constant number of BFS traversals. Computation of the transformation matrix (multiplying matrices with dimensions $2 \times n$ and $n \times 2$) including the application of an SVD (on a $2 \times 2$ matrix) also works in $O(n)$ time.

Applying a modified CoSE algorithm incrementally, starting with a low cooling factor, in Phase III reduces the number of iterations significantly at the cost of additional overhead per iteration for maintaining already established constraints. Remember that each iteration of the original CoSE algorithm takes $O(n + m)$ time. The overhead due to fixed node and alignment constraints will obviously not affect the asymptotic run time. Dealing with relativity constraints however is more involved but can also be handled within the asymptotic time allocated, assuming the number of relativity constraints involving each node is independent of the graph size. This is a reasonable assumption for user-defined constraints. Notice however that CoLa may need to introduce additional relative placement constraints quadratic in the number of nodes (on top of those defined by the user) to avoid node-node overlaps and handle compound structures.

Hence, the run time of fCoSE is asymptotically upper bounded by its Phase III, which is expected to run in $O(n + m)$ time per iteration like CoSE but needs much fewer iterations as it starts out from a relatively stable initial layout.

### 5 Evaluation

We evaluated fCoSE in terms of both layout quality and run time performance by comparing it with CoLa. CoLa is the closest algorithm to ours due to its support for varying constraint types, non-uniform node dimensions and compound structures with an arbitrary level of nesting. The evaluation of quality focuses on widely accepted layout metrics such as node-node overlaps, node-edge overlaps, edge-edge crossings, average edge length and total area, while the execution duration is measured to evaluate run time performance. In addition, we compared fCoSE’s run time performance against that of its predecessor CoSE.
Algorithm 2 Enforcing Relative Placement Constraints

function ENFORCECONSTRAINTS(G, C, C', C')
    for each dir ∈ {h, v} do
        fixedNodes ← nodes in C'
        metaToOrgMap ← {}
        ▷ bidirectional map btw meta and original nodes in alignment constraints
        M ← {m_i | c_i ∈ C & c_i.dir ≠ dir} ▷ a meta node for each alignment set
        ▷ defined in opposite direction
        M' ← {m_i ∈ M & ∃(x ∈ c_i.nodes) ∧ x ∈ fixedNodes}
        fixedNodes ← fixedNodes ∪ M'
        for each m_i ∈ M' do ▷ set meta node positions based on average position of nodes represented
            m_i.currPos(dir) ← AVERAGEPOS(c_i.nodes, dir)
        for each m_i ∈ M do
            metaToOrgMap.add(m_i, c_i.nodes)
        D^dir ← CALCDA(G(C', 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
            ALINDNDEGREEZEROVERTICES(C, dir) ▷ align zero indegree vertices of C in current direction
            for each node v in C do
                v.predList ← {v}
                if v.indegree(dir) = 0 then
                    queue.enqueue(v)
                    v.newPos(dir) ← v.currPos(dir)
                else
                    v.newPos(dir) ← −∞
                while !queue.empty() do
                    u ← queue.dequeue()
                    for each neighbor v of u where e = (u, v) do
                        pos ← u.newPos(dir) + e.weight
                        if v.newPos(dir) < pos then ▷ constraint violated
                            if v ∈ fixedNodes then
                                v.newPos(dir) ← v.currPos(dir)
                            if v.newPos(dir) < pos then
                                discr ← pos − v.newPos(dir)
                                for each node w ∈ u.predList do
                                    v.newPos(dir) ← v.newPos(dir) − discr
                            else
                                v.newPos(dir) ← pos
                        v.indegree(dir) ← v.indegree(dir) − 1
                        if v.indegree(dir) = 0 then
                            queue.enqueue(v)
                            v.predList ← v.predList ∪ {u}
                    for each node u in M then
                        if u = m_i ∈ M then
                            for each node v ∈ c_i.nodes do
                                v.currPos(dir) ← m_i.newPos(dir)
                    else
                        u.currPos(dir) ← u.newPos(dir)
    end

5.1 Experiment Setup

We implemented CoSE in JavaScript as an extension to Cytoscape.js [30], a graph visualization and analysis library. CoLa and CoSE are also available as Cytoscape.js extensions. Hence, we used these three extension libraries and ran our experiments on an ordinary computer with Intel i7-4790 3.60GHz x 4 CPU and 16GB RAM.

5.2 Dataset

An evaluation was performed both on real-life graphs and on two randomly generated compound graph datasets with...
10 to 5000 nodes with average degree up to 7. The random datasets were generated by converting the Rome graph dataset [38] one of the benchmark datasets used frequently in graph visualization with biconnected, undirected, and 4-planar graphs, and from an assorted selection of dense graphs in the Network Repository [39] containing benchmark datasets into compound graphs as described in the supplementary material. The generated graphs are with increasing proportions of constraints: 25%, 50%, 75% and 100%.

5.3 Results and Discussion
We compared fCoSE with CoLa on some real-life graphs such as the dependency graph and the underwater sensor network in Fig. 13 (refer to the supplementary material for larger versions and more examples), for all of which fCoSE provides a better run time and visual quality performance.

We have also conducted experiments on randomly generated and constrained graphs using the previously defined setup (see Fig. 14 for an example; refer to the supplementary material for a larger version and other examples). We repeated each test 5 times with a new set of constraints in each run and averaged the results. Here we present the comparison of fCoSE with CoLa only on the small-sized random graphs because CoLa does not scale well to graphs with more than a few hundred nodes, its run time increasing excessively. For the Rome graph dataset, not surprisingly, fCoSE outperforms CoLa in terms of run time performance in all constraint types (Fig. 15). In terms of layout quality, fCoSE yields shorter average edge lengths up to 27% on the graphs with fixed node and hybrid constraints and comparable results are observed in other constraint types. In addition, fCoSE produces 37 to 74% fewer edge crossings and 50 to 81% fewer node-edge overlaps in all constraint types. Both have comparable performance on the graphs with fixed node and hybrid constraints in terms of the number of node-node overlaps and total area, while the resulting values in these metrics for CoLa are slightly less for node-node overlaps (up to 10 overlaps) and significantly better for total area on the graphs with alignment and relative placement constraints. We observe that Cola generates more compact layouts at the cost of poor readability in terms of other metrics. Fig. 16 presents the results on various metrics in small-sized graphs with hybrid constraints.

We also compared fCoSE with CoSE for their run time performance on medium-sized graphs. fCoSE provides about 2x speedup over CoSE with constraint-free graphs. It is still faster with constrained graphs as well, even though it needs to do additional work to satisfy constraints ignored by CoSE. Here we remark that alignment and relative placement constraints bring only a slight overhead in the run time of fCoSE, whereas fixed node and hybrid constraints decrease the run time, probably due to mandatory stableness of the fixed nodes yielding faster convergence. One other observation is that the ratio of the fixed node constraints does not affect the layout quality metrics, while an increase in the ratio of the alignment and relative placement constraints affects these metrics negatively as satisfying these constraints becomes more challenging.

The results for the denser Network Repository graphs are generally in line with the results for the Rome graphs. However, as the density of the graphs increases, graphs turn into “hairballs” and a drastic decrease in quality metrics are observed, making visual analysis unproductive.

A more detailed comparison between fCoSE and CoLa in small-sized graphs and the performance of fCoSE in medium-sized graphs, including fCoSE - CoSE runtime comparison, as well as a table detailing the results for denser Network Repository graphs can be found in the supplementary material.

5.4 Extensibility and Limitations
In addition to user-specified constraints supported, fCoSE implicitly tries to satisfy constraints such as avoiding node-node overlaps and placing child nodes within the bounds of a parent compound node. It would also be straightforward to extend the supported constraint types to for instance include orthogonal ordering of nodes by using the current set. In fact, fCoSE may be extended with other constraint types as long as the new constraints can be enforced during Phase II. Once established, maintaining constraints during the last phase should be straightforward. For instance, the user might specify a region of arbitrary shape to use for the drawing. Notice however that since the user might not be able to guess the region for a “snug fit”, it might be a better idea to take a scalable shape, which can be contracted or expanded as needed by the algorithm. A potential improvement for the relative placement constraint would be allowing users to separate nodes with an exact amount as opposed to a minimal one, which should not be very difficult by treating the pair together as a block during the last phase.

fCoSE inherits limitations of force-directed algorithms such as not explicitly addressing edge-edge crossings or efficient usage of space (especially when the display area is assumed to be rectangular). Obviously, the user-specified constraints make the already difficult (NP-hard) problem of producing a good layout [5] even more difficult. A specific limitation of fCoSE is not supporting constraints on compound nodes. Lifting this restriction would be quite a challenge as changes in the geometry of a compound node will affect those of its children and vice versa.

6 Conclusion
We have presented a new algorithm fCoSE for the automatic layout of compound graphs with support for a fairly rich set of user-defined placement constraints. fCoSE performs well in small to medium-sized graphs in terms of both run time and widely accepted layout metrics when compared to its competitors, making it suitable for interactive graph analysis. An open-source implementation of fCoSE along with a demo page can be found on GitHub (refer to the supplementary material for details).

Possible future work includes support for additional constraint types, being able to constrain compound nodes, considering a deliberative approach while selecting nodes in preprocessing step and an improved polishing phase where aligned nodes are allowed to change order by swapping to further relax the underlying system.
Fig. 15. Comparison of run time between fCoSE and CoLa in small-sized graphs (10-200 nodes) with fixed node (top-left), alignment (top-right), relative placement (bottom-left) and hybrid (bottom-right) constraints. Percentages show the ratio of the constrained nodes in the graphs.

Fig. 16. Comparison of layout metrics (average edge length - top-left, edge crossings - top-right, node-node overlaps - middle-left, node-edge overlaps - middle-right, total area (in 10^6 square units) - bottom) between fCoSE and CoLa in small-sized graphs (10-200 nodes) with hybrid constraints. Percentages show the ratio of the constrained nodes in the graphs.
ACKNOWLEDGMENTS

This work was supported by the Scientific and Technological Research Council of Turkey (Ref: 118E131 and 5180088) and Google Summer of Code.

REFERENCES


