Hierarchical Clustering of Trees: Algorithms and Experiments*

Irene Finocchi and Rossella Petreschi

Dipartimento di Scienze dell'Informazione,
Università di Roma "La Sapienza", Via Salaria 113, 00198 Roma, Italy.
{finocchi, petreschi}@dsi.uniroma1.it

Abstract. We focus on the problem of experimentally evaluating the quality of hierarchical decompositions of trees with respect to criteria relevant in graph drawing applications. We suggest a new family of tree clustering algorithms based on the notion of t-divider and we empirically show the relevance of this concept as a generalization of the ideas of centroid and separator. We compare the t-divider based algorithms with two well-known clustering strategies suitably adapted to work on trees. The experiments analyze how the performances of the algorithms are affected by structural properties of the input tree, such as degree and balancing, and give insight in the choice of the algorithm to be used on a given tree instance.

1 Introduction

Readable drawings of graphs are a helpful visual support in many applications, as they usually convey information about the structure and the properties of a graph more quickly and effectively than a textual description. This led in the last years to the design of several algorithms for producing aestethically pleasing layouts of graphs and networks, according to different drawing conventions and aestethic criteria [7]. As the amount of data processed by real applications increases, many realistic problem instances consist of very large graphs and suitable tools for visualizing them are more and more necessary. Actually, standard visualization algorithms and techniques do not scale up well, not only for efficiency reasons, but mostly due to the finite resolution of the screen, which represents a physical limitation on the size of the graphs that can be successfully displayed.

Common approaches to overcome this problem and to make it possible visualizing graphs not fitting in the screen exploit clustering information [2,9,10,16] and/or navigation strategies [1,6,17]. In particular, the *clustering approach* consists of using recursion combined with graph decomposition in order to build a sequence of meta-graphs from the original graph G: informally speaking, a meta-graph in the sequence is a "summary" of the previous one; the first and more detailed meta-graph coincides with G itself. To grow a meta-graph, suitably chosen sets of related vertices are grouped together to form a *cluster* and induced edges between clusters are computed. Clusters are then represented as

* Work partially supported by the project "Algorithms for Large Data Sets: Science and Engineering" of the Italian Ministry of University and of Scientific and Technological Research (MURST 40%).

A.L. Buchsbaum and J. Snoeyink (Eds.): ALENEX 2001, LNCS 2153, pp. 117–131, 2001.

single vertices or filled-in regions, thus obtaining a high-level visualization of the graph while reducing the amount of displayed information by hiding irrelevant details and uninteresting parts of the structure. There is no universally accepted definition of cluster: usual trends consist of grouping vertices which either share semantic information (e.g., the space of IP addresses in a telecommunication network) or reflect structural properties of the original graph [19,21,22].

The sequence of meta-graphs defines a hierarchical decomposition of the original graph G which can be well represented by means of a rooted tree, known in literature as *hierarchy tree* [3], whose leaves are the vertices of G and whose internal nodes are the clusters. In order to grow a hierarchy tree out of a graph G, a simple top-down strategy works as follows: starting from the root of the hierarchy tree, which is a high-level representation of the whole graph, at the first iteration the vertices of G are partitioned by a clustering algorithm and the clusters children of the root (at least 2) are generated. The procedure is then recursively applied on all these clusters. It is obvious that very different hierarchy trees can be associated to the same graph, depending on the clustering algorithm used as a subroutine. Suitable selections of nodes of a hierarchy tree lead to different high-level representations (*views*) of G: the user of the visualization facility can then navigate from a representation to another one by shrinking and expanding clusters in the current view. Due to the fact that hierarchy trees are interactively visited, they should fulfil some requirements that are relevant from a graph drawing perspective. First of all, any view obtained from a hierarchy tree of a graph G should reflect the topological structure of G so as not to mislead the viewer. Furthermore, structural properties such as limited degree, small depth and balancing of the hierarchy tree greatly help in preserving the mental map of the viewer during the navigation and make it possible to provide him/her with several different views of the same graph.

The main goal of this paper is to experimentally evaluate with respect to the aforementioned criteria the quality of hierarchy trees built using different clustering subroutines. More precisely, in our study we are concerned with hierarchical decompositions of trees, not only because such structures frequently arise in many practical problems (e.g., evolutionary or parse trees), but also because tree clustering procedures represent an important subroutine for partitioning generic graphs (for example, they could be applied to the block-cut-vertex tree of a graph). As a first contribution, we suggest new tree clustering algorithms hinging upon the notion of t-divider, which generalizes concepts like centroid and separators [4,18,20], and we present three partitioning strategies, each aimed at optimizing different features of the hierarchy tree. We then explore the effectiveness of the t-divider based algorithms for different values of t by comparing them against two well-known clustering procedures due to Gonzalez [14] and Frederickson [12], suitably adapted to work on general trees.

The quality measures considered in the experimentation are defined in agreement with the requirements on the structure of the hierarchy tree arising in graph drawing applications. All the experiments are carried out on structured randomly generated instances, which allow us to study how the performances of the algorithms are affected by parameters such as degree and balancing of the input tree. In particular, looking at the experimental results, we can detect which algorithms are more sensitive to the structure of the input instance and identify hard and easy problem families for some of the analyzed algorithms. The

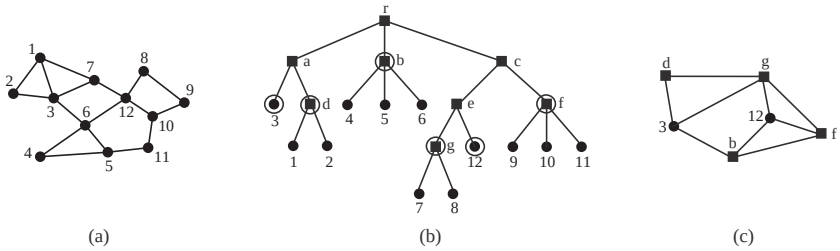

Fig. 1. (a) A graph G; (b) a hierarchy tree of G and a covering C={3,d,b,g,12,f} on it; (c) view of G induced by covering C.

results give insight in the choice of the algorithm to be used when a particular tree instance is given. Furthermore, they provide experimental evidence for the utility of the theoretical notion of t-divider, proving that the use of centroids, that is very frequent in literature, does not usually yield the best solution.

In order to prove the feasibility of the clustering approach for the visualization of large trees we have been also developing a prototype of visualization facility which implements all the algorithms discussed in the paper and supports the navigation of the hierarchy tree as described above. The use of such a prototype gave us visual feedback on the behaviour of the algorithms and was helpful for tuning the code and for testing heuristics to design more efficient variations. The prototype is implemented in ANSI C and visualizations are realized in the algorithm animation system Leonardo [5], available for Macintosh platform at the URL `http://www.dis.uniroma1.it/~demetres/Leonardo/`. The tree clustering package is available at the URL `http://www.dsi.uniroma1.it/~finocchi/experim/treeclust-2.0/`.

In the rest of this paper we first recall preliminary definitions and concepts related to hierarchy trees in Section 2. In Section 3, after introducing the concept of t-divider of a tree, we design three strategies for partitioning trees based on such a concept. Gonzalez's and Frederickson's clustering approaches are summarized in Section 4. Section 5 describes the experimental framework and the results of the experimentation are discussed in Section 6. Conclusions and directions for further research are finally addressed in Section 7.

2 "Attractive" Hierarchy Trees

In this section we recall the definition of hierarchy tree and we discuss the concepts of covering and of view of a graph on a hierarchy tree associated to it [3]. We then point out which properties make a hierarchy tree attractive from a graph visualization point of view, i.e., which requirements should be fulfilled by a hierarchy tree so that graph drawing applications can substantially benefit from its usage.

Definition 1. *A hierarchy tree $HT = (N, A)$ associated to a graph $G = (V, E)$ is a rooted tree whose set of leaves concides with the set of vertices of G.*

Each node $c \in N$ represents a *cluster* of vertices of G, that we call vertices *covered* by c. Namely, each leaf in HT covers a single vertex of G and each

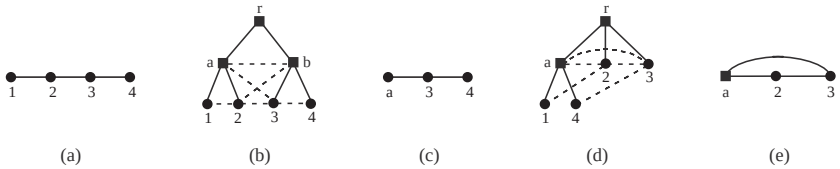

Fig. 2. Validity of hierarchy trees of a 4-vertex chain.

internal node c covers all the vertices covered by its children, i.e., all the leaves in the subtree rooted at c. For brevity, we write $u \prec c$ to indicate that a vertex $u \in V$ is covered by a cluster $c \in N$. The *cardinality* of a cluster c is the number of vertices covered by c; moreover, for any $c \in N$, we denote by $S(c)$ the subgraph of G induced by the vertices covered by c. Two clusters c and c' which are not ancestors of each other are connected by a *link* if there exists at least an edge $e = (u, v) \in E$ such that $u \prec c$ and $v \prec c'$ in HT; if more than one edge of this kind exists, we consider only a single link. We denote by L the set of all such links. Given a subset N' of nodes of HT, the *graph induced by* N' is the graph $G' = (N', L')$, where L' contains all the links of L whose endpoints are in N'. From the above definitions G' contains neither self-loops nor multiple edges.

Definition 2. *Given a hierarchy tree HT of a graph G, a covering of G on HT is a subset C of nodes of HT such that $\forall v \in V \ \exists!\, c \in C : v \prec c$. A view of G on HT is the graph induced by any covering of G on HT.*

Trivial coverings consist of the root of HT and of the whole set of its leaves. Figure 1(a) and Figure 1(b) show a 12-vertex graph and a possible hierarchy tree of it. The internal nodes of the hierarchy tree are squared and, for clarity, no link is reported. A covering consisting of clusters {3,d,b,g,12,f} is highlighted on the hierarchy tree and the corresponding view is given in Figure 1(c).

As this work is concerned with recursive clustering of trees and forests, in the rest of the paper we assume that the graph to be clustered is a free tree $T = (V, E)$ with $n = |V|$ vertices. Under this hypothesis it is natural to require any view of T obtained from HT to be a tree. In [11] HT is said to be *valid* iff it satisfies this property. It is worth observing that not any hierarchy tree associated to a tree is valid. Figure 2(b) and Figure 2(c) show a simple example of a valid hierarchy tree associated to a chain on four vertices and a view of the chain on it, while Figure 2(d) depicts a non-valid hierarchy tree associated to the same chain: the view associated to covering $\{a, 2, 3\}$ contains a cycle, as in Figure 2(e). All the existing links are shown on the hierarchy trees as dotted lines. Due to the importance of the notion of validity for graph drawing purposes, it is natural to restrict the attention on algorithms that generate valid hierarchy trees. In [11] necessary and sufficient conditions for the validity of hierarchy trees are presented; we reveal in advance that all the clustering algorithms described in Section 3 and in Section 4 meet such requirements. For the purposes of this paper it is enough to recall a weaker sufficient condition:

Theorem 1. *[11] Let $T = (V, E)$ be a free tree and let $HT = (N, A)$ be a hierarchy tree of T. Then HT is valid if for each $c \in N$ s.t. $S(c)$ is disconnected*

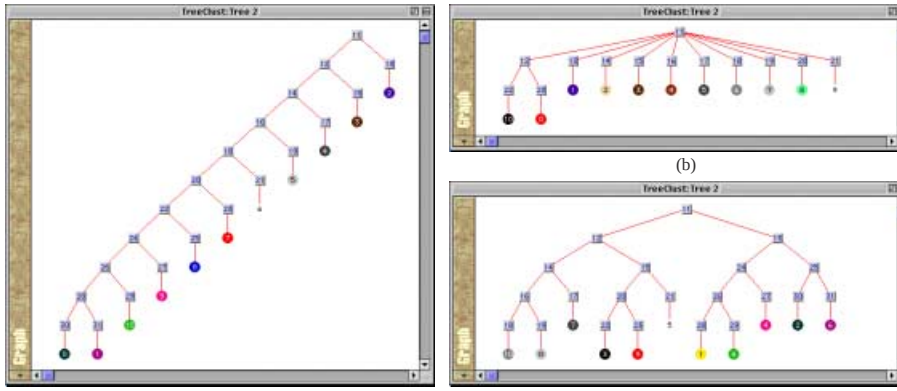

Fig. 3. Attractive and unattractive hierarchy trees associated to a 11-vertex star centered at vertex 0. The trees are grown up by the algorithms presented in Section 3 and in Section 4. Snapshots are produced in the algorithm animation system Leonardo [5].

there exists a vertex $v \in V$, $v \not\prec c$, *such that each connected component of* $S(c)$ *contains a neighbor of* v *in* T.

In addition to validity, we can identify other interesting requirements on the structure of hierarchical decompositions that make them attractive. As an example, let us consider the hierarchy trees in Figure 3; all of them are associated to the same tree (a 11-vertex star centered at vertex 0) and are valid. Let us now suppose that the user of the visualization facility needs a view containing no more than 5 vertices, one of which must be vertex 0. It is easy to see that such a view can be obtained only from the hierarchy tree in Figure 3(c). The main problem with the tree in Figure 3(a) is that it is really deep and unbalanced: any covering of cardinality ≤ 5 can show at the maximum detail only vertices 2, 3, 4, and 5, i.e., the leaves in the highest levels of the hierarchy tree. Similarly, the tree in Figure 3(b) is shallow and has a too large degree: it contains only a view with at most 5 nodes, i.e., the view consisting of its root. This simple example gives some insight on the fact that navigating balanced hierarchy trees with limited degree provides the viewer with better possibilities of interaction with the visualization tool, since a high number of views can be obtained from such trees and preserving the mental map becomes easier. The following structural properties are therefore relevant for hierarchy trees that are going to be used in visualization applications:

Limited degree: the expansion of a cluster should not result in the creation of a high number of new nodes in the view. Indeed, adding all of a sudden many new clusters to the view could imply drastic changes in its drawing.

Small depth: as suggested in [8,9], a logarithmic depth appears to be a reasonable choice. Indeed, traversing the hierarchy tree should not require too much time, but an excessively small depth (e.g., constant) is indicative of big differences between consecutive views.

Balancing: nodes on the same level of the hierarchy tree should have similar cardinalities. In this way, any navigation from the root to a leaf takes approximately the same time, independently of the followed path.

3 Divider-Based Clustering Algorithms

In this section we first introduce the concept of t-divider and then we devise three clustering strategies which hinge upon this concept. From now on we call *rank(T)* the number of vertices in a tree T.

Definition 3. *Given a free tree $T = (V, E)$ with n vertices and a constant $t \geq 2$, a vertex $v \in V$ of degree $d(v)$ is called t-divider if its removal disconnects T into $d(v)$ trees each one containing at most $\lfloor \frac{t-1}{t} n \rfloor$ vertices.*

The concept of t-divider is a natural generalization of that of *centroid* [4] and *separator* [18,20], which are a 2-divider and a 3-divider, respectively. The existence of centroids and the fact that, $\forall\, t_1 \geq t_2$, any t_2-divider is also a t_1-divider, imply the existence of at least a t-divider for any value $t \geq 2$. We remark that t-dividers are not necessarily unique. The following theorem holds:

Theorem 2. *Let $T = (V, E)$ be a free tree and let v be any vertex of V. For any constant $t \geq 2$, if v is not a t-divider, the removal of v disconnects T into k subtrees $T_1 ... T_k$ such that: (a) it exists a unique T_i with more than $\lfloor \frac{t-1}{t} n \rfloor$ vertices; (b) any t-divider of T belongs to T_i.*

Based on Theorem 2, an efficient algorithm for finding a t-divider of a tree starts from any vertex v verifying if v is a t-divider. If so, it stops and returns v. Otherwise, it iterates on the subtree of maximum rank T_i obtained by the removal of v; the new iteration starts from the vertex $w \in T_i$ adjacent to v.

The concept of t-divider can be easily extended from trees to forests. We say a forest $\mathcal{F}$ to be *t-divided* if all its trees have rank $\leq \lfloor \frac{t-1}{t} n \rfloor$. Given a non t-divided forest $\mathcal{F}$, we can t-divide it by removing a single vertex, called t-divider for $\mathcal{F}$. It is easy to prove that this vertex must be searched in the maximum rank tree of $\mathcal{F}$, say T_i, and that any t-divider of T_i is also a t-divider for $\mathcal{F}$.

Let us now consider a generic step during the top-down construction of the hierarchy tree HT and let c be the node of HT considered at that step. Let us assume that $S(c)$ is a tree named T. The following algorithm is a straightforward application of the concept of t-divider:

Algorithm `SimpleClustering` (SC): After a t-divider d has been found, the subtrees $T_1 ... T_k$ obtained from T by removing d are generated. We then create k children of node c: these children are the subtrees $T_1 ... T_k$ where we add back vertex d to the one having minimum rank. SC is then recursively applied on all the generated clusters.

The height of the hierarchy tree returned by algorithm `SimpleClustering` is $O(\log n)$ and an upper bound on the degree of its internal nodes is the degree of T itself. Moreover, for any cluster c, the subgraph $S(c)$ induced by the vertices covered by c is connected, i.e., it is a tree. From now on, where there is no ambiguity we refer to this property as *connectivity* of clusters. Though algorithm `SimpleClustering` can be implemented using "off the shelf" data structures and

generates hierarchy trees with small depth, the structure of a hierarchy tree HT grown by SC may be irregular and may depend too much on the input tree: namely, the degree of the internal nodes of HT may be large, according to the degree of the t-divider found by the algorithm at each step (see Figure 3(b)).

In the rest of this section we therefore present two extensions of this algorithm which aim at overcoming these drawbacks. We move from the more realistic hypothesis that the degree of the hierarchy tree should be limited by a constant $g \geq 2$. Under this assumption, we exploit the idea that the regularity of the structure of the hierarchy tree can be best preserved if clusters covering non-connected subgraphs are allowed: in other words, if one is willing to give up the property of connectivity, we expect that more balanced hierarchy trees can be built. On the other side, if clusters are allowed to be non-connected, special attention must be paid to guarantee the validity of the hierarchy tree.

Finding connected clusters. The algorithm that we are going to present produces hierarchy trees with bounded degree $g \geq 2$ and guarantees the connectivity of clusters. Hence, it always returns valid hierarchy trees.

Algorithm `ConnectedClustering` (CC): after finding the t-divider d and generating the subtrees $T_1 ... T_k$, the algorithm checks the value k. If $k \leq g$, it works exactly as algorithm SC. If this is not the case, the subtrees are sorted in non-increasing order according to their ranks: say $T_{i_1} ... T_{i_g} ... T_{i_k}$ the sorted sequence. The children of node c will be $T_{i_1} ... T_{i_{g-1}}, T'$, where $T' = \{d\} \cup \bigcup_{h=g}^{k} T_{i_h}$.

It is easy to see that T' is connected thanks to the presence of the t-divider d and that the degree of HT is at most g (it could be smaller than g due to the case $k \leq g$). Unfortunately, nothing is guaranteed about the rank of T'. Hence, HT may be very unbalanced up to reach linear height (see Figure 3(a)).

Finding balanced clusters. In the following we present an algorithm aimed at producing hierarchy trees with a limited degree $g \geq 2$ and as most balanced as possible. To improve balancing we admit the existence of non-connected clusters. Hence, from now on we assume that our clustering procedure works on a forest $\mathcal{F}$ instead of a tree. We call $F_1 \dots F_h$ the h trees in the forest. We also assume that each tree F_i has a representative vertex r_i which satisfies the following property: any r_i is connected to a vertex $v \notin \mathcal{F}$, v belonging to the original tree to be clustered. Our clustering algorithm always produces clusters which maintain this invariant property: then any hierarchy tree built by this algorithm is valid according to Theorem 1.

Before introducing algorithm `BalancedClustering`, we need to briefly recall a classical partitioning problem concerning the scheduling of a set of jobs, each having its own length, on p identical machines: the goal is to minimize the schedule makespan, i.e., the time necessary to complete the execution of the jobs. This problem has both an on-line and an off-line version and is NP-complete even when $p = 2$ [13]. A simple approximation algorithm for it consists of considering the jobs one by one and of assigning a job to the machine currently having the smallest load [15]. In the off-line case, very good solutions (i.e., an approximation ratio equal to $\frac{4}{3} - \frac{1}{3p}$) can be obtained if the jobs are previously sorted in non-increasing order with respect to their lengths, so as to consider longer jobs first. In the sequel we will refer to this algorithm as `Scheduling`.

We now come back to our original clustering problem. Based on the concept of t-divider of a forest, algorithm `BalancedClustering` works as follows:

Algorithm `BalancedClustering` (BC): it first verifies if $\mathcal{F}$ is already t-divided. If so, it calls procedure `Scheduling`, where $p = g$, the jobs coincide with the trees in the forest, and the length of a job is the rank of the corresponding tree. If $\mathcal{F}$ is not t-divided, after a t-divider for $\mathcal{F}$ has been found in the maximum rank tree F_i, a set of subtrees of F_i, named $T_1 ... T_k$, is generated. Without loss of generality, we assume that T_1 is the unique tree that contains the representative vertex r_i. We call $T_2 \dots T_k$ *lower* trees of $\mathcal{F}$ and any other subtree of the forest, T_1 included, *upper* tree. As to maintain the validity of the hierarchy tree that we are generating, the algorithm has to deal separately with upper and lower trees in order not to mix them in a same cluster. Hence, it builds g_u clusters containing upper trees and $g_l = g - g_u$ clusters with only lower trees. The cluster containing T_1 is the only cluster allowed to contain trees of both types thanks to the fact that the t-divider, which belongs to T_1, connects any lower tree to T_1 itself. The number g_u is chosen proportionally to the total rank of the upper trees. Once g_u and g_l have been chosen, algorithm BC makes two consecutive calls to procedure `Scheduling`: the first call partitions the upper trees into g_u clusters, while the second one partitions the lower trees into g_l clusters. The cluster containing T_1 may be also augmented during this phase, if convenient.

Algorithm `BalancedClustering` is able to effectively balance the cardinalities of clusters, as Figure 3(c) and the experiments presented in Section 6 suggest. This good result is obtained in spite of loosing the property of connectivity. However, we remark that the disconnectivity of clusters remains "weak", because the distance on the original tree between any pair of representative vertices in the same cluster is always equal to 2 (cfr. Theorem 1).

4 Two Well-Known Clustering Algorithms

In this section we briefly recall two well-known clustering algorithms that we implemented in our package after adapting them to work on general trees.

The first algorithm, due to Gonzalez [14], is designed for clustering a set of points into a fixed number k of clusters and has been in wide use during the last twenty years. In the first step Gonzalez's algorithm picks an arbitrary point to be the representative of the first cluster. In each of the next $k - 1$ steps a new representative point is chosen through a max-min selection: the i-th representative is a point which maximizes the minimum distance from the first $i - 1$ representatives. Cluster S_i consists then of all points closer to the i-th representative than to any other one. When specialized to trees, the farthest-point algorithm (FP) considers as points the vertices of the tree and replaces the Euclidean distance between two points with the length of the path between the corresponding vertices on the tree. In our implementation the choice of the first representative vertex is not random: indeed, a vertex maximizing the average distance from all the other vertices is chosen. It is easy to see that this vertex is always a leaf. We observed in the experimentation that this choice greatly improves the behavior of the algorithm.

The second algorithm (FR) is due to Frederickson [12] and generates connected clusters with limited cardinalities, thus producing quite balanced hierarchy trees. Even if it was designed for clustering cubic trees, it can be easily generalized to trees with bounded degree. Vertices are partitioned on the basis of the topology of the tree with a bottom-up strategy that grows up the clusters in a recursive way starting from the leaves. A cluster is built when its current cardinality falls into a prefixed range $[z, D(z-1)+1]$, where D is the tree degree and z is a value to be suitably chosen. In our implementation z is selected in $\{1..n\}$ in order to limit the degree g of HT as much as possible: if all the values for z produce more than g clusters, we choose the value that minimizes the number of created clusters. Since algorithm FR is not always able to satisfy the requirements on g, the degree of the hierarchy trees it returns is not bounded.

5 Experimental Setup

We implemented the tree clustering algorithms described in Section 3 and in Section 4 in ANSI C and we debugged and visualized them in the algorithm animation system Leonardo [5]. To prove the feasibility of the clustering approach for the visualization of large trees, we also realized a prototype of a visualization facility which hinges upon this approach, letting the user navigate through the hierarchy tree by expanding and shrinking clusters. The prototype, comprehensive of algorithm implementations, visualization code, and tree generator, is available at the URL `http://www.dsi.uniroma1.it/~finocchi/experim/treeclust-2.0/`. In the rest of this section we review some interesting aspects of our experimental framework related to instance generation and performance indicators.

The random tree generator. The problem instances that we used in the tests are synthesized and have been randomly generated. Since both the degree and the balancing of the input tree appeared to us to be crucial factors for the performances of the t-divider based algorithms, we designed a tree generator which produces structured instances taking into account these parameters. The generator works recursively and takes as input four arguments, named n, d, D, and β. n is the number of vertices of the tree T to be built; d and D are a lower and an upper bound for its degree, respectively; β is the unbalancing factor of T, i.e., a real number in $[0, 1]$ which indicates how much T must be unbalanced: the bigger is β, the more unbalanced will be T. Let us suppose that during a generic step the generator has to build a tree T, rooted at a vertex v, with parameters n, d, D, and β. If $n-1 < d$, then all the $n-1$ vertices are considered as children of v: hence, T can have vertices with degree smaller than d, though all the children of such vertices are always leaves. If $n-1 \geq d$, the number r of children of v is randomly chosen by picking up a value in $[d, min\{n-1, D\}]$ and each of the $n-1$ vertices of T (except for v) is randomly assigned to any of the r subtrees. We call n_i the number of vertices assigned to the subtree T_i rooted at the i-th child. If no n_i is $\geq \lfloor \beta \cdot n \rfloor$, a further step to make T more unbalanced is performed. If w.l.o.g. $n_1 = max_{1 \leq i \leq r} n_i$, exactly $\lfloor \beta \cdot n \rfloor - n_1$ vertices are randomly removed from the subtrees which they belong to and are added to T_1. This construction satisfies the unbalancing factor β required for vertex v.

Performance indicators. We considered three kinds of quality measures, aimed at highlighting different aspects of the structure of the hierarchy trees grown up

by the algorithms. In particular, based on the considerations in Section 2, we studied *balancing*, *depth*, and *degree* properties.

Measure `Unbalancing` estimates how much a hierarchy tree is balanced and is computed according to the following idea: the children of a cluster c in HT should have "not too different" cardinalities. Let n be the cardinality of c, let $d(c)$ be its degree, and let n_i be the cardinality of the i-th child of c. In the most favourable scenario, each child of c covers $\frac{n}{d(c)}$ vertices. A good estimate of the unbalancing of cluster c is obtained by counting how much the real cardinalities of its children are different from the optimal child cardinality $\frac{n}{d(c)}$. The quantity $\sum_{i=1}^{d(c)} |n_i - \frac{n}{d(c)}|$ is then normalized with respect to n, so as not to count bigger contributions for clusters covering more vertices. Measure `Unbalancing` is the sum over all the clusters in the hierarchy tree of the values defined above and is averaged on the total number of clusters of HT which are not leaves. The lower is the value of this measure, the more balanced is the hierarchy tree. We remark that we suitably designed measure `Unbalancing` to be able to correctly compare even hierarchy trees having internal nodes with different degrees.

A second performance indicator relates to the depth of the hierarchy tree and is referred to as `External Path Length`. It represents the sum of the depths of the leaves of the hierarchy tree. In this case it is not necessary to average this measure on the number of leaves, which is always the same for all the hierarchy trees associated to the same tree instance. Finally, measures `Average Degree` and `Maximum Degree` refer to the average and to the maximum degree of the hierarchy tree, respectively.

6 Experimental Results

In this section we empirically study the behavior of the previously described tree clustering algorithms, using different values of t in the analysis of the t-divider based ones. The main goals of our experiments are the following: (a) comparing the algorithms, studying if/how/which algorithms are affected by the degree and balancing of the input tree and giving insight on the choice of a "good" algorithm for a given value of g and a given problem instance; (b) validate/disprove the practical utility of the theoretical concept of t-divider; (c) checking if the idea of relaxing the connectivity of clusters and the use of Graham's heuristic effectively help to improve balancing of the hierarchy tree.

The experimentation contemplates three kinds of tests, where either the number n of vertices, or the unbalancing factor β, or the maximum degree D of the input tree vary. All the experiments used from 50 to 100 trials per data point and the sequence of seeds used in each test was randomly generated starting from a base seed. The algorithms based on t-dividers are referred to in the charts by means of their name (SC, CC, or BC) followed by the value t that they employ. We sometimes report the results of the tests for different values of g: though the general trend of the curves is maintained, the relative performances of the algorithms often vary. We preliminarly remark that the values of the metrics are usually smaller for bigger values of g: roughly speaking, we can explain this by observing that, when g is bigger, the algorithms have usually more possibilities for decomposing the original tree and so can do it better. We now give more insight on the tests we performed.

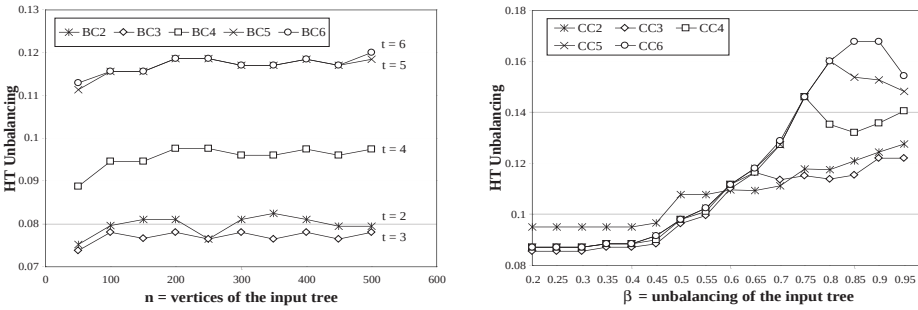

Fig. 4. Utility of t-dividers.

Experiments on t-dividers. Figure 4 presents the results of the comparison of algorithms BC and CC for different values of t, i.e., $2 \leq t \leq 6$. The graph on the left part of Figure 4 shows that algorithm BC has an almost constant behavior as the instance size increases and that the choice of t greatly affects its performances. In particular, the best results are obtained for $t = 3$, thus proving that the use of centroids ($t = 2$), which is frequent in literature, does not necessarily yield the best solutions. This is even more evident in the graph on the right part of Figure 4, where the interrelation between the balancing of the input tree and the best choice for t is analyzed w.r.t. algorithm CC: $t = 2$ represents the worst choice for quite balanced trees, i.e., for $\beta < 0.6$. It is worth to point out that all the curves have a local maximum in $\beta = \frac{t-1}{t}$. This trend depends on the choice of the t-divider vertex: for a fixed t, as long as $\beta \leq \frac{t-1}{t}$, the same vertex continues to be chosen as t-divider, while the tree becomes more and more unbalanced. As soon as β gets bigger than $\frac{t-1}{t}$, the vertex chosen as t-divider changes, implying a more balanced partition of the vertices. Similar results have been obtained for different parameters of the experiments and for algorithm SC, as well. Hence, in the tests presented later on, for each algorithm we will limit to report the chart corresponding to the best choice of t.

Algorithms' comparison. The first test that we discuss analyzes the behavior of the algorithms when the maximum degree of the input tree increases. The charts in Figure 5 have been obtained by running the algorithms on 300-vertex trees with unbalancing factor 0.5, minimum degree 2, and maximum degree ranging from 3 to $30 = \frac{n}{10}$ (this implies that the average degree increases, too). Each row reports the graphs obtained by running the algorithms with a different value g of the maximum degree required for HT, $2 \leq g \leq 4$. A first reading of the charts is sufficient to grasp the trend of curves and to separate the algorithms into two well distinct groups: algorithm SC and algorithm BC are able to benefit of the increase of the degree, while algorithms CC, FP, and FR get worse as D increases. This behavior, that holds w.r.t. both `Unbalancing` and `External Path Length`, strenghtens the following intuition: if D is large, the t-dividers found by the algorithms at each step will probably have big degrees and so will disconnect the tree into numerous clusters of small cardinalities. This immediately clarifies why SC gets better. For what concerns the balanced approach, it is then easy to recombine numerous small clusters to form exactly g clusters, while the connected strategy is not well suited for this: due to the necessity of maintaining the

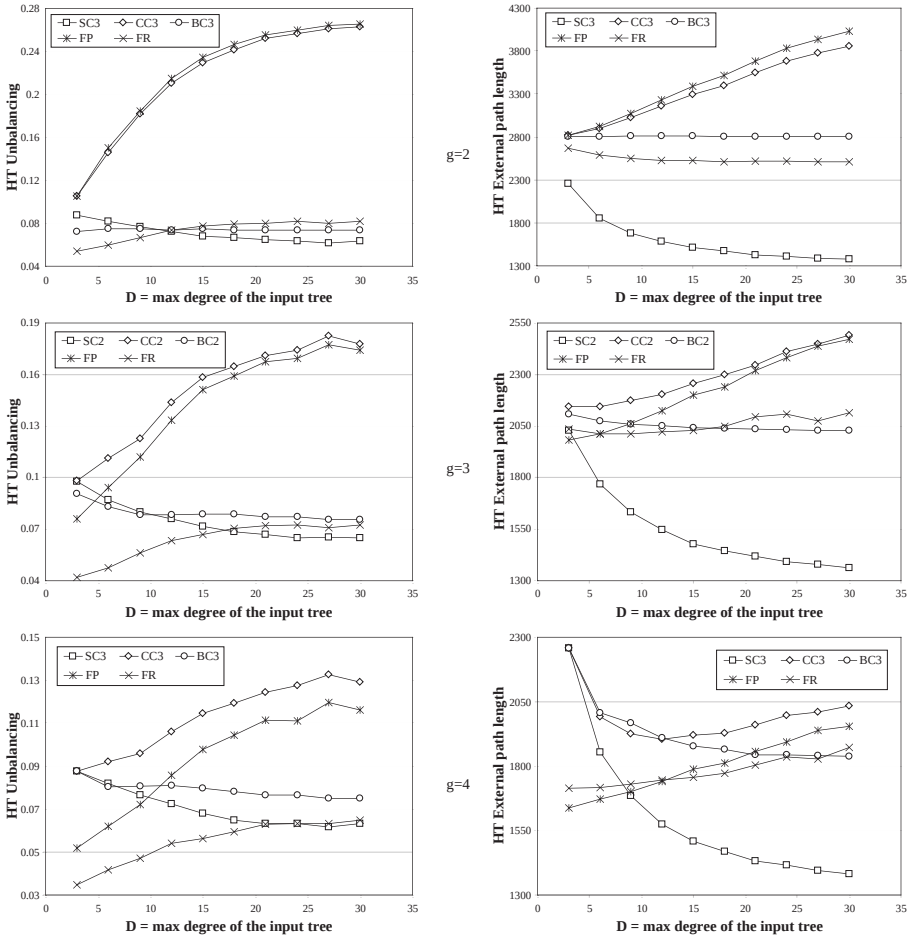

Fig. 5. Experiments for increasing maximum degree of the input tree: charts on each row are for $g = 2, 3, 4$, respectively.

connectivity of clusters, it will usually obtain an unbalanced partition consisting of $g - 1$ small clusters and a single big cluster. The worsening behaviour of FP and FR can be explained similarly to that of algorithm CC. However, we remark that algorithm FR obtains very good results w.r.t. measure `Unbalancing`, especially for larger value of g, despite the increasing trend of its curve.

It is also worth observing that, as we increase the value of g keeping the other parameters fixed, the curves get closer and their relative position changes. Let us focus, for instance, on the unbalancing measure concerning algorithms BC and FP: the related curves do not cross for $g = 2$, a crossing is introduced in $D = 5$ when $g = 3$, and such a crossing point moves forward as g gets bigger ($D = 12$ for $g = 4$). We underline that before the crossing point algorithm FP exhibits a better behavior than algorithm BC. This suggests that the choice of the best algorithm to use on a given instance is much influenced by a subtle

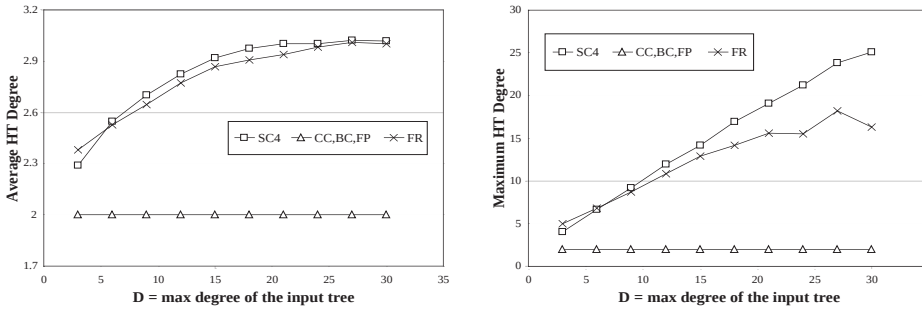

Fig. 6. Algorithms' comparison w.r.t. average and maximum degree of the hierarchy tree. In both the experiments $g = 2$.

interplay between the maximum degree g allowed for HT and the average degree of the input tree. Similar experiments should be reported for different choices of parameter β in order to be able to recognize, given a problem instance and a value for g, whether FP is preferable to BC. Due to the lack of space we do not report such graphs in this paper.

Even if algorithms FR and SC behave very well w.r.t. measures `Unbalancing` and `External Path Length`, as Figure 5 shows, we have to notice that both of them do not limit the degree of the hierarchy tree, being often impractical in graph drawing applications. Figure 6 highlights that both the average and the maximum degree of the hierarchy trees returned by these algorithms are much bigger that the value required for g.

We also studied how the relative performances of the algorithms are affected by the instance size and by the balancing of the input tree (see Figure 7). In the left chart we considered 300-vertex trees with degree in $[2, 3]$ and we increased the value of the unbalancing factor β from 0.2 to 0.95, with step 0.05. We found that algorithm SC is much influenced by the growth of β, suggesting that unbalanced trees are more difficult to partition than balanced ones. In the right chart we therefore considered the same range for the degree and we fixed β to 0.8; we varied the vertices in the range 50 to 500. With respect to measure `Unbalancing` no algorithm is affected by the increase of n, i.e., the measure is almost constant. This depends on the fact that, for a fixed β, smaller trees returned by our generator locally reflect the global structure of larger ones, yielding the algorithms to a similar behavior. The relative positions of the unbalancing curves are not very surprising. As the algorithms are concerned, it is easy to see that both algorithm CC and algorithm FP do not take into account balancing problems at all, possibly obtaining very unbalanced trees. Also the simple clustering strategy is affected by the balancing of the input instances and this is conveyed on the hierarchy trees. As expected from theoretical considerations, algorithms BC and FR come out to be the best ones w.r.t. the choice of parameters in this test.

7 Conclusions

In this paper we have focused on the problem of experimentally evaluating with respect to criteria relevant in graph drawing applications the quality of hierarchi-

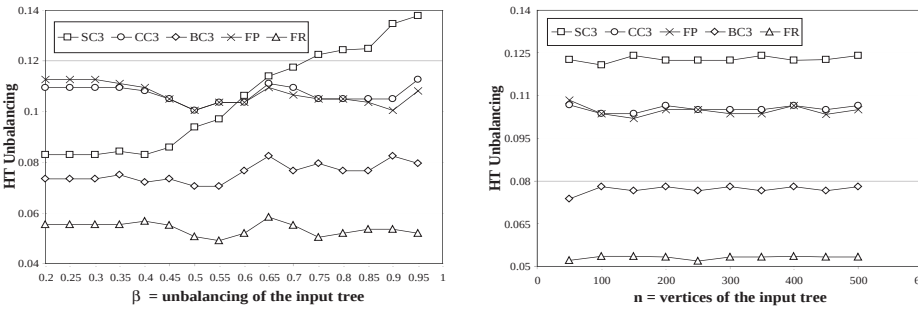

Fig. 7. Algorithms' comparison for increasing unbalancing and increasing number of vertices of the input tree.

cal decompositions of trees built using different clustering subroutines. We have designed new tree clustering algorithms hinging upon the notion of t-divider and we have empirically shown the relevance of this concept as a generalization of the ideas of centroid and separator. In particular, our experiments prove that the use of centroids, that is very frequent in literature, does not usually yield the best solution in practice. From the comparison of the algorithms' behavior, it comes out that the choice of the best algorithm to use on a given tree instance depends on structural properties of the instance (e.g., balancing and degree) and on the quality measure to be optimized on the hierarchy tree, either balancing or depth or degree. Algorithms FR and SC turn out to behave very well w.r.t. balancing and depth, respectively, but they do not limit the degree of the hierarchy tree, being often impractical for graph drawing purposes. Algorithm BC obtains fairly good results w.r.t. all the performance indicators we considered.

We are currently working on extending the concepts and ideas presented throughout the paper to edge-weighted trees; this is a better model of real situations, since the edge weights represent the level of correlation between vertices. From an experimental point of view, we are also studying the quality of hierarchical decompositions associated to trees obtained from real-life applications. The interested reader can download our visualization prototype at the URL `http://www.dsi.uniroma1.it/~finocchi/experim/treeclust-2.0/`.

References

1. Abello, J., and Krishnan, S.: *Navigating Graph Surfaces*, in Approximation and Complexity in Numerical Optimization, Continous and Discrete Problems, Kluwer Academic Publishers, 1–16, 1999.
2. Batagelj, V., Mrvar, A. and Zaversnik, M.: *Partitioning approach to visualization of large graphs*, Proc. 7th Symposium on Graph Drawing (GD'99), LNCS 1731, 90-97, 1999.
3. Buchsbaum, A.L., and Westbrook, J.R.: *Maintaining hierarchical graph views*, Proc. 11th ACM-SIAM Symposium on Discrete Algorithms (SODA'00), 566-575, 2000.
4. Buckley, F., and Harary, F.: *Distance in Graphs*, Addison Wesley, MA, 1990.

5. Crescenzi, P., Demetrescu, C., Finocchi, R. and Petreschi, R.: *Reversible execution and visualization of programs with LEONARDO*, Journal of Visual Languages and Computing, April 2000.
6. Demetrescu, C., Di Battista, G., Finocchi, I., Liotta, G., Patrignani, M., and Pizzonia, M.: *Infinite trees and the future*, Proc. 7th Symposium on Graph Drawing (GD'99), LNCS 1731, 379–391, 1999.
7. Di Battista, G., Eades, P., Tamassia, R. and Tollis, I.: *Graph Drawing: Algorithms for the visualization of graphs*, Prentice Hall, Upper Saddle River, NJ, 1999.
8. Duncan, C.A., Goodrich, M.T. and Kobourov, S.G.: *Balanced aspect ratio trees and their use for drawing very large graphs*, Proc. 6th Symposium on Graph Drawing (GD'98), LNCS 1547, 111-124, 1998.
9. Duncan, C.A., Goodrich, M.T. and Kobourov, S.G.: *Planarity-preserving clustering and embedding for large planar graphs*, Proc. 7th Symposium on Graph Drawing (GD'99), LNCS 1731, 186-196, 1999.
10. Eades, P. and Feng, Q.: *Multilevel visualization of clustered graphs*, Proc. 4th Symposium on Graph Drawing (GD'96), LNCS 1190, 101-112, 1997.
11. Finocchi, I., and Petreschi, R.: *On the validity of hierarchical decompositions of trees*, manuscript, June 2000.
12. Frederickson, G.N.: *Data structures for on-line updating of minimum spanning trees, with applications*, SIAM Journal on Computing, 14:4, 781-798, 1985.
13. Garey, M.R., and Johnson, D.S.: *Computer and intractability: a guide to the theory of NP-completeness*, W.H.Freeman, 1979.
14. Gonzalez, T.: *Clustering to minimize the maximum intercluster distance*, Theoretical Computer Science, 38, 293-306, 1985.
15. Graham, R.L.: *Bounds for multiprocessing timing anomalies*, SIAM Journal on Applied Mathematics, 17, 263-269, 1969.
16. Harel, D. and Koren, Y.: *A fast multi-scale method for drawing large graphs*, Proc. 5th Working Conference on Advanced Visual Interfaces (AVI'00), ACM Press, 282-285, 2000.
17. Huang, M.L. and Eades, P.: *A fully animated interactive system for clustering and navigating huge graphs*, Proc. 6th Symposium on Graph Drawing (GD'98), LNCS 1547, 374-383, 1998.
18. Lipton, R.J. and Tarjan, R.E.: *A separator theorem for planar graphs*, SIAM Journal on Applied Mathematics, 36(2), 177–189, 1979.
19. Newbery, F.J. : *Edge concentration: a method for clustering directed graphs*, Proc. 2nd Workshop on Software Configuration Management, 76-85, Princeton, New Jersey, 1989.
20. Nishizeki, T., and Chiba, N.: *Planar Graphs: Theory and Algorithms*, North Holland, 1988.
21. Stein, B. and Niggeman, O.: *On the nature of structure and its identification*, Proc. 25th Workshop on Graph-Theoretic Concepts in Computer Science (WG'99), LNCS, 1999.
22. Wei, Y.C., and Cheng, C.K.: *Ratio cut partitioning for hierarchical designs*, IEEE Transactions on Computer Aided Design, 10(7), 911-921, 1991.