Drawing Graphs using Modular Decomposition

Charis Papadopoulos[1] and Constantinos Voglis[2]

[1] Department of Informatics, University of Bergen, N-5020 Bergen, Norway
charis@ii.uib.no
[2] Department of Computer Science, University of Ioannina, P.O.Box 1186, GR-45110 Ioannina, Greece voglis@cs.uoi.gr

Abstract. In this paper we present an algorithm for drawing an undirected graph G which 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.

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 [1]. In this work we present a drawing algorithm which 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 [4].

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.

Similar approaches for computing the layout of a graph are based on a specific decomposition of it. Based on this scheme, optimal algorithms have been developed for drawing a series-parallel digraph [1], and for upward planarity testing of a single-source digraph [2]. Also, many techniques for drawing hierarchical clustered graphs, deal with a graph and its tree representation [6–8]. All these methods address the problem of visualization, by drawing the non-leaf nodes of the tree as simple closed curves. Force directed methods have also been developed to support and show the structure of a clustered graph which is a 2-level decomposition scheme [13, 18].

2 Definitions and Background Results

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; a *stable set* is a set of pairwise non-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 $0 \leq i \leq h$.

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 [4]. 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. A module M of a graph G is called *parallel* if $G[M]$ is a disconnected graph, *series* if $\overline{G}[M]$ is a disconnected graph and *neighborhood* if both $G[M]$ and $\overline{G}[M]$ are connected graphs.

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 [5, 15].

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, the following lemma holds (see [4]):

Lemma 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.*

2.2 Modular Decomposition Based Drawing $\Gamma(G)$

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 $G(t)$.

Definition 1. *A drawing with the previous constraints is called a* modular decomposition based drawing $\Gamma(G)$ *of the graph G which is a mapping between the vertices and the Euclidean space $\mathbf{R}^2$: $\Gamma(G) : V(G) \rightarrow \mathbf{R}^2$.*

Definition 2. *A* relative drawing $\Gamma'(t, T(G))$ *is an md-drawing of the representative graph $G(t)$, relative to $c(t)$.*

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 [5, 15].

In bottom-up fashion, we traverse the md-tree $T(G)$ and calculate the relative drawing $\Gamma'(t,T)$ for every internal node t. 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. 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.

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 $T(G)$. The algorithm, called *Module_Drawing*, is given in detail in Algorithm 1.

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 rectangle boxes $b(t)$ and the centers $c(t)$ for every $t \in T$;
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 a S-node **then** $\Gamma'(t,T) \leftarrow$ *Draw_Complete*(t,T);
   4.3 **else** {*t is a 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 their initial 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**
   5.1 $c(t_i) \leftarrow c(t_i) + dis(t)$
6. Return the drawing $\Gamma(G) = \Gamma'(r,T)$ computed in the root r of T;

Algorithm 1: *Module_Drawing*.

Due to lack of space, the formal description of functions *Draw_Edgeless* and *Draw_Complete* is omitted, whereas the function *Draw-Prime* is described in detail in Sect. 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.

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. 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.

Function Draw_Complete is basically a circular drawing algorithm, even though 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. 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. Function Draw_Complete process each node $t_i \in ch(t)$ one by one, and calculates its final radius by considering the size of the two adjacent nodes on the cycle. For every node t_i a value $f(t_i)$ is computed that represents the maximum distance from $c(t_i)$ to a point on its boundary $b(t_i)$. Finally, node t_i is positioned on the minimum possible radius, according to $f(t_i)$ and the preferred edge length k, so that any overlapping is avoided. We note that for a complete graph with uniform nodes the drawing is a perfect cycle.

For the time complexity of functions Draw_Edgeless and Draw_Complete, the following holds:

Lemma 2. *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 a 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 [14, 12, 11]. It is based on the Fruchterman & Reingold (FR) spring embedder algorithm [9] and follows the general guidelines of Harel & Koren [12]. 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 [9], 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 formulas for the forces will be presented later in the section. We will first describe the heuristics that we use to avoid overlapping. According to [12], the first modification to the original FR algorithm will result the following formulas for the attractive f_a and the repulsive f_r forces:

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

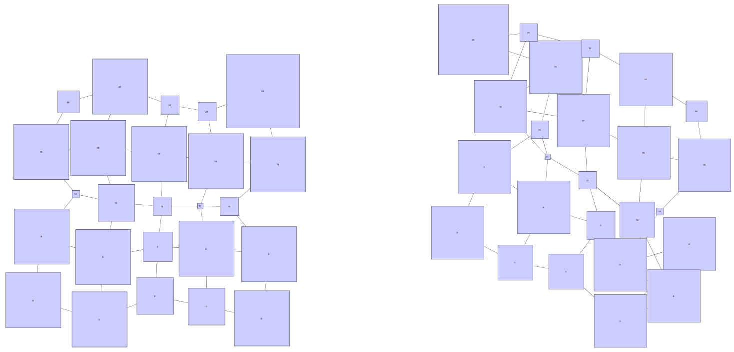
where $r_{MFR} = f(t_i, t_j)$ and $f(t_i, t_j)$ is the shortest distance between the boundaries of the boxes $b(t_i)$ and $b(t_j)$. The variable k is the preferred edge length for the drawing and ϵ is a small positive number.

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 [12]).

We noticed 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 decide to use a factor w in the calculation of the edge attractive forces, inversely proportional to the graph's density. In this manner, we weaken edge attractive forces and allow 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 graph G we have: $0 < D^{-1}(G) \leq 0.57$.

Using the previous inequality of $D^{-1}(G)$, we set a threshold in the middle of the interval and consider dense the graphs G s.t. $D^{-1}(G) < 0.28$ and sparse the



(a) (b)

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

graphs s.t. $D^{-1}(G) > 0.28$. If a graph is considered sparse, after a certain point in the algorithm we use $D(G)$ as the multiplicative factor.

In Fig. 1 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 Fig. 1(a) the factor $w = D^{-1}(G) = 0.31$ is used, in the early iterations, for the calculation of the attractive forces. Since the graph is considered sparse, this factor is reversed ($w = D(G)$) at final iterations and so the layout becomes more compact. In Fig. 1(b) the multiplicative factor w is set to one in all iterations.

Having describe the two main features of our spring embedder algorithm, we can present the attractive and repulsive forces of function Draw_Prime (DP) as follows:

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

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

$$\text{and} \qquad w = \begin{cases} D^{-1}(G), & \text{at early iterations} \\ D(G), & \begin{array}{l}\text{at final iterations, and} \\ \text{if } \; D^{-1}(G) > 0.28.\end{array} \end{cases}$$

We mention that the early and the final iterations coincide with the first and the second part of the cooling schedule, respectively. We denote by ℓ the number of the main iterations needed by our spring embedder algorithm. We conclude with the following lemma.

Lemma 3. *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.*

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 $T(G)$.

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 [4]. 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[3] contains at most five children [4]. 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 [4].

Theorem 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.*

6 Implementation and Examples

We have implemented our algorithm in C++. The implementation takes as input an undirected graph G in GraphML format [3]. 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 $T(G)$; a file that contains all the relative drawings computed in each level of $T(G)$. For visualization purposes, we use the yEd environment [16].

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 [10] for details). More specifically, the input graph, which we will call *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. We label the 51 vertices of the graph and assign an additional label, besides P or S or N label, to the 8 internal nodes of the md-tree. In Fig. 2(a) we present the final drawing of Trans graph using Module_Drawing, in Fig. 2(b) we show its modular decomposition tree and in Fig. 2(c) we present level-by-level relative drawings and how they are combined to result the final layout.

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

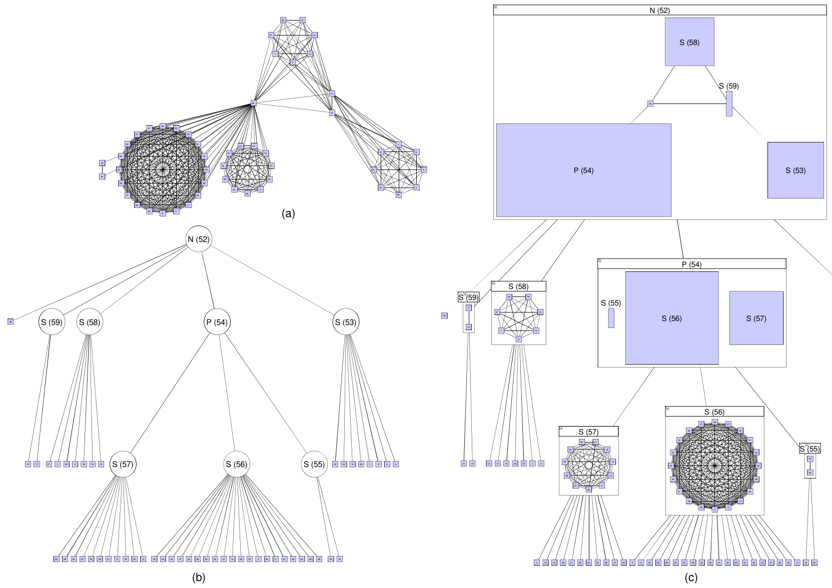


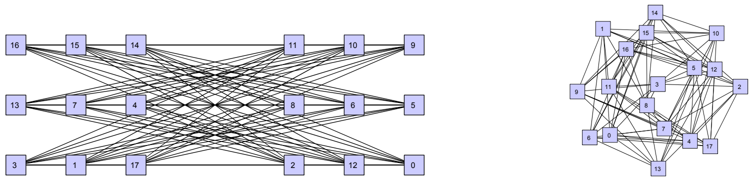
Fig. 2: Illustration of Module_Drawing on *Trans* graph

Starting from level 3 of the tree in Fig. 2(c), we notice three S-nodes. The application of the function Draw_Series results 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.

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 [17].

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 [16] 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

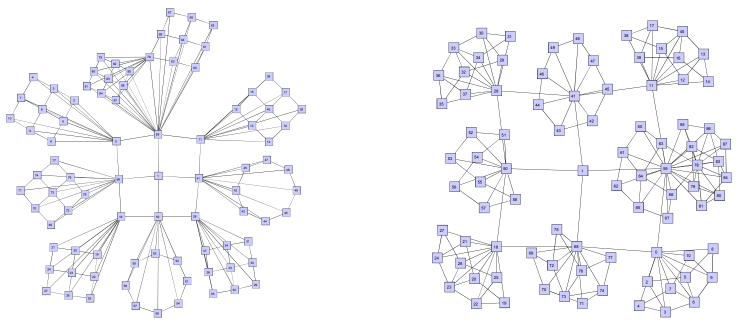


(a) (b)

Fig. 3: Drawings of $K_{9,9}$ using (a) Module_Drawing and (b) Smart Organic Layout.

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.

In Figs. 3–5 the final drawings of our algorithm are shown on the left side whereas the drawings of the same graph using SOL are shown on the right side. Notice that our algorithm manage to expose underlying structures (smaller grids, circles, paths e.t.c) in all the examples. 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.

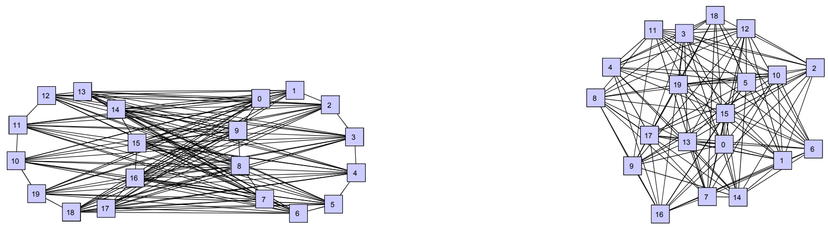


(a) (b)

Fig. 4: Drawings of a graph using (a) Module_Drawing and (b) Smart Organic Layout.

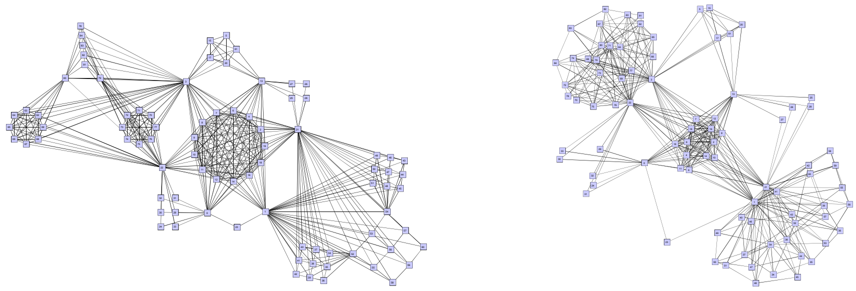
In Fig. 6 we show a graph with an md-tree of 3 levels. Notice that our method reveals three underlying structures: a gear graph[4], an $\mathcal{A}$-shaped graph and a complex of grids. In Fig. 7, we show the md-tree of the graph, in order to illustrate the intermediate steps of our method. It is useful to consider the md-tree representation, as a visualization abstraction of the input graph.

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



(a) (b)

Fig. 5: Drawings of a graph using (a) Module_Drawing and (b) Smart Organic Layout.



(a) (b)

Fig. 6: Drawings of a graph using (a) Module_Drawing and (b) Smart Organic Layout.

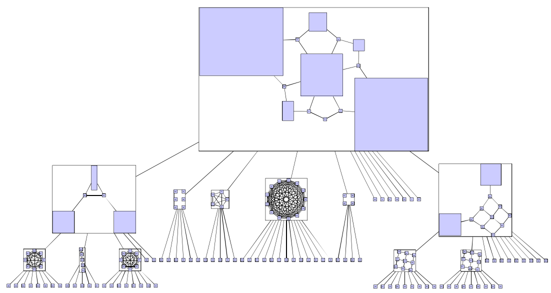


Fig. 7: The md-tree of the graph depicted in Fig. 6.

7 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, besides its efficiency in terms of time, also exposes the structure of a graph. Revealing the structure of a graph by drawing it, can prove to be helpful in identifying, and thus, recognizing, in which certain class the graph belongs.

References

1. G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis, *Algorithms for the Visualization of Graphs*, Prentice-Hall, 1999.
2. P. Bertolazzi, G. Di Battista, C. Mannino, and R. Tamassia, Optimal upward planarity testing of single-source digraphs, *SIAM J. Comput.* **27** (1998) 132–169.
3. U. Brandes, M. Eiglsperger, I. Herman, M. Himsolt, and M.S. Marshall: GraphML progress report: structural layer proposal, *Proc. 9th Int. Symp. Graph Drawing (GD'01)*, LNCS **2265** (2001) 501–512.
4. A. Brandstädt, V.B. Le, and J.P. Spinrad, *Graph Classes: A Survey*, SIAM Monographs on Discrete Mathematics and Applications, 1999.
5. E. Dahlhaus, J. Gustedt, and R.M. McConnell, Efficient and practical algorithms for sequential modular decomposition, *J. Algorithms* **41** (2001) 360–387.
6. P. Eades and Q.W Feng, Drawing clustered graphs on an orthogonal grid. *Proc. 5th Int. Symp. Graph Drawing (GD'97)*, LNCS **1353** (1997) 146ů-157.
7. P. Eades, Q.W. Feng, and X Lin, Straight-line drawing algorithms for hierarchical graphs and clustered graphs, *Proc. 4th Int. Symp. Graph Drawing (GD'96)*, LNCS **1190** (1996) 113-ů128.
8. Q.-W. Feng, R. F. Cohen, and P. Eades, Planarity for clustered graphs. *Proc. 3rd European Symp. Algorithms (ESA'95)*, LNCS **979** (1995) 213-ů226.
9. T. Fruchterman and E. Reingold, Graph drawing by force-directed placement, *Software-Practice and Experience*, **21** (1991) 1129–1164.
10. J. Gagneur, R. Krause, T. Bouwmeester, and G. Casari, Modular decomposition of protein-protein interaction networks, *Genome Biology* **5**:R57 (2004).
11. E. R. Gansner and S. C. North, Improved force-directed layouts, *Proc. 6th Int. Symp. Graph Drawing (GD'98)*, LNCS **1547** (1998) 364–373.
12. D. Harel and Y. Koren, Drawing graphs with non-uniform vertices, *Proc. of Working Conference on Advanced Visual Interfaces (AVI'02)*, ACM Press 2002, 157–166.
13. M.L. Huang and P. Eades, A fully animated interactive system for clustering and navigating huge graphs, *Proc. 6th Int. Symp. Graph Drawing (GD'98)*, LNCS **1547** (1998) 374ů-383.
14. W. Li, P. Eades, and N. Nikolov, Using spring algorithms to remove node overlapping, *Proc. Asia Pacific Symp. Information Visualization (APVIS'05)*, 2005.
15. R.M. McConnell and J. Spinrad, Modular decomposition and transitive orientation, *Discrete Math.* **201** (1999) 189–241.
16. yEd - Java Graph Editor, `http://www.yworks.com/en/products_yed_about.htm`.
17. C. Walshaw, A multilevel algorithm for force-directed graph drawing, *J. Graph Algorithms Appl.* **7** (2003) 253–285.
18. X. Wang and I. Miyamoto, Generating customized layouts, *Proc. 3rd Int. Symp. Graph Drawing (GD'95)*, LNCS **1027** (1995) 504–515.