
Journal of Graph Algorithms and Applications
http://jgaa.info/ vol. 11, no. 2, pp. 481–511 (2007)

Drawing graphs using modular decomposition

Charis Papadopoulos
Department of Informatics
University of Bergen
http://www.ii.uib.no/∼charis
charis@ii.uib.no

Costas Voglis
Department of Computer Science
University of Ioannina
http://www.cs.uoi.gr/∼voglis
voglis@cs.uoi.gr

Abstract

In this paper we present an algorithm for drawing an undirected graph G that takes advantage of the structure of the modular decomposition tree of G. Specifically, our algorithm works by traversing the modular decomposition tree of the input graph G on n vertices and m edges in a bottom-up fashion until it reaches the root of the tree, while at the same time intermediate drawings are computed. In order to achieve aesthetically pleasing results, we use grid and circular placement techniques, and utilize an appropriate modification of a well-known spring embedder algorithm. It turns out, that for some classes of graphs, our algorithm runs in $O(n+m)$ time, while in general, the running time is bounded in terms of the processing time of the spring embedder algorithm. The result is a drawing that reveals the structure of the graph G and preserves certain aesthetic criteria.

Article Type	Communicated by	Submitted	Revised
Regular paper	P. Eades and P. Healy	January 2006	January 2007

This work is supported by the Research Council of Norway through grant 166429/V30 and co-funded by the European Union in the framework of the project "Support of Computer Science Studies in the University of Ioannina" of the "Operational Program for Education and Initial Vocational Training" of the 3rd Community Support Framework of the Hellenic Ministry of Education, funded by national sources and by the European Social Fund (ESF)

1 Introduction

The problem of automatically generating a clear and readable layout of complex structures inside a graph is receiving increasing attention in the literature [9]. In this work we present a drawing algorithm that takes advantage of the modular decomposition of a graph. Our goal is to highlight the global structure of the graph and reveal the regular structures within it. The usage of the modular decomposition has been considered by many authors in the past to efficiently solve other algorithmic problems [5, 1, 6, 20, 19].

Our approach takes advantage of the modular decomposition of the input graph G, which is a recursive tree-like partition that reveals *modules* of G, i.e., sets of vertices having the same neighborhood. By exploiting the properties of these modules and especially the properties of the modular decomposition tree $T(G)$, we are able to draw the modules separately using different techniques for each one. To achieve aesthetically pleasing results, we utilize a grid placement technique, a circular drawing paradigm, and a modification of a spring embedder method, on the appropriate modules. Our algorithm relies on creating intermediate drawings in a systematic fashion by traversing the modular decomposition tree of the input graph from bottom to top, while at the same time certain parameters are appropriately updated. In the end, the drawing of the graph G is obtained by traversing $T(G)$ from the root to the leaves, in order to compute the final coordinates of the vertices in the drawing area, using the parameters computed in the previous traversal of $T(G)$. It turns out that this way of processing $T(G)$ enables us to visualize the graph in various levels of abstraction.

Many techniques for drawing hierarchical clustered graphs deal with a graph and its tree representation [2, 11, 12, 15, 17, 27, 34, 35]. All these methods address the problem of visualization, by drawing the non-leaf nodes of the tree as simple closed curves. Brandenburg [2] uses an underlying tree-like structure of a graph in order to layout graph grammars. Force directed methods have also been developed to support and show the structure of a clustered graph that is a 2-level decomposition scheme [24, 39]. Also, in [27, 28], the drawing of a clustered graph is considered as a problem of avoiding overlapping vertices with non-uniform size. So far in the literature, two main classes of solutions to the non-uniform vertex overlapping problem exist: the layout adjustment algorithms [13, 21, 23, 29], which are post-processing approaches employed after a final layout is generated, and the force-integration approaches [22, 28, 37, 39], where overlapping is avoided at the same time the layout is created, by a force directed algorithm. This is achieved by modifying the attractive and repulsive forces of the existing methods. Related to modular decomposition there is an algorithm for generating drawings of directed graphs based on a specific parse tree [34, 35]. It is important to note that the approach for the directed graphs apply only on certain digraphs which are decomposed by two different operations (known as series and parallel) [34].

The theory of modular decomposition originates from Gallai's work for computing a transitive orientation [20]. Modular decomposition is also known as *sub-*

stitution decomposition, *X-join decomposition*, and *clan decomposition*. There has been an increasing interest for designing a simple and fast algorithm for computing the modular decomposition of a graph [7, 8, 14, 31, 32]. Some of the linear-time algorithms that use different approaches are given by [8, 31]. Quite recently it has been proposed a linear-time algorithm for computing the modular decomposition of a directed graph [30]. A great number of NP-hard optimization problems can be efficiently solved if a solution for the decomposed subgraphs is known by using modular decomposition. To name a few of them we refer to the problem of computing the treewidth and the minimum fill-in [1]. Moreover it has been proposed to obtain efficient algorithms by expressing optimization problems in monadic second-order logic [6]. Other application areas of modular decomposition arise in biological clustering of proteins [19].

To achieve aesthetically pleasing results, our algorithm utilizes a grid placement technique, a circular drawing paradigm, and a modification of a spring embedder method, on the appropriate modules. It turns out that our drawing methods must take into account the size of non-uniform vertices. Under this constraint, we propose a modified spring embedder algorithm that falls in the category of force-integration approaches. Vertex-to-vertex overlapping is avoided by applying vertex size constraints gradually and by introducing a reducing factor, based on the density of the input graph, that acts on the attractive forces between vertices. We note that considering other aesthetic criteria, such as the number of vertex-to-edge crossings, would conflict with our basic goal of the exposal of desired subgraphs.

In addition, there are cases in which the final drawing of our algorithm takes time linear in the size of the input graph. This arises from the fact that certain classes of graphs have been considered by many authors in the past (see [5]), who explored the structural and algorithmic properties of their modular decomposition trees. It follows that the structure of their trees ensures that each tree node can be processed in time linear in the size of the given part of the tree. Thus, since $T(G)$ can be constructed in time linear in the size of the graph G [8, 31], the processing of the entire modular decomposition tree and consequently its drawing takes time linear in the size of the input graph. Furthermore, our drawing algorithm highlights the global structure of the graph and reveals the regular structures inside the graph.

However in some cases the algorithm produces some unnecessary edge crossings and vertex-to-edge overlapping since it tries to display the symmetry of vertices being in the same module. For that purpose we slightly modify our basic algorithm and propose an alternative drawing approach that takes the vertex-to-edge crossings into account. We achieve this by relaxing certain constrains that used to hold together all the vertices of a module. Although the readability of the new drawing is being improved (in the sense of less edge crossings), the symmetry of certain subgraphs is relegated. Therefore we propose both algorithms in order to achieve the goal of exposing regular structures and the ability of reading a clear drawing of a graph.

Our work is organized as follows. In Section 2 we establish the notation and related terminology and we present background results. In particular, we show

structural properties of a unique tree representation of a graph and establish our drawing conventions with respect to the tree representation. In Section 3, we describe our drawing algorithm, while in Section 4 we propose a modification of a spring embedder algorithm and give their efficient analysis for a given graph. In Section 5 we compute the time complexity of our algorithm and show that the drawings of some classes of graphs can be computed in linear time. Section 6 presents some examples of graphs computed by our drawing algorithm. In Section 7 we slightly modify the algorithm and propose another drawing approach, and, finally, in Section 8 we conclude our work and discuss possible future extensions.

2 Preliminaries

We consider finite undirected graphs with no loops or multiple edges. For a graph G, we denote by $V(G)$ and $E(G)$ the vertex set and the edge set of G, respectively. Let S be a subset of the vertex set of a graph G. Then, the subgraph of G induced by S is denoted by $G[S]$. A *clique* is a set of pairwise adjacent vertices. The *degree* of a vertex x in the graph G, denoted $d(x)$, is the number of edges incident on x. For a graph G on n vertices and m edges, $D(G) = 2m/n$ is the *average degree* of G. The complement of a graph G is denoted by $\overline{G}$.

Let T be a rooted tree. For convenience, we refer to a vertex of a tree as a node. The parent of a node t of T is denoted by $p(t)$, whereas the node set containing the children of t in T is denoted by $ch(t)$. Let h be the height of the tree T. Then, we denote by L_i the node set containing the nodes of the i-th level of T, for $0 \leq i \leq h$ where the root is at level 0.

2.1 Modular Decomposition

A subset M of vertices of a graph G is said to be a *module* of G, if every vertex outside M is either adjacent to all vertices in M or to none of them. The emptyset, the singletons, and the vertex set $V(G)$ are *trivial* modules and whenever G has only trivial modules it is called a *prime* (or *indecomposable*) *graph*. It is easy to see that the chordless path on four vertices, P_4, is a smallest non-trivial prime graph, since graphs with three vertices are decomposable [5]. A non-trivial module is also called *homogeneous set*. A module M of the graph G is called a *strong module*, if for any module $M' \neq M$ of G, either $M' \cap M = \emptyset$ or one module is included into the other. Let M be a module of a graph G. If $G[M]$ is a disconnected graph, then M is called a *parallel* module. If $\overline{G}[M]$ is a disconnected graph, then M is called a *series* module. If both $G[M]$ and $\overline{G}[M]$ are connected graphs, then M is called a *neighborhood* module.

The *modular decomposition* of a graph G is a linear-space representation of all the partitions of $V(G)$ where each partition class is a module. The *modular decomposition tree* $T(G)$ of the graph G (or *md-tree* for short) is a unique labelled tree associated with the modular decomposition of G in which the leaves of $T(G)$

are the vertices of G and the set of leaves associated with the subtree rooted at an internal node induces a strong module of G. Thus, the md-tree $T(G)$ represents all the strong modules of G. An internal node is labelled by either P (for parallel module), S (for series module), or N (for neighborhood module). It is shown that for every graph G on n vertices and m edges, the md-tree $T(G)$ is unique up to isomorphism, the number of nodes in $T(G)$ is $O(n)$ and it can be constructed in $O(n+m)$ time [8, 31]. More details about the modular decomposition of a graph can be found in [7, 8, 14, 31, 32].

Let t be an internal node of the md-tree $T(G)$ of a graph G. We denote by $M(t)$ the module corresponding to t, which consists of the set of vertices of G associated with the subtree of $T(G)$ rooted at node t; note that $M(t)$ is a strong module for every (internal or leaf) node t of $T(G)$. Let $t_1, t_2, \ldots, t_p$ be the children of the node t of md-tree $T(G)$. We denote by $G(t)$ the *representative graph* of node t defined as follows: $V(G(t)) = \{t_1, t_2, \ldots, t_p\}$ and $t_i t_j \in E(G(t))$ if there exists edge $v_k v_\ell \in E(G)$ such that $v_k \in M(t_i)$ and $v_\ell \in M(t_j)$. For the P-, S-, and N-nodes, it is easy to see that the following lemma holds by definition (see also Proposition 18 in [6] and Theorem 2.2 in [31]).

Lemma 2.1. *Let G be a graph, $T(G)$ its modular decomposition tree, and t an internal node of $T(G)$. Then, $G(t)$ is an edgeless graph if t is a P-node, $G(t)$ is a complete graph if t is an S-node, and $G(t)$ is a prime graph if t is an N-node.*

In Figure 1 we show a graph G on 14 vertices and its modular decomposition tree $T(G)$. Since G is disconnected, the root (node t_1) of $T(G)$ is a P-node. The graphs $G[M(t_2)]$ and $G[M(t_3)]$ are the two connected components of G which are shown on the left and right side, respectively, in Figure 1(a). Their representative graphs $G(t_2)$ and $G(t_3)$ consist of 4 ($= |ch(t_2)|$) and 7 ($= |ch(t_3)|$) vertices, respectively.

In general, using modular decomposition for solving a problem can be quite challenging. A great number of NP-hard optimization problems, such as weighted maximum clique and coloring, can be easily solved if a solution is known for every representative graph in the modular decomposition tree. A typical algorithm for exploiting the modular decomposition often has the following structure (see for example [1] and [6]). First, the algorithm computes the modular decomposition tree $T(G)$ of the input graph G using one of the known linear-time algorithms [8, 30, 31]; then, in a bottom-up fashion, the algorithm computes for each node t of $T(G)$ the optimal value for the subgraph $G[M(t)]$ of G induced by the set of all leaves of the subtree of $T(G)$ rooted at t. Thus, the computation starts by assigning the optimal value to the leaves. Then the algorithm computes the optimal value of each interior node t by using the optimal values of all the children of t depending on the type of the node. Finally the optimal value of the root is the optimal value of the problem for the input graph G.

Thus to specify such a modular decomposition based algorithm we only have to describe how to obtain the value for the leaves and which formula to evaluate or which subproblem to solve on P-nodes, S-nodes, and N-nodes using the values of all children as input (see for an exposition [5]). In the present work we utilize a modular decomposition based algorithm to draw arbitrary graphs, combined

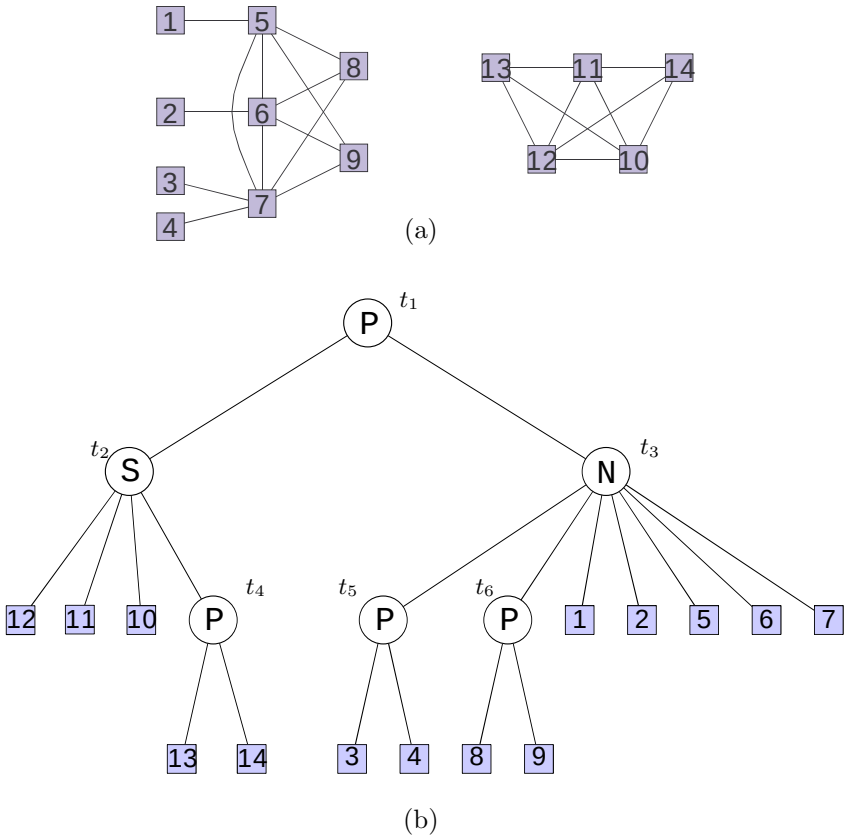


Figure 1: (a) A graph G and (b) its modular decomposition tree $T(G)$.

to well known drawing techniques such as circular drawings and force directed methods.

2.2 Modular Decomposition Based Drawing

Our drawing algorithm is based on the modular decomposition tree of a given graph G. We deal with box-shaped vertices with a specific size. For every $t \in T(G)$ we define $c(t) = (x(t), y(t)) \in \mathbf{R}^2$ to be the coordinates of the center of node t, and $b(t) = (w(t), h(t)) \in \mathbf{R}^2$ to be the dimensions of the box of node t, where $w(t)$ and $h(t)$ are the width and the height of the box, respectively. In other words, $c(t)$ is the center of the box $b(t)$. We adopt the straight-line drawing convention and we impose the following constraints:

C1 vertices do not overlap;

C2 vertices in every strong module $M(t)$, induced by an internal node t of $T(G)$, are drawn close (in terms of their Euclidean distance) to each other;

C3 vertices in every strong module $M(t)$, induced by an internal node t of $T(G)$, are drawn according to the structure (edgeless or complete or prime) of the representative graph $G(t)$;

Observe that we do not consider other aesthetic criteria that are applied commonly in graph drawing algorithms such as the constraint of no vertex-to-edge crossing [9]. The reason is that constraints C1–C3 tend to produce drawings of graphs that reveal the structures of the induced modules. On the other hand classical constraints may be useful in terms of general readability of a drawing, but they cancel the expository properties of C1–C3. A natural technique is to incorporate both kinds of criteria with the best possible manner so that the output drawing exposes important structures while at the same time certain aesthetic criteria are satisfied. However the criteria C1–C3 are somehow incompatible with other aesthetic criteria in the sense that a group of vertices (module) cannot be spread in the layout. Therefore we put as a primary goal the achievement of C1–C3 and we also propose an alternative drawing approach that tries to incorporate other aesthetic criteria.

Definition 2.1. A drawing with the previous constraints is called a *modular decomposition based drawing* (or md-drawing for short) $\Gamma(G)$ of the graph G which is a mapping between the vertices and the Euclidean space $\mathbf{R}^2$: $\Gamma(G) : V(G) \to \mathbf{R}^2$.

Our problem is to produce a drawing $\Gamma(G)$ for a given graph G with non-uniform, box-shaped vertices. Clearly $\Gamma(G) = \Gamma(G[M(root)])$. To produce the final drawing, our algorithm needs to compute a drawing based on the representative graph $G(t)$, for every internal node $t \in T(G)$.

Definition 2.2. A *relative drawing* $\Gamma'(t, T(G))$ is an md-drawing of the representative graph $G(t)$.

We mention that the frame boundaries of a drawing are the horizontal and vertical sides of the smallest rectangle that covers the drawing. The frame boundaries of a relative drawing $\Gamma'(t, T(G))$, define the dimensions of the box $b(t)$ of node t.

2.3 Modular Decomposition as a Clustering Technique

Modular decomposition can be thought as a quite effective clustering technique, that aligns naturally with graph drawing notions, since it clusters vertices with the same neighborhood. This is even more evident if we compare modular decomposition to previous suggestions that were either too specialized (i.e., partition into cycle of cliques [3]), too broad (biconnected components [9]) or too heuristic (i.e., structural clustering [25]). It is also of great importance that we suggest a technique that automatically creates and draws clusters, in opposition to other drawing methods that are applied on an already clustered graph.

A useful insight on qualifying tree-like decompositions is given in [16]. This work presents some design criteria that should be fulfilled by the decomposition trees so that graph visualization applications can substantially benefit from their usage. Two of the general clustering criteria are that good decompositions should (i) exhibit a strong relationship between vertices in the same cluster and (ii) preserve the relevant connectivity properties of the graph, so that the viewer can grasp them even observing a contraction of it. Modular decomposition satisfies both these criteria, because a cluster (module) consists of vertices having the same neighborhood. We note that for directed graphs the usage of the corresponding modular decomposition achieves both goals as shown in previous works [34, 35].

Finally, other clustering techniques [25] may not preserve the information of the graph in various levels of contractions. The representative graph of an arbitrary clustered graph, also known as the *quotient* graph [8, 25], results by contracting all clusters into single nodes. Notice that an edge in the quotient graph may represent few edges from the original graph connecting some vertices which belong in different clusters. Thus, by examining the representative graph of clusters computed with any arbitrary manner, one looses information about the connectivity of the graph. On the other hand in modular decomposition based clustering if two nodes, which represent a contraction of two clusters (modules) are connected, then all the vertices of these clusters are also connected. In this way there is no loss of information by examining the edges of the representative graph obtained by contracting the modules of the graph (see also [34]).

3 The Algorithm

Let G be a graph on n vertices $v_1, v_2, \ldots, v_n$ with non-uniform dimensions $b(v_1), b(v_2), \ldots, b(v_n)$, respectively, and m edges. Our algorithm first computes the md-tree $T(G)$ using one of the known linear-time algorithms [8, 31]. For every internal node $t \in T(G)$ we perform an initialization step, by setting its box dimensions to zero. We recall that the boxes of the leaves of $T(G)$ have predefined dimensions, since they are the vertices of G. In this step, we also set the center of every node $t \in T(G)$ to zero.

In bottom-up fashion, we traverse the md-tree $T(G)$ and calculate the relative drawing $\Gamma'(t, T)$ for every internal node t. More precisely, according to the type of node t (P or S or N) we apply a specific drawing algorithm to layout all the children of t, using the representative graph $G(t)$; recall that $G(t)$ is either an edgeless, or a complete, or a prime graph. This relative drawing modifies the coordinates of the center $c(t_i)$ for every node $t_i \in ch(t)$, taking into account the dimensions of the non-uniform boxes $b(t_i)$. This implies that only the children of t obtain the new coordinates according to the relative drawing. In order to apply the new coordinates to the subtree rooted at t, and finally to the graph $G[M(t)]$, we store the displacements from the previous coordinates, $dis(t_i)$ for every t_i. We use $dis(t_i)$ to update the coordinates of the vertices of $G[M(t)]$

in a later step. In addition, we update the dimensions of the box of node t in proportion to the frame boundaries of the relative drawing $\Gamma'(t,T)$.

Finally, we traverse the md-tree $T(G)$ in a top-down fashion and for every internal node $t \in T(G)$, we add the displacement $dis(t)$ to the centers of the boxes of every child node $t_i \in ch(t)$. In this way, all the vertices of $G[M(t)]$ obtain the right coordinates relative to the center of their ancestor node t. Thus, the final output drawing $\Gamma(G)$ of G, is $\Gamma(G[M(root)])$.

We mention that every relative drawing uses a predefined constant k_i as the preferred edge length of the drawing at the level set L_i, $0 \leq i \leq h-1$, of the md-tree $T(G)$. Two approaches can be considered regarding the preferred edge lengths k_i. In the first, all k_i hold the same value and in the second, k_i vary as an inversely proportional function of i (see also [38]). With the second approach, each module of G can be drawn further apart and can be distinguished more clearly, whereas with the first approach more compact drawings are created.

The algorithm, called *Module_Drawing*, is given in detail in Algorithm 1. We note that the preferred edge lengths k_i are global variables. The relative drawings are done by means of three functions, namely, *Draw_Edgeless*, which is applied on a P-node, *Draw_Complete*, which is applied on a S-node, and *Draw-Prime*, which is applied on a N-node. Recall that the leaves of $T(G)$ are the vertices of G.

Algorithm **Module_Drawing**

Input: A graph G on n vertices and m edges.

Output: An md-drawing $\Gamma(G)$ of the graph G.

1. Construct the modular decomposition tree T of the graph G;
2. Initialize the centers $c(t) \leftarrow (0,0)$ for every $t \in T$ and set $b(t) \leftarrow (0,0)$ for every internal node $t \in T$;
3. Compute the node sets $L_0, L_1, \ldots, L_h$ of the levels $0, 1, \ldots, h$ of T, and assign values to the preferred edge lengths k_i;
4. **for** $i = h-1$ down to 0 **do** { *bottom-up fashion*}
   for every internal node $t \in L_i$ **do**
   - 4.1 **if** t is a P-node **then** $\Gamma'(t,T) \leftarrow$ *Draw_Edgeless*(t,T);
   - 4.2 **else if** t is an S-node **then** $\Gamma'(t,T) \leftarrow$ *Draw_Complete*(t,T);
   - 4.3 **else** {t *is an N-node*} $\Gamma'(t,T) \leftarrow$ *Draw-Prime*(t,T);
   - 4.4 Compute the displacement $dis(t_i)$, for each node $t_i \in ch(t)$, with respect to the previous placement;
   - 4.5 Update the size of the rectangle box $b(t)$, according to the frame boundaries of $\Gamma'(t,T)$;
5. **for** $i = 0$ down to $h-1$ **do** { *top-down fashion*}
   for every internal node $t \in L_i$ **do**
   for every child $t_i \in ch(t)$ **do** $c(t_i) \leftarrow c(t_i) + dis(t)$;
6. Return the drawing $\Gamma(G) = \Gamma'(r,T)$ computed in the root r of T;

Algorithm 1: *Module_Drawing*.

The formal descriptions of functions *Draw_Edgeless* and *Draw_Complete* are given below whereas the function *Draw-Prime* is described in detail in Section 4. All these functions are aware of the preferred edge length, denoted by k, which may be different for each level of $T(G)$. We note here, that one can use different drawing techniques for each relative drawing to fulfill desired aesthetic criteria. Our approach draws edgeless graphs on an underlying grid, complete graphs in a circular way, and prime graphs using a spring embedder method. We point out that there are other well known techniques for drawing the nodes (intermediate drawings) of the tree, such as the inclusion tree layout convention [27] or more general the floor-planning approach used in VLSI design [26]. Although these approaches can be applied in order to produce the final drawing of the graph, we choose the technique described in Algorithm Module_Drawing for clarity and simplicity reasons.

3.1 Function Draw_Edgeless

Vertices are placed by function Draw_Edgeless, keeping in mind that there are no connecting edges between them. This is achieved by a grid placement of the nodes in an arbitrary order. Since the nodes have non-uniform sizes, the task of minimizing the drawing area is NP-hard by the two dimensional bin packing problem [27]. A way to avoid this is to allow only two possible arrangements of the nodes, either horizontal or vertical. In [27] this restricted problem is solved in polynomial time by using a dynamic programming approach. Here we do not allow this restriction and propose a simple algorithm for the grid placement of the nodes, even though one can easily apply the approach of [27] whenever the problem of minimizing the drawing area is primary.

The approach that we propose is quite simple. The Euclidean distance between the boundaries of two nodes placed adjacent on the grid is at least k. For symmetry reasons, we distribute evenly the space between the nodes in each row, so that a complete alignment is achieved. Each row is then processed one by one and it is placed below the previous one, keeping distance of at least k from the bottom boundary of the previous row. The function Draw_Edgeless is given in details below.

Function **Draw_Edgeless**(t, T)

1. Set $r = \left\lceil \sqrt{|ch(t)|} \right\rceil$, $c = \left\lceil \sqrt{|ch(t)|} \right\rceil$ and define a grid F of r rows and c columns;
2. For every $t_l \in ch(t)$, map t_l in arbitrary order, to be at position $F(i, j)$;
3. **for** every row i of F **do**
   3.1 For every t_l mapped in row i, set $y(t_l) = 0$ and $x(t_l)$ such that the shortest horizontal distance between the boundaries of the neighborhood boxes is k.

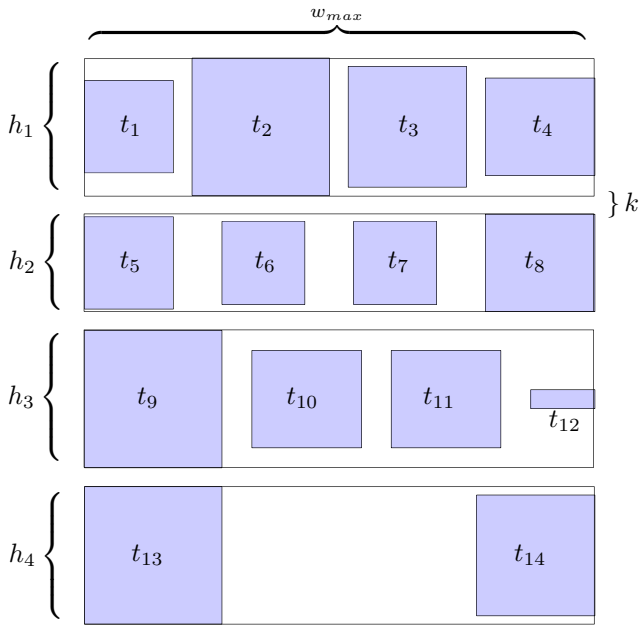


Figure 2: Illustrating Function Draw_Edgeless.

3.2 Set $w_i = (c-1)\cdot k + \sum_{t_l \in \text{row } i} w(t_l)$ and $h_i = \max_{t_l \in \text{row } i}(h(t_l))$;

4. Set $w_{max} = \max(w_i)$;

5. **for** $i = 2$ to r **do**

   5.1 For every t_l mapped in row i of F, update $x(t_l)$ such that the total length of row i be w_{max};

   5.2 For every t_l mapped in row i of F, update $y(t_l)$ such that the shortest vertical distance between the neighborhood rows of height h_{i-1} and h_{i+1} is k;

6. Return $\Gamma'(t, T)$, containing the centers $c(t_l)$ for every $t_l \in ch(t)$;

In Figure 2 we illustrate the output of function Draw_Edgeless on 14 disconnected vertices with non-uniform sizes, using preferred edge length $k = 30$. The algorithm computes a grid F of 4 rows and 4 columns. In Step 3 of function Draw_Edgeless, the quantities h_1, h_2, h_3, h_4 are computed according to the height of the vertices of each row. Also in Step 4 the quantity w_{max} is set as the maximum length, among the 4 rows of F. Note that, in the last row of the grid, vertices t_{13} and t_{14} are aligned left and right, respectively.

3.2 Function Draw_Complete

Function Draw_Complete is basically a circular drawing algorithm, eventhough the representative graph $G(t)$ is a complete graph. We have chosen to draw complete graphs in this way, in order to expose the structure of a series module (see constraint C3). Furthermore, a circular drawing satisfies the aesthetic criterion of symmetry and is the usual way of representing complete graphs in textbooks.

Six and Tolis [36] proposed an approach for placing group of vertices in a circular way so that the number of edge crossings is low. Their algorithm is based on a force-directed technique and requires at least n^2 steps for a graph on n vertices. However in our case all edges between the corresponding group of vertices (modules) are present, since $G(t)$ is a complete graph. Thus we do not seek to achieve a drawing with low number of edge crossings within $G(t)$, since we focus on the close placement of the vertices of $G(t)$ and the ability of distinguishing the property of $G(t)$ being a complete graph. Next we present a quite simple deterministic algorithm for placing non-uniform sized vertices in a circular way that runs in $O(n)$ time and without the knowledge of the edges of the graph.

The vertices of the series module are placed in an arbitrary order on equal arcs, on the circumference of a cycle centered at $c(t)$. The initial radius is determined by the smallest sized box. More specifically, function Draw_Complete process each node $t_i \in ch(t)$ one by one. It computes a value $f(t_i)$ that represents the maximum distance from the center $c(t_i)$ of the box $b(t_i)$, to a point on its boundary. We use $f(t_i)$ to determine the distance between two adjacent vertices in the circle, in the following way. For every vertex t_i we calculate two radiuses r_1 and r_2 that define two concentric circles centered at $c(t)$. In the circle defined by r_1, vertex t_i and its previous t_{i-1} are placed on a θ degrees arc, so that the minimum distance from their boundaries, is k. The radius r_2 defines a circle where t_i and t_{i+1} are placed in the same way. To prevent overlaps, we set $r(t_i)$ to be the maximum of these two radiuses. In the special case where the box $b(t_i)$ of t_i has the same dimensions with any of the two adjacent vertices in the circle C, we set $r(t_i)$ to be the radius defined by the equal sized box adjacent vertex. With this approach, we achieve that equal sized boxes are placed on the same circle, without any overlap, since the unequal sized vertex will be placed on a different circle. Finally, we draw every vertex t_i on the circle defined by radius $r(t_i)$, with angle θ. Obviously, for a complete graph with uniform nodes the drawing is a perfect circle. The formal description of function Draw_Complete is given in details below.

Function **Draw_Complete**(t, T)

1. For every $t_i \in ch(t)$ set $f(t_i)$ as the half of the length of the diagonal of box $b(t_i)$;
2. Define a circle C on $|ch(t)|$ vertices and set $\theta = \frac{2\pi}{|ch(t)|}$;
3. For every $t_i \in ch(t)$ map t_i in arbitrary order, on the circle C;

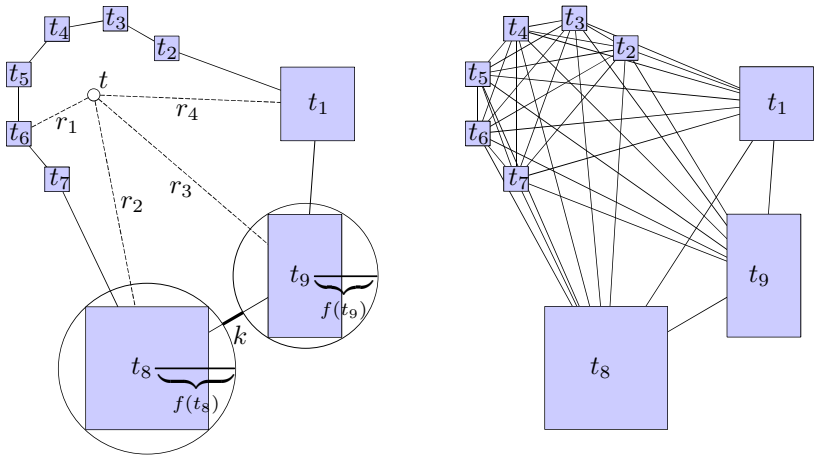


Figure 3: Illustrating Function Draw_Complete.

4. **for** every $t_i \in ch(t)$ **do**

   4.1. Set $r_1 = \frac{f(t_i)+k+f(t_{i-1})}{2\sin(\theta/2)}$ and $r_2 = \frac{f(t_i)+k+f(t_{i+1})}{2\sin(\theta/2)}$, where t_{i-1} and t_{i+1} are the two adjacent vertices of t_i in the circle C;

   4.2. **if** $b(t_i) = b(t_{i-1})$ **then** $r(t_i) = r_1$;
   else if $b(t_i) = b(t_{i+1})$ **then** $r(t_i) = r_2$;
   else $r(t_i) = \max(r_1, r_2)$;

5. For every $t_i \in ch(t)$, put (calculate $c(t_i)$) t_i on a circle, centered on $c(t)$ with radius $r(t_i)$;

6. Return $\Gamma'(t, T)$, containing the centers $c(t_i)$ for every $t_i \in ch(t)$;

In Figure 3(a) we present an illustration of Draw_Complete on 9 vertices with preferred edge length $k = 30$. The algorithm computes 4 different radiuses, according to the size of each box, namely r_1, r_2, r_3, r_4. Note that vertices $t_2, t_3, \ldots, t_7$ are placed on a circle of radius r_1, since they have the same sizes. It is also clear, that each value $f(t_i)$ defines a circumcircle around the box $b(t_i)$ (see for example $f(t_8), f(t_9)$). Figure 3(b) is the final relative drawing of the S-node t, with 9 children. Remark that the representative graph $G(t)$ is a complete graph.

For the time complexity of functions Draw_Edgeless and Draw_Complete, it is easy to see that the following lemma holds.

Lemma 3.1. *Let $T(G)$ be a modular decomposition tree of graph G and let $ch(t)$ be the set of children of a P-node (resp. an S-node) $t \in T(G)$. Function Draw_Edgeless (resp. Draw_Complete) constructs a relative drawing $\Gamma'(t, T)$ in $O(|ch(t)|)$ time.*

4 Modified Spring Embedder

In this section we describe in detail a spring embedder algorithm for the implementation of function Draw_Prime. Recall that this function is applied on an N-node $t \in T(G)$. Since the representative graph $G(t)$ is a prime graph, function Draw_Prime requires the vertex set $V(G(t))$ and the edge set $E(G(t))$.

The main task of Draw_Prime is to combine the aesthetic properties of a spring embedder algorithm with the constraint that no vertex-to-vertex overlapping occurs. The fact that Draw_Prime is applied on the representative graph $G(t)$ that contains vertices with non-uniform sizes, makes the drawing task more demanding.

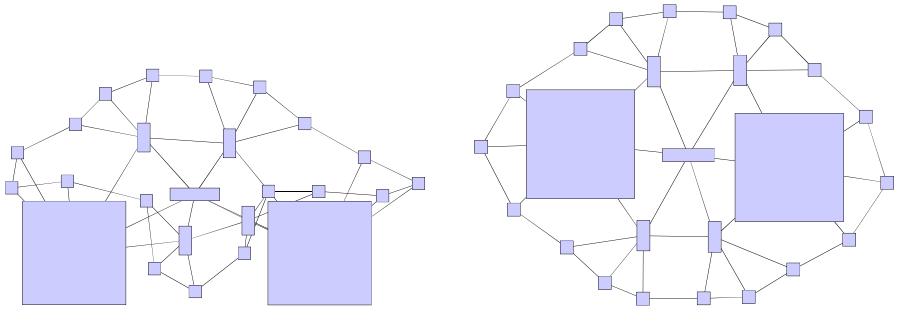
The function Draw_Prime falls in the category of force-integration approaches [28, 22]. It is based on the Fruchterman & Reingold (FR) spring embedder algorithm [18] and follows the general guidelines of Harel & Koren [22]. Draw_Prime consists of a main iteration loop, that is repeated until some termination criteria are met. There are three basic steps to each iteration: (i) calculate the effect of the edge-attractive forces (ii) calculate the effect of vertex-to-vertex repulsive forces and (iii) limit the total displacement by a quantity called *temperature* which is decreased over the iterations. The temperature is decreased by a *cooling schedule*, the choice of which greatly affects the quality of the drawing. To summarize, Draw_Prime starts with an initial random placement of the vertices and an initial temperature, and performs the main iteration loop, until the underlying physical system reaches an equilibrium state. As presented in [18], we choose a two phase cooling scheme: the first phase starts with a constant initial temperature and reduces it using an exponential cooling scheme, and the second phase, which starts after a number of iterations, maintains a constant low temperature.

As already mentioned, we must take into account the size of the children t_i of a node t so that vertices of $G(t)$ would not overlap. To achieve this, we have modified the formulas for the attractive and the repulsive forces between the vertices of the graph. The final formulae for the forces will be presented later in the section. We will first describe the heuristics that we use to avoid overlapping. According to [22], the first modification to the original FR algorithm will result the following formulae for the attractive f_a and the repulsive f_r forces:

$$FR: \ f_a(r_{FR}) = \frac{r_{FR}^2}{k} \quad \text{and} \quad f_r(r_{FR}) = \frac{k^2}{r_{FR}}$$

$$Modified\ FR: \ f_a(r_{MFR}) = \frac{r_{MFR}^2}{k} \quad \text{and} \quad f_r(r_{MFR}) = \frac{k^2}{\max(r_{MFR}, \epsilon)},$$

where $r_{FR} = ||c(t_i) - c(t_j)||_2$, $r_{MFR} = f(t_i, t_j)$, and $f(t_i, t_j)$ is the shortest



(a) (b)

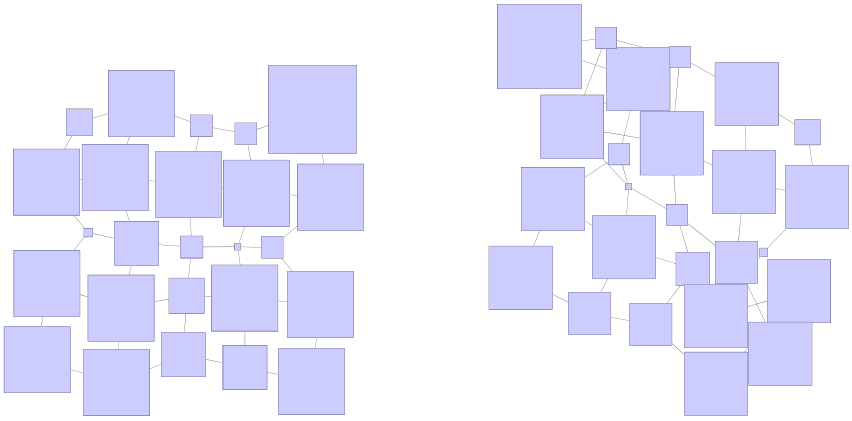
Figure 4: Drawings of a graph using Modified FR forces (a) from the first iteration and (b) after the first 50 iterations.

distance between the boundaries of the boxes $b(t_i)$ and $b(t_j)$. The variable k is the preferred edge length for the drawing and ϵ is a small positive number.

The next extension is to impose the vertex size constraints gradually. Specifically, at the early iterations of our spring embedder the vertices of the prime graph are considered dimensionless, and thus, we use the forces of the FR algorithm. This policy, combined with a large initial temperature, allows the layout to escape possible local optimum states. In this way a possible cluttered layout is found at early stages of the algorithm, and then, we use the Modified FR repulsive and attractive forces to fully prevent overlaps (see also [22]). In Figure 4(a) we show a drawing example of a graph taken from [22] using only Modified FR attractive and repulsive forces and in Figure 4(b) we present a better layout of the same graph using our technique.

After thorough experimental testing we noticed that in many cases, and especially for some small choices of preferred edge length k, the results were not satisfactory. Vertex overlapping could not be avoided, since k was small with respect to the dimensions of the vertices. This defect appeared most frequently when the representative prime graph $G(t)$ was dense. The reason is that the large number of attractive forces, combined with a small value of k, do not allow large vertices to be in a certain distance in order to avoid overlapping. To overcome this problem we decided to use a factor w in the calculation of the edge attractive forces, inversely proportional to the graph's density. In this manner, we succeed in weakening edge attractive forces, and allowing the algorithm to position vertices without overlaps.

Hereafter we will denote by G the representative graph $G(t)$. To compute the reducing factor w, we use the average degree $D(G)$ that can be thought as a measure for the connectivity of G. To be more precise, we use $D^{-1}(G)$ as the factor in the Modified FR edge attractive force calculation f_a. It follows that the use of $D^{-1}(G)$ as a multiplicative factor weakens the attractive forces between vertices. Note that, since the smallest prime graph is a P_4, for a prime



(a) (b)

Figure 5: Drawings of a 5×5 grid using (a) $w = D^{-1}(G) = 0.31$ and (b) $w = 1$.

graph G we have: $0 < D^{-1}(G) \leq 0.57$.

So far, we have explained how the usage of $D^{-1}(G)$ will aid in the process of obtaining a better layout of dense graphs. However, when the graph is sparse, such an approach will lead to wasted drawing space, since vertices will be attracted by a small number ($|E(G)|$) of weak forces. To overcome this, after a certain point in the algorithm we use $D(G)$ as the multiplicative factor so that the final layout would be more compact. The distinction between sparse and dense graphs is rather vague. One approach is to put a threshold in the middle of the interval of the previous inequality of $D^{-1}(G)$ and consider dense the graphs G such that $D^{-1}(G) < 0.28$ and sparse the graphs such that $D^{-1}(G) > 0.28$. However this approach leads to a possible misjudgement of sparse and dense graphs, since there is no reason of such a uniform distribution of $D^{-1}(G)$. Another way is to choose a number q, $1 < q < 2$, and define the graph to be sparse if $|E(G)| = O(|V(G)|^q)$ [33]. Applying such a heuristic for distinguishing sparse and dense graphs gives the desired multiplicative factor for each corresponding case. Experimental results, lead us to use the multiplicative factor $D^{-1}(G)$ from the beginning, regardless whether the graph is sparse or dense. The reason is that this technique provides additional help to overcome possible local minimum states. Moreover it can be thought of as a heuristic in order to allow the drawing to occupy large drawing area at early iterations so that its structure has the ability to be unfolded by weakening forces. At final iterations we strengthen the attractive forces in order to reduce the area and achieve a more compact layout.

In Figure 5 we show two drawings of a 5×5 grid with random dimensioned vertices. The preferred edge length is set to $k = 60$, which is a small number with respect to the dimensions of the vertices. In Figure 5(a) the factor $w = D^{-1}(G) = 0.31$ is used in the early iterations for the calculation of the attractive forces. We consider grid graphs to be sparse, which implies that the factor is

reversed ($w = D(G)$) at final iterations and, thus, the layout becomes more compact. In Figure 5(b) the multiplicative factor w is set to one in all iterations.

Having described the two main features of our spring embedder algorithm, we can present the attractive and repulsive forces of function Draw_Prime (DP) as follows. We mention that the early and the final iterations coincide with the first and the second part of the cooling schedule, respectively.

$$DP:\ f_a(r_{DP}) = \frac{w \cdot r_{DP}^2}{k} \quad \text{and} \quad f_r(r_{DP}) = \frac{k^2}{\max(r_{DP}, \epsilon)} \tag{1}$$

where, $r_{DP} = \begin{cases} ||c(t_i) - c(t_j)||, & \text{at early iterations} \\ f(t_i, t_j), & \text{at final iterations} \end{cases}$

and $w = \begin{cases} D^{-1}(G), & \text{at early iterations} \\ D(G), & \text{at final iterations, and} \\ & \text{if } G \text{ is sparse.} \end{cases}$

For the practical behavior of algorithm Draw_Prime, it is crucial to discuss about the termination criteria. Commonly, the algorithm stops if it reaches a maximum number of iterations. Nevertheless, we use a heuristic to determine whether the drawing reached a state of minimum energy and the algorithm can be stopped. This heuristic is based on the total displacement of all the vertices at one main iteration.

In the j-*th* iteration we calculate the total displacement $\sigma(j)$ for all $t_i \in ch(t)$:

$$\sigma(j) = \sum_{t_i \in ch(t)} ||s(t_i)||_2. \tag{2}$$

Every κ iterations we compute the mean value $\widetilde{\sigma}(j, \kappa)$ of the differences of total displacement

$$\widetilde{\sigma}(j, \kappa) = \frac{1}{\kappa} \sum_{l=j}^{j+\kappa-1} |\sigma(l+1) - \sigma(l)|. \tag{3}$$

The algorithm stops if

$$sc \equiv \left| \frac{\widetilde{\sigma}(j+\kappa, 2\kappa) - \widetilde{\sigma}(j, \kappa)}{\widetilde{\sigma}(j, \kappa)} \right| < \epsilon,$$

where $\epsilon < 1$ is small positive number. Quantity sc has been extensively tested as a termination criterion with very promising results. In detail function Draw_Prime is presented below.

Function **Draw_Prime**(t, T)

---

1. **while** iterations $\leq$ *maxiter* **do**

   1.1 **for** every vertex $t_i \in V(G(t))$ **do**
   Set the displacement $s(t_i) := 0$;

for every vertex $t_j \neq t_i \in V(G(t))$ **do**
calculate the displacement $s(t_i)$ using the repulsive force f_r between vertices t_i and t_j as described in Equation (1);

1.2 **for** every edge $t_i t_j \in E(G(t))$ **do**
calculate the displacements $s(t_i)$, $s(t_j)$ using the attractive force f_a
between vertices t_i and t_j as described in Equation (1);

1.3 **for** every vertex $t_i \in V(G(t))$ **do**
calculate the coordinates $c(t_i)$ using $\min(\mathcal{T}, s(t_i))$;

1.4 **if** convergence criterion is met **then break**;

1.5 **if** $\mathcal{T} \leq \mathcal{T}_{min}$ **then**
if $G(t)$ is dense **then** $w = 1/w$;
else
reduce the temperature $\mathcal{T}$;

2. Return $\Gamma'(t, T)$, containing the centers $c(t_i)$ for every $t_i \in V(G(t))$;

We denote by ℓ the number of the main iterations needed by our spring embedder algorithm. Fruchterman & Reingold suggested that ℓ is proportional to the size of the graph [18]. Tunkelang in his PhD thesis viewed ℓ as a linear function of the number of vertices [37]. Since the second step inside the main iteration of Draw_Prime requires $|V(G)|^2$ operations, we conclude with the following lemma.

Lemma 4.1. *Let $T(G)$ be a modular decomposition tree of graph G and let $ch(t)$ be the set of children of an N-node $t \in T(G)$. Function Draw_Prime constructs a relative drawing $\Gamma'(t, T)$ in $O(\ell \cdot |ch(t)|^2)$ time, where ℓ is the number of main iterations that a spring embedder algorithm performs.*

A faster drawing technique for prime graphs can be incorporated in our general framework of Algorithm Module_Drawing, as long as it conforms with the following constraints: (i) takes node sizes into account and (ii) prevents overlaps, regardless of the preferred edge length.

5 Time Complexity

Next, we introduce the definition of the prime cost of a graph which we will need in our analysis. Let G be a graph and $T(G)$ be its modular decomposition tree. We denote by $\alpha(G) = \{t_1, t_2, \ldots, t_s\}$ the set of the N-nodes of $T(G)$. We define the *prime cost* of G as the value

$$\phi(G) = \sum_{t \in \alpha(G)} \ell \cdot |ch(t)|^2,$$

where $ch(t)$ denotes the set of children of node t in $T(G)$ and ℓ is the number of iterations of the spring embedder.

It is not difficult to see that for any n-vertex graph G, we have $\phi(G) = O(\ell \cdot n^2)$; for an n-vertex P_4-free graph (also known as cograph) G we have $\phi(G) = 0$, since its md-tree (also known as cotree) does not contain any N-node [5]. It follows that in other classes of graphs their prime cost is constant. For example, any N-node of the md-tree of a P_4-reducible graph[1] contains at most five children [5]. Hence for an n-vertex P_4-reducible graph G we have $\phi(G) = O(1)$. We notice that these classes of graphs arise in applications such as examination scheduling problems and semantic clustering of index terms [5].

Theorem 5.1. *Let G be a graph on n vertices and m edges. Algorithm Module_Drawing constructs an md-drawing $\Gamma(G)$ in $O(n + m + \phi(G))$ time, where $\phi(G)$ is the prime cost of the input graph G.*

Proof. Step 1 of Module_Drawing takes $O(n + m)$ time, since the construction of the modular decomposition tree $T(G)$ of the graph G can be implemented in linear time using one of the known algorithms in [8, 31]. Step 2 and the computation of the level sets $L_0, L_1, \ldots, L_{h-1}$ of the tree $T(G)$ in Step 3 can be performed in $O(n)$ time, since the tree $T(G)$ contains $O(n)$ nodes. Additionally, note that exactly one of the functions Draw_Edgeless, Draw_Complete and Draw_Prime is applied on each of the nodes of $T(G)$.

When the functions Draw_Edgeless and Draw_Complete are applied on a P-node and an S-node t, respectively, the relative drawing is computed in $O(|ch(t)|)$ time (Lemma 3.1). Lastly, function Draw_Prime requires $O(\ell \cdot |ch(t)|^2)$ time (Lemma 4.1), when applied on a N-node t. Steps 4.4 and 4.5 require $|ch(t)|$ time and constant time respectively.

The top-down traversal of the md-tree $T(G)$ in Step 5 is performed in $O(n)$ time, since each node t of $T(G)$ is processed once. Keeping in mind that $T(G)$ contains $O(n)$ nodes, the overall time complexity of Module_Drawing is $O(n + m + \phi(G))$ where $\phi(G)$ is the prime cost of the input graph G. ∎

Based on the previous result and by the definition of a prime cost we obtain a linear-time algorithm for computing an md-drawing for certain classes of graphs. Such classes are the classes of extended P_4-reducible graphs, P_4-reducible graphs, cographs, along with their subclasses, such as, the trivially perfect graphs and the threshold graphs [5].

6 Implementation and Examples

We have implemented our algorithm in C++. The implementation takes as input an undirected graph G in GraphML format [4]. The vertices are thought of as rectangles with a predefined size, i.e., with a specific height and width. Three files are produced in GraphML format: a file that contains the final drawing of G, a file that contains the md-tree $T(G)$, and a file that contains

[1] A P_4-reducible graph is a graph for which no vertex belongs to more than one P_4.

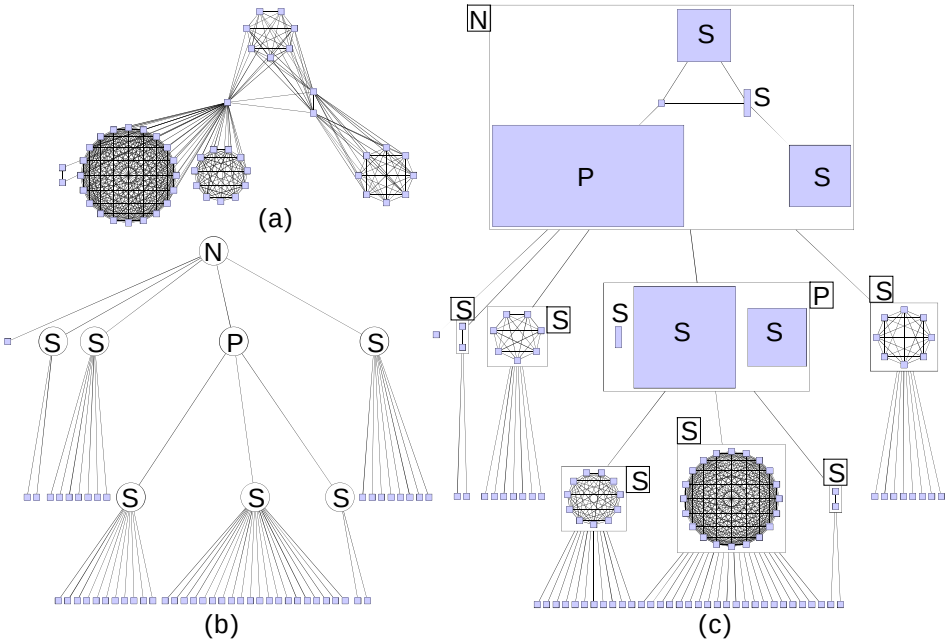


Figure 6: Illustration of Module_Drawing on *Trans* graph

all the relative drawings computed in each level of $T(G)$. For visualization purposes, we use the yEd environment [40].

6.1 An Example of Module_Drawing

In this section, we illustrate how our algorithm produces a final drawing, by showing level-by-level relative drawings, on the md-tree of the input graph. For this purpose we use an input graph from a real life application, which describes a protein interaction network (see [19] for details). More specifically, the input graph, which we will call a *Trans* graph, describes a network of proteins that define transcriptional regulator complexes. The md-tree of the Trans graph contains 1 P-node, 6 S-nodes, and 1 N-node. In Figure 6(a) we present the final drawing of Trans graph using Module_Drawing, in Figure 6(b) we show its modular decomposition tree and in Figure 6(c) we present level-by-level relative drawings and how they are combined to result the final layout.

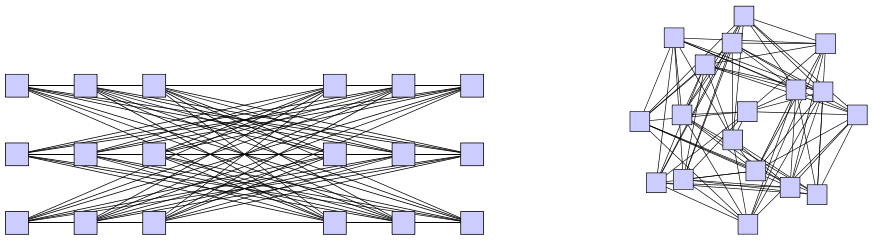
Starting from level 3 of the tree in Figure 6(c), we notice three S-nodes. The application of the function Draw_Series results in the relative drawings as shown in the corresponding boxes. Their parent, which is a P-node, causes them to be drawn on a 1×3 grid. Finally, the root of the md-tree is an N-node; in particular $G(root)$ is an $\mathcal{A}$-shaped graph, that consists of 1 parallel module, 3 series modules, and 1 simple vertex. The final drawing reveals all modules and gives a useful insight of the structure of the Trans graph. Moreover, function

Draw_Prime, which is the most expensive part of our algorithm in terms of time complexity, is applied on a graph of 5 vertices instead of 51. Notice that in the final drawing depicted in Figure 6(a) the two left most vertices are adjacent with the simple vertex but their edges are hard to follow since they cross vertices of the clique that lies next to them. However the parallel module consisting of three disjoint cliques is clearly exposed.

6.2 Drawing Examples

In all the examples we choose to draw the vertices of a graph over its edges. The height and width of all the vertices are set to 30 points. As already mentioned in the description of Module_Drawing, we increase the preferred edge length k_i of the i-th level, starting from the level $h-1$ of $T(G)$. Thus, we set k_{h-1} to a constant and $k_i = (h-i) \cdot k_{h-1}$, for $i = h-2, h-3, \ldots, 0$. Obviously, $k_i < k_{i-1}$. We note that an alternative scheme for increasing the preferred edge length between levels is presented in [38].

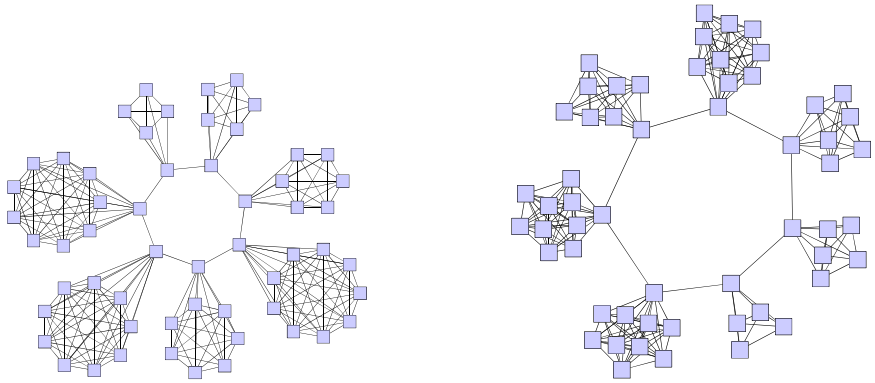
For each example drawn by our algorithm, we present an additional drawing created by a spring embedder method. For this purpose we apply the Smart Organic Layout (SOL) utility of yEd [40] with desired parameters. We make clear that, there is no reason to compare our method to any spring embedder algorithm, since their drawing goals are different. We use a general purpose drawing algorithm, such as spring embedder, to obtain a reference layout of a graph. Note also that we incorporate a spring embedder method in the general framework of our approach.



(a) (b)

Figure 7: Drawings of $K_{9,9}$ using (a) Module_Drawing and (b) Smart Organic Layout.

In Figure 7(a) we present the drawing of a complete bipartite graph $K_{9,9}$, using preferred edge lengths $k_1 = 60$, and $k_0 = 120$, whereas in Figure 7(b) we present the drawing of the same graph using SOL with preferred edge length set to 140. Notice, that our algorithm runs in linear time on the size of the input graph, since complete bipartite graphs are cographs [5], and manages to expose the two partitions. Due to the density of the input graph, spring embedder algorithm does not output an aesthetically pleasing result and overlaps the two partitions. However observe that in our drawing due to the exposure of the



(a) (b)

Figure 8: Drawings of a cycle of cliques using (a) Module_Drawing and (b) Smart Organic Layout.

parallel modules by the grid placement there are many vertex-to-edge crossings that make non-adjacent vertices hard to identify (especially for the vertices lying in the same y-coordinate).

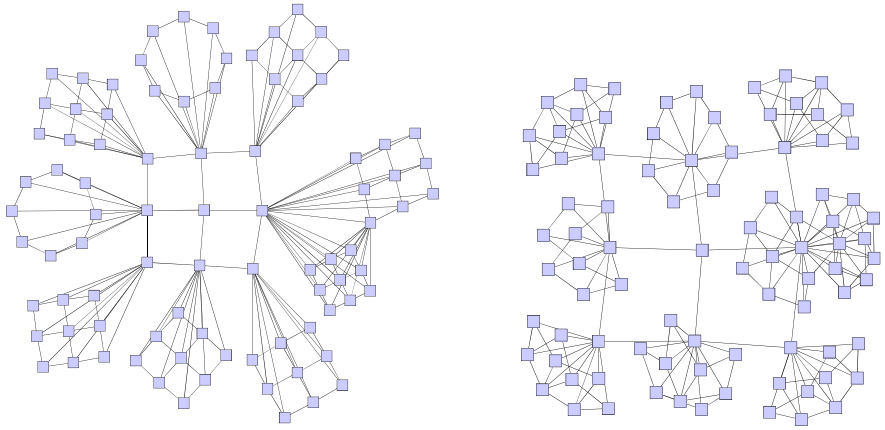
In Figure 8 we present a drawing of a graph known as *cycle of cliques* [3]. Module_Drawing manages to expose all the cliques in an aesthetic pleasant way. Also notice that Draw_Prime algorithm is applied on a graph on 14 vertices (twice the number of the vertices of the basic cycle), whereas a classical spring embedder algorithm should take 56 vertices into account.

In the next two figures (Figures 9-10) we present graphs that contain only neighborhood modules. In general, the output of Module_Drawing is similar to this of SOL, since both employ spring embedder algorithms. More specifically in Figure 9 (resp. Figure 10) an underlying grid (resp. path) structure is revealed. Nevertheless, only our algorithm manages to expose the rest of the structure hidden in the graph (smaller grids, circles, paths e.t.c). This observation arises from the fact that we apply a spring embedder algorithm without the force impact of the vertices that belong to other modules.

Another interesting example can be obtained when two subgraphs are completely connected to each other. More precisely, in Figure 11(a) we distinguish two cycles both on 10 vertices, where each vertex of the one cycle is adjacent to every vertex of the other. Clearly, this structure is vanished in Figure 11(b).

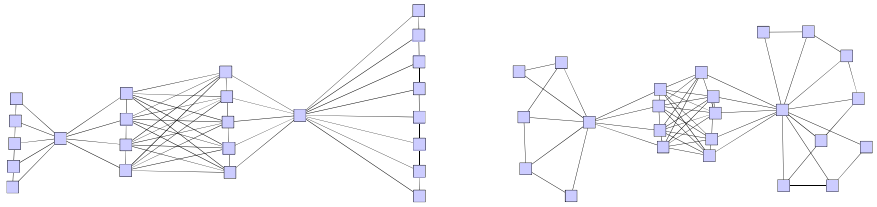
The last example is a graph with an md-tree of 3 levels. We present the output of our method in Figure 12(a). Notice that our method reveals three underlying structures: a gear graph[2], an $\mathcal{A}$-shaped graph and a complex of grids. These structures are even more obvious if one looks at the level-by-level drawings of our method. In Figure 13, we show the md-tree of the graph with

[2]A gear graph is a wheel graph with a vertex added between each pair of adjacent vertices of the outer cycle.



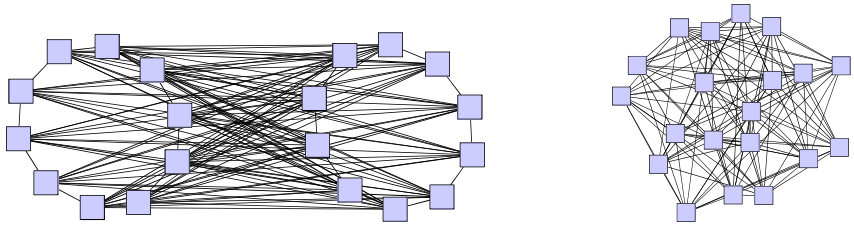
(a) (b)

Figure 9: Drawings of a graph using (a) Module_Drawing and (b) Smart Organic Layout.



(a) (b)

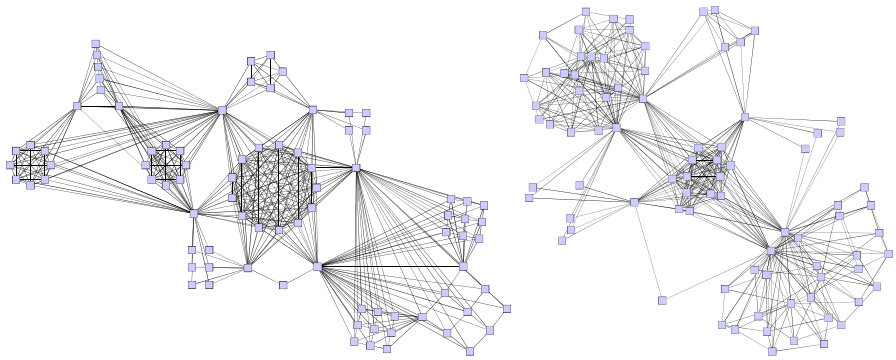
Figure 10: Drawings of a graph using (a) Module_Drawing and (b) Smart Organic Layout.



(a) (b)

Figure 11: Drawings of a graph using (a) Module_Drawing and (b) Smart Organic Layout.

all the intermediate drawings computed by our method. It is useful to consider



(a) (b)

Figure 12: Drawings of a graph using (a) Module_Drawing and (b) Smart Organic Layout.

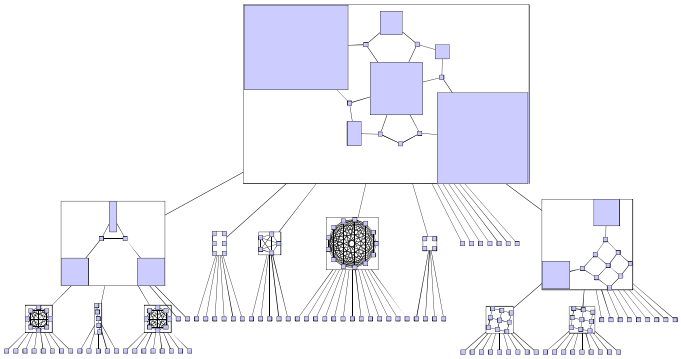


Figure 13: The md-tree of the graph depicted in Figure 12.

this kind of representation as a visualization abstraction of the input graph.

7 An Alternative Approach

As we noted above in Figures 6(a) and 7(a) there are some cases in which the output drawing computed by our algorithm has many vertex-to-edge crossings so that certain adjacencies between vertices are hidden. This is because in each intermediate drawing $\Gamma(t, G)$ the algorithm disregards edges outside $G(t)$ that will play role in a later step. For the complete and prime representative graphs it makes sense to keep them apart for the rest of the vertices since they have structural significance within the graph. However for the case of edgeless representative graphs (parallel modules) the reduction of their vertex-to-edge crossings may compromise criteria C2 and C3. Here we propose an alternative approach that tries to reduce the vertex-to-edge crossings by slightly modifying

Module_Drawing.

The basic idea is to consider forthcoming edges for parallel modules caused by their grandparents in $T(G)$. Recall that the parent of a parallel module is a P-node and, thus, its grandparent, if any, must be an S- or an N-node, since no P-node has a child another P-node in $T(G)$. If the parent of a P-node is an S-node then instead of placing all the modules of the S-node in a circular way, we choose a parallel module with the maximum number of connected components to be drawn in another cycle outside the one corresponding to the S-node. The placement of the connected components is done in an arbitrary order. If the parent of a P-node is an N-node then the children of the P-node are considered as part of the representative graph of the N-node. Thus with the new approach the spring embedder acts on every vertex of the P-node separately, whereas with the previous algorithm those vertices are considered as a unified vertex in the prime graph.

For that purpose before computing the intermediate drawings we modify $T(G)$ and denote the output tree by $T^*(G)$. In the modified tree $T^*(G)$ we have four types of internal nodes, instead of three types in $T(G)$. Let t be an internal node of $T(G)$ and let $ch(t) = \{t_1, \ldots, t_q\}$ be its children in $T(G)$. We distinguish between the following two cases:

(a) If t is an S-node then let $t_i \in ch(t)$ be the P-node having the maximum number of children in $T(G)$ among the other P-nodes in $ch(t)$. We remove the subtree rooted at t_i from $T(G)$ and replace t_i's label by S*. The new parent of t becomes node t_i and the old parent of t now becomes parent of t_i. That is, in $T^*(G)$ the children of t_i are $\{t\} \cup ch(t_i)$ and t_i's parent is t_i's grandparent in $T(G)$. In the resulting tree $T^*(G)$ node t_i marks its child t.

(b) If t is an N-node then for every P-node $t_i \in ch(t)$ all the children of t_i in $T(G)$ become children of t in $T^*(G)$ and node t_i is removed.

Observe that for the N-nodes we modify all of their children labelled as P-nodes, whereas for the S-nodes we consider at most one child labelled as a P-node. In Figure 14 we show the modified tree $T^*(G)$ of the graph shown in Figure 1. Notice that node t_2^* represents node t_4 in $T(G)$ and has label S* in $T^*(G)$. The number of children of t_3^* is increased by two with respect to that of t_3, since two of t_3's children in $T(G)$ are P-nodes. By the construction of $T^*(G)$ it follows that the representative graphs of the S*- and N-nodes have the following properties. If t is an S*-node then $G(t)$ is a star graph[3] where the central vertex (the vertex of degree more than one) corresponds to the parent of t in $T(G)$. If t is an N-node then $G(t)$ is not necessarily a prime graph, since it contains modules coming from the P-nodes in $T(G)$. Moreover note that an S-node in $T^*(G)$ may have only one child.

Given the modified tree $T^*(G)$ of a graph G, we need to reconsider Module_Drawing in order to obtain the alternative drawing of G. Observe first that

[3] A graph on n vertices is called *star* if it is connected and $n-1$ vertices have degree 1.

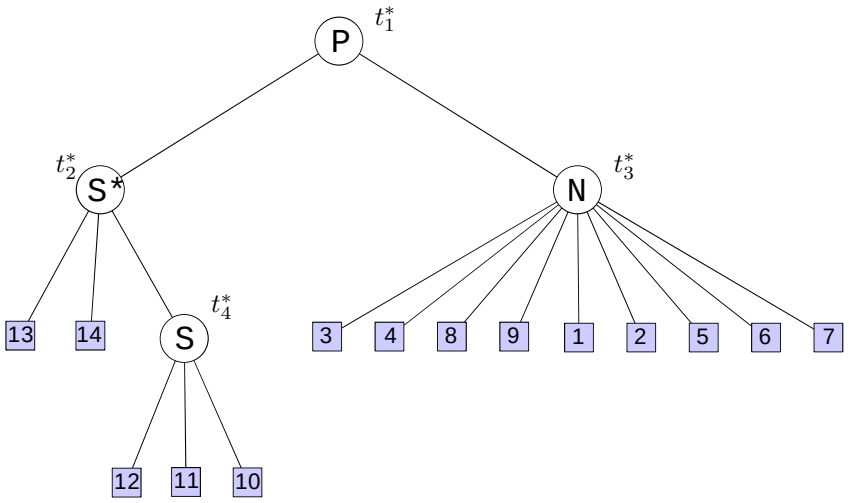
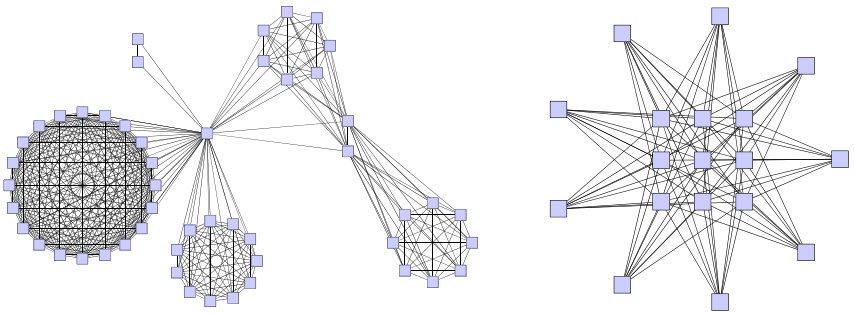


Figure 14: The modified tree $T^*(G)$ of the graph depicted in Figure 1.

for P-, S-, and N-nodes we apply the same corresponding functions as in the previous case. The difference now in $T^*(G)$ is that we have a new type of internal node t labelled by S* so that $G(t)$ is a star graph. Its drawing results by placing the central vertex in the center of a cycle and the rest of the vertices are placed on the circumference. Recall that the vertices of $G(t)$ have non-uniform sizes. There is a straightforward way to obtain such a placement by first applying Draw_Complete on the vertices of $G(t)$ excluding the central vertex and, then, place the central vertex on the center of the cycle. Caution is required so that the central vertex does not cause any overlapping with the rest of the vertices placed on the circumference. Thus before applying Draw_Complete we increase the preferred edge length by a proportional function of the diagonal length of the central box. By that way we guarantee that no vertex overlapping occurs in the relative drawing of $G(t)$. It is interesting to note that the main difference with the previous version is that we introduce a new type of label in the modified tree that requires the proper description of the corresponding relative drawing technique.

Regarding the running time observe that modifying the tree $T(G)$ takes $O(n)$ time since both trees $T(G)$ and $T^*(G)$ contain $O(n)$ nodes. The function for drawing star graphs for the S*-nodes requires the same amount of time needed for applying Draw_Complete. However the vertex set of a representative graph of an N-node in $T^*(G)$ may be larger than the corresponding one in $T(G)$, implying an additional cost to the overall running time.

A drawing of the Trans graph computed by the second approach is shown in Figure 15 (a). Notice that the spring embedder is able to distinguish the vertices of the parallel module shown in Figure 6 in such a way that the adjacencies between those vertices and the simple vertex are easy to follow. In Figure 15 (b)



(a) (b)

Figure 15: Two drawings computed by the second approach. (a) and (b) contain the graphs shown in Figures 6 and 7, respectively.

we present the drawing computed by the alternative algorithm for the complete bipartite graph shown in Figure 7. Observe that although the vertex-to-edge crossings are not minimized in the output drawing, there are no such crossings between the vertices of the parallel module placed in the outer cycle.

8 Concluding Remarks

In this paper we have presented a divide-and-conquer technique for drawing undirected graphs, based on their modular decomposition tree, where each disjoint induced subgraph (module) is drawn according to its corresponding structure (edgeless, complete or prime). For certain classes of graphs, the structure of their modular decomposition trees ensures that each tree node can be processed in linear time. It turns out that our algorithm also reveals the structure of a graph and exposes regular structures of its subgraphs. Thus, modular decomposition can be thought as a quite effective clustering technique that can be exploited to give a better insight in many graphs. By grouping vertices with the same neighborhood a great reduction on the size of the graph can be achieved, without loosing any information for the connectivity among modules.

It is interesting to design efficient drawing algorithms for certain classes of prime graphs, by exploiting their structural properties. In this way the application of the spring embedder, which is the most expensive part of our algorithm, will be avoided. Also, further research includes an investigation for the edge crossings which appear between modules. It would be interesting to know if the adjacency between two modules of vertices can be drawn in a planar way by applying confluent drawings introduced in [10]. Finally, another interesting point is to apply other linear-time hierarchical decomposition algorithms, such as split decomposition which divides a connected graph into stars, cliques and prime graphs [7].

Acknowledgements

The authors would like to express their thanks to the anonymous referees whose suggestions helped improve the presentation of the paper.

References

[1] H. Bodlaender and U. Rotics. Computing the treewidth and the minimum fill-in with the modular decomposition. *Algorithmica*, 36:375–408, 2003.

[2] F. Brandenburg. Designing graph drawings by layout graph grammars. In *Proc. 2nd Int. Symp. Graph Drawing (GD'94)*, volume 894 of *Lecture Notes in Computer Science*, pages 416–427, 1994.

[3] F. Brandenburg. Graph clustering 1: circles of cliques. In *Proc. 5th Int. Symp. Graph Drawing (GD'97)*, volume 1353 of *Lecture Notes in Computer Science*, pages 158–168, 1997.

[4] U. Brandes, M. Eiglsperger, I. Herman, M. Himsolt, and M. Marshall. Graphml progress report: structural layer proposal. In *Proc. 9th Int. Symp. Graph Drawing (GD'01)*, volume 2265 of *Lecture Notes in Computer Science*, pages 501–512, 2001.

[5] A. Brandstädt, V. Le, and J. Spinrad. *Graph Classes: A Survey*. SIAM Monographs on Discrete Mathematics and Applications, 1999.

[6] B. Courcelle, J. Makowsky, and U. Rotics. Linear time solvable optimization problems on graphs of bounded clique-width. *Theory Comput. Syst.*, 33:125–150, 2000.

[7] E. Dahlhaus. Parallel algorithms for hierarchical clustering and applications to split decomposition and parity graph recognition. *J. Algorithms*, 36:205–240, 2000.

[8] E. Dahlhaus, J. Gustedt, and R. McConnell. Efficient and practical algorithms for sequential modular decomposition. *J. Algorithms*, 41:360–387, 2001.

[9] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Algorithms for the Visualization of Graphs*. Prentice-Hall, 1999.

[10] M. Dickerson, D. Eppstein, M. Goodrich, and J. Meng. Confluent drawings: visualizing non-planar diagrams in a planar way. In *Proc. 11th Int. Symp. Graph Drawing (GD'03)*, volume 2912 of *Lecture Notes in Computer Science*, pages 1–12, 2004.

[11] P. Eades and Q. Feng. Drawing clustered graphs on an orthogonal grid. In *Proc. 5th Int. Symp. Graph Drawing (GD'97)*, volume 1353 of *Lecture Notes in Computer Science*, pages 146–157, 1997.

[12] P. Eades, Q. Feng, and X. Lin. Straight-line drawing algorithms for hierarchical graphs and clustered graphs. In *Proc. 4th Int. Symp. Graph Drawing (GD'96)*, volume 1190 of *Lecture Notes in Computer Science*, pages 113–128, 1996.

[13] P. Eades, W. Lai, K. Misue, and K. Sugiyama. Layout adjustment and the mental map. *Journal of Visual Languages and Computing*, 6:379–405, 1995.

[14] A. Ehrenfeucht, H. Gabow, R. McConnell, and S. Sullivan. An $o(n^2)$ divide and conquer algorithm for the prime tree decompositon of two structures and modular decomposition of graphs. *J. Algorithms*, 16:283–294, 1994.

[15] Q.-W. Feng, R. F. Cohen, and P. Eades. Planarity for clustered graphs. In *Proc. 3rd European Symp. Algorithms (ESA'95)*, volume 979 of *Lecture Notes in Computer Science*, pages 213–226, 1995.

[16] I. Finocchi. *Hierarchical Decompositions for Visualizing Large Graphs*. PhD thesis, Università degli Studi di Roma "La Sapienza", 2001.

[17] A. Frick, A. Ludwig, and H. Mehldau. A fast adaptive layout algorithm for undirected graphs. In *Proc. 2nd Int. Symp. Graph Drawing (GD'94)*, volume 894 of *Lecture Notes in Computer Science*, pages 388–403, 1994.

[18] T. Fruchterman and E. Reingold. Graph drawing by force-directed placement. *Software-Practice and Experience*, 21:1129–1164, 1991.

[19] J. Gagneur, R. Krause, T. Bouwmeester, and G. Casari. Modular decomposition of protein-protein interaction networks. *Genome Biology*, 5::R57, 2004.

[20] T. Gallai. Transitiv orientierbare graphen. *Acta Math. Acad. Sci. Hungar.*, 18:25–66, 1967.

[21] E. R. Gansner and S. C. North. Improved force-directed layouts. In *Proc. 6th Int. Symp. Graph Drawing (GD'98)*, volume 1547 of *Lecture Notes in Computer Science*, pages 364–373, 1998.

[22] D. Harel and Y. Koren. Drawing graphs with non-uniform vertices. In *Proc. of Working Conference on Advanced Visual Interfaces (AVI'02)*, ACM Press, pages 157–166, 2002.

[23] K. Hayashi, M. Inoue, T. Masuzawa, and H. Fujiwara. A layout adjustment problem for dijsoint rectangles preserving orthogonal order. In *Proc. 6th Int. Symp. Graph Drawing (GD'98)*, volume 1547 of *Lecture Notes in Computer Science*, pages 183–197, 1998.

[24] M. Huang and P. Eades. A fully animated interactive system for clustering and navigating huge graphs. In *Proc. 6th Int. Symp. Graph Drawing (GD'98)*, volume 1547 of *Lecture Notes in Computer Science*, pages 374–383, 1998.

[25] M. Kaufmann and D. Wagner, editors. *Drawing Graphs: Methods and Models*, volume 2025 of *Lecture Notes in Computer Science*. Springer Verlag, 2001.

[26] T. Lengauer. *Combinatorial Algorithms for integrated Circuit Layout*. Wiley-Teubner Series, 1990.

[27] T. Li and P. Eades. Integration of declarative and algorithmic approaches for layout creation. In *Proc. 2nd Int. Symp. Graph Drawing (GD'94)*, volume 894 of *Lecture Notes in Computer Science*, pages 376–386, 1994.

[28] W. Li, P. Eades, and N. Nikolov. Using spring algorithms to remove node overlapping. In *Proc. Asia Pacific Symp. Information Visualization (APVIS'05)*, 2005.

[29] K. Marriott, P. Stuckey, V. Tam, and W. He. Removing node overlapping in graph layout using constrained optimization. *Constraints*, 8:143–172, 2003.

[30] R. McConnell and F. Montgolfier. Linear-time modular decomposition of directed graphs. *Discrete Applied Math.*, 145:189–209, 2005.

[31] R. McConnell and J. Spinrad. Modular decomposition and transitive orientation. *Discrete Math.*, 201:189–241, 2001.

[32] J. Muller and J. Spinrad. Incremental modular decomposition. *J. ACM*, 36:1–19, 1989.

[33] B. Preiss. *Data Structures and Algorithms with Object-Oriented Design Patterns in C++*. John Wiley & Sons, 1998.

[34] F.-S. Shieh and C. McCreary. Directed graph drawing using clan based decomposition. In *Proc. 3rd Int. Symp. Graph Drawing (GD'95)*, volume 1027 of *Lecture Notes in Computer Science*, pages 472–482, 1995.

[35] F.-S. Shieh and C. McCreary. Clan-based incremental drawing. In *Proc. 8th Int. Symp. Graph Drawing (GD'00)*, volume 1984 of *Lecture Notes in Computer Science*, pages 384–395, 2000.

[36] J. Six and I. G. Tollis. A framework for user-grouped circular drawings. In *Proc. 11th Int. Symp. Graph Drawing (GD'03)*, volume 2912 of *Lecture Notes in Computer Science*, pages 135–146, 2003.

[37] D. Tunkelang. *A Numerical Optimization Approach to General Graph Drawing*. PhD thesis, Carnegie Mellon University, 1999.

[38] C. Walshaw. A multilevel algorithm for force-directed graph drawing. *J. Graph Algorithms Appl.*, 7:253–285, 2003.

[39] X. Wang and I. Miyamoto. Generating customized layouts. In *Proc. 3rd Int. Symp. Graph Drawing (GD'95)*, volume 1027 of *Lecture Notes in Computer Science*, pages 504–515, 1995.

[40] yEd Java Graph Editor. http://www.yworks.com.