Divider-based algorithms for hierarchical tree partitioning

Irene Finocchia\textsuperscript{a,1}, Rossella Petreschi\textsuperscript{b,2}

\textsuperscript{a}Dipartimento di Informatica, Sistemi e Produzione, Università degli Studi di Roma “Tor Vergata”, Via di Tor Vergata 110, 00133 Roma, Italy

\textsuperscript{b}Dipartimento di Informatica, Università degli Studi di Roma “La Sapienza”, Via Salaria 113, 00198 Roma, Italy

Received 30 September 2001; received in revised form 30 October 2002; accepted 19 February 2003

Abstract

We present algorithms for computing hierarchical decompositions of trees satisfying different optimization criteria, including balanced cluster size, bounded number of clusters, and logarithmic depth of the decomposition. Furthermore, every high-level representation of the tree obtained from such decompositions is guaranteed to be a tree. These criteria are relevant in many application settings, but appear to be difficult to achieve simultaneously. Our algorithms work by vertex deletion and hinge upon the new concept of \( t \)-divider, that generalizes the well-known concepts of centroid and separator. The use of \( t \)-dividers, combined with a reduction to a classical scheduling problem, yields an algorithm that, given a \( n \)-vertex tree \( T \), builds in \( O(n \log n) \) worst-case time a hierarchical decomposition of \( T \) satisfying all the aforementioned requirements.

\( \)\textsuperscript{2003 Published by Elsevier B.V.}

Keywords: Graph partitioning problems; Hierarchical decompositions; Trees; Centroids

\textsuperscript{1} Author partially supported by the EU under contract number IST-2001-33555 (COSIN), by the Italian Ministry of University and Scientific Research (project “ALINWEB: Algorithmics for Internet and the Web”), and by the University of Rome “La Sapienza” (project for Young Researchers “Algorithms and Techniques for the Visualization of Large Graphs”, 2001).

\textsuperscript{2} Author partially supported by the Italian Ministry of University and Scientific Research (Project “ALINWEB: Algorithmics for Internet and the Web”) and by the University of Rome “La Sapienza” (project “Parallel and Distributed Codes”, 2002).

E-mail addresses: finocchi@disp.uniroma2.it (I. Finocchi), petreschi@dsi.uniromal.it (R. Petreschi).
1. Introduction

Graphs arising in real applications are becoming increasingly large. Designing efficient data structures to manage them is thus an important task. A common approach to speed up the processing of a large graph \( G \) consists of using decomposition techniques to build a clustered graph from \( G \): informally speaking, such a graph represents a summary of \( G \) and is obtained by grouping together suitably chosen disjoint sets of vertices, called clusters, and by computing induced edges between them. Graph decomposition techniques are often used recursively, so as to build a sequence of meta-graphs from \( G \): each meta-graph in the sequence is a summary of the previous one, the first and more detailed meta-graph coinciding with \( G \) itself. Such a sequence defines a hierarchical decomposition of \( G \) represented by a rooted tree, known in literature as hierarchy tree, whose leaves are vertices of \( G \) and whose internal nodes are clusters. Different selections of nodes of the hierarchy tree lead to different high-level representations of \( G \) that can be traversed by performing shrinks and expansions of clusters.

This paper is concerned with hierarchical decompositions of trees. Tree clustering procedures are an important subroutine for partitioning generic graphs: they can be applied, for instance, to the block-cut-vertex tree of a graph in order to obtain a first rough partition of its vertices. In addition, tree-like structures frequently arise in many practical problems (e.g., evolutionary and parse trees). A few examples of application settings where recursive tree decompositions have proven to be effective are listed in the following. In the field of dynamic graph algorithms, Frederickson’s technique for maintaining a minimum spanning tree of a graph under updates of the costs of its edges hinges upon a multi-level topological partition of the vertices of the spanning tree useful for reducing the update time [11]. Tree layout algorithms also benefit from decompositions: clusters can be visualized as single vertices or filled-in regions, making it possible to display effectively the global structure of large trees in a limited area [8]. The profile minimization problem, which is NP-complete on general graphs, has been polynomially solved on trees thanks to a recursive partition of the vertices of the tree by means of centroids [15]. Other applications include parallel and distributed computations, operating systems, external searching, allocation of service centers; we refer the interested reader to [2,3,7,14] for more details on these topics.

According to the application at hand, different optimization criteria can be considered when building clusters. It is in general well accepted that good decompositions should exhibit a strong relationship between vertices in the same cluster and a low coupling between clusters. Additional objective functions should be optimized when dealing with recursive partitions. Among them, structural properties of the hierarchy tree such as limited degree, small depth, and balancing deserve special attention. For instance, in a distributed setting, where clusters correspond to processors and vertices to tasks to be performed, having a bounded number of clusters of almost equal size enhances locality, decreases communication, and guarantees better load balancing. Finally, it is quite natural to require any high-level representation of a graph obtained from its hierarchy tree to reflect the topology and the properties of the graph itself. An immediate motivation for this comes from graph drawing applications: in order not to mislead the viewer, it is desirable, e.g., for any representation of a planar graph to
be planar or for any representation of a tree to be connected and acyclic. A hierarchy
tree that satisfies this property is said to be valid [9]. Building valid hierarchy trees
or checking the validity of a given hierarchy tree in polynomial time may be quite
difficult.

1.1. Related work

Due to the special structure of trees, most graph partitioning algorithms either are not
efficient or fail to find appropriate tree decompositions. For this reason a lot of research
has been devoted since the 1980s to designing specific tree partitioning algorithms
tailored to a variety of applications (see, e.g., [2,4,7,14,17,19,21] and the references
therein). Independently of the optimized objective function, we can roughly distinguish
two main approaches to tree partitioning, according to the fact that clusters are obtained
by deleting vertices or edges.

A well-known technique based on edge deletion, the shifting algorithm technique,
has been presented in [19] and applied to many optimization problems on trees in
several subsequent papers [1–3,20]. A partition of a tree is identified by associating
cuts to its edges. Cuts are assigned via a sequence of shifts, i.e., basic operations that
move a cut from an edge to an adjacent one; different shifting rules allow it to optimize
different functions. Other edge deletion algorithms for tree partitioning are described
in [4,14]. In particular, [14] suggests algorithms for partitioning a \( n \)-vertex tree into \( g \)
balanced clusters: the size of each cluster is in the range \([ (1 - \varepsilon/2) n/g, (1 + \varepsilon) n/g ]\),
where parameter \( \varepsilon \in [0,1] \) can be given as input.

At first sight, removing vertices may appear to be less flexible than removing edges:
based on the degree of the deleted vertex, the tree may be disconnected into several
subtrees of very different size, and optimizing both cluster size and number of subtrees
may be more difficult. However, an accurate choice of the vertex to be removed (e.g.,
choosing a centroid or a center of the tree) allows it to guarantee upper/lower bounds
on the size or on the diameter of each cluster. An example can be found in [15].

To conclude, we remark that most of the aforementioned algorithms may not be good
at optimizing simultaneously properties of the hierarchy tree such as balancing, depth,
and degree. Moreover, since typically they are not applied recursively, they do not
address at all the problem of building valid hierarchy trees: actually, it is not difficult
to see that many of the partitioning algorithms that find disconnected clusters (such as
the algorithms in [14]) may produce non-valid decompositions.

1.2. Results and techniques

In this paper we consider the problem of computing hierarchical decompositions
of trees that: (1) are valid; (2) have logarithmic depth; (3) exhibit balanced cluster
size; and (4) have bounded degree. We present efficient algorithms based on vertex
deletion for computing such decompositions. We first show that it is easy to guarantee
either logarithmic depth or bounded degree for the hierarchy tree, but not both, except
for special classes of trees. We therefore present an algorithm that overcomes this
drawback: if \( n \) is the number of vertices of the original tree, it builds in \( O(n \log n) \)
worst-case running time hierarchy trees that exhibit limited degree, balanced cluster size, logarithmic depth, and are valid. We remark that $\Omega(n)$ is a trivial lower bound on the construction of any hierarchy tree.

The backbone of our algorithms is the new concept of $t$-divider, that generalizes concepts well known in literature, such as centroids and separators [6,16,18]. The performances of the almost-optimal partitioning algorithm are achieved by exploiting the use of $t$-dividers, a reduction to a classical scheduling problem, and the following simple idea: when the degree of the hierarchy tree must be limited by a constant, small depth and balanced cluster size can be more easily guaranteed if clusters covering non-connected subgraphs are allowed. In order to preserve the structure of the original tree $T$ in the decomposition, we consider a “weak” form of connectivity relaxation, forcing clusters to satisfy a short distance property: for each pair of disconnected components in the same cluster there exist two representative vertices whose distance in $T$ is 2. Roughly speaking, the short distance property makes “unnatural” partitions of the tree not possible and allows us to prove that the algorithm builds valid hierarchy trees.

1.3. Structure of the paper

Section 2 recalls terminology and preliminary definitions related to hierarchy trees. Section 3 introduces the notion of $t$-divider of a tree, providing a structural characterization of the set of $t$-dividers. Section 4 presents two naïve partitioning strategies and discusses their drawbacks. Section 5 describes the improved scheduling-based algorithm, analyzing its performances. Section 6 sums up and addresses directions for further research.

2. Background on hierarchy trees

In this section we give preliminary definitions and notation used throughout the paper. In particular, we recall the definition of hierarchy tree and we discuss the concepts of covering and of contraction of a graph on a hierarchy tree associated with it [5,9].

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

According to standard terminology, we call depth of $HT$ the maximum distance from a leaf to the root and degree of a node of $HT$ the number of its children. W.l.o.g. we consider hierarchy trees whose internal nodes have degree $\geq 1$.

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 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 $c$. We say
that $c$ is a singleton if its cardinality is equal to 1. 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 neither coincident nor 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 it follows that $G'$ contains neither self-loops nor multiple edges.

**Definition 2.** Let $HT(N,A)$ be a hierarchy tree associated with a graph $G(V,E)$. A set $C \subseteq N$ is a covering of $G$ on $HT$ if and only if $\forall v \in V$ there exists unique $c \in C$ such that $v \prec c$. A contraction 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. Fig. 1b shows a possible hierarchy tree of the 12-vertex graph given in Fig. 1a. (Throughout the paper we use letters and integer numbers to refer to internal nodes and leaves of the hierarchy tree, respectively.) The internal nodes of the hierarchy tree in Fig. 1b are squared and, for clarity, no link is shown. A covering consisting of clusters $\{3, d, b, g, 12, f\}$ is highlighted on the hierarchy tree and the corresponding contraction of the graph is depicted in Fig. 1c.

Since we are concerned with hierarchical decompositions of trees and forests, in the rest of this paper we assume that the graph to be clustered is a free tree $T(V,E)$. Unless otherwise stated, we denote with $n$ the number of its vertices, also called size of tree $T$. As observed in Section 1, under this hypothesis a natural requirement on the hierarchy tree is that any contraction of $T$ obtained from it is a tree: if this holds, $HT$ is said to be valid [9]. More formally:

**Definition 3** (Finocchi and Petreschi [9]). Let $HT(N,A)$ be a hierarchy tree associated with a tree $T(V,E)$. A contraction of $T$ on $HT$ is valid if and only if it is a tree. $HT$ is valid if and only if all the contractions of $T$ obtained from $HT$ are valid.
It is worth observing that not any hierarchy tree associated with a tree is valid. For instance, Figs. 2b and 2c show a valid hierarchy tree associated with a chain of four vertices and a contraction of the chain on it, while Fig. 2d depicts a non-valid hierarchy tree associated with the same chain: the contraction associated with covering \{a,2,3\} contains a cycle, as shown in Fig. 2e. All the existing links are reported on the hierarchy trees as dashed lines.

In the rest of this paper we restrict our attention on algorithms that generate valid hierarchy trees. A structural characterization of valid hierarchy trees is proved in [9]. Before reporting it, we need some additional terminology.

Definition 4. Let c be a node of a hierarchy tree HT associated with a free tree T. Let u and v be two vertices of T covered by c.

- u,v are a broken pair of cluster c if and only if they are neither coincident nor connected in the subgraph S(c) induced by the vertices covered by c;
- a broken pair u,v is a minimum-distance broken pair of c if and only if, for each w \neq u,v in the unique path of T between u and v, w \not< c.

Theorem 1 (Finocchi and Petreschi [9]). Let HT(N,A) be a hierarchy tree associated with a free tree T(V,E). HT is valid if and only if for each minimum-distance broken pair u,v of HT the distance between u and v in T is 2.

Note that the validity of HT is trivially guaranteed if all its clusters cover connected subgraphs, since no broken pair exists. For the purposes of this paper, we state a weaker sufficient condition that will be useful thereafter to prove the validity of the hierarchy trees grown by the algorithm described in Section 5.

Corollary 1. Let HT(N,A) be a hierarchy tree associated with a free tree T(V,E). HT is valid if, for each c \in N such that S(c) is disconnected, it holds: there exists a vertex v \in V, v \not< c, such that each connected component of S(c) contains a neighbor of v in T.

Proof. The condition in the statement of this theorem implies that, for each pair of connected components of S(c), there exist two vertices u and w, disconnected in S(c),
whose distance in $T$ is 2: actually, $u$ and $w$ are neighbors of vertex $v$ in $T$. Hence, the condition in the statement of Theorem 1 holds and the hierarchy tree is valid.

3. Properties of $t$-dividers

In this section we introduce the concept of $t$-divider of trees and forests: $t$-dividers generalize the well-known concepts of centroid and separator and are the backbone of the tree decomposition algorithms presented in Sections 4 and 5.

**Definition 5.** Given a $n$-vertex free tree $T(V, E)$ and a constant $t \geq 2$, a vertex $v \in V$ is a $t$-divider of $T$ if and only if its removal disconnects $T$ into trees of size $\leq \lceil (t - 1)/t \rceil n$.

The notion of $t$-divider is a natural generalization of the concept of centroid, well known from graph theory literature [6]: a centroid of a $n$-vertex tree is a vertex whose deletion results in a forest with trees of size $\leq \lceil n/2 \rceil$ and is obviously a 2-divider. Furthermore, a 1-separator of a tree is a vertex whose removal partitions the vertex set of the tree into two disjoint sets $A$ and $B$ such that neither $A$ nor $B$ contains more than $a \cdot n$ vertices, for $a < 1$ [18]. It can be easily proved that 1-separators are $t$-dividers for $t = 3$ (this implies $a = \frac{2}{3}$): for details we refer to the proof of Theorem 9.1 in [18]. Fig. 3 shows the $t$-dividers of a tree for different values of $t$ and highlights that $t$-dividers are not necessarily unique. Note also that in this example the set of $t$-dividers is the same for each $t$ such that $4 \leq t \leq 10$.

In general, if $t \geq t' \geq 2$, any $t'$-divider of $T$ is also a $t$-divider, because $(t' - 1)/t' \leq (t - 1)/t$. This consideration, together with the fact that each tree has at least a centroid [6], implies that there exists at least a $t$-divider for any constant $t \geq 2$.

In the following we first state basic properties of $t$-dividers and of the tree decompositions obtained from their removal, and then we present a characterization of the set of $t$-dividers.

**Lemma 1.** Let $T(V, E)$ be a free tree and let $v$ be a vertex of $T$. For any constant $t \geq 2$, if $v$ is not a $t$-divider, the removal of $v$ disconnects $T$ into $k$ subtrees $T_1, \cdots, T_k$.

---

Fig. 3. $t$-dividers of an 11-vertex tree for different values of $t$. 

---
such that, $\forall j \in [1, k]$, size$(T_j) \neq \lfloor(t - 1)/t\rfloor n$ and there exists a unique $T_i$ with more than $\lfloor((t - 1)/t)n\rfloor$ vertices.

**Proof.** As $v$ is not a $t$-divider, at least a tree among $T_1 \cdots T_k$, say $T_i$, must have size $> \lfloor((t - 1)/t)n\rfloor$. The number of all the vertices of $T$ different from $v$ and $\not\in T_i$ is equal to $n - \text{size}(T_i) - 1 < n - \lfloor((t - 1)/t)n\rfloor - 1 \leq [n/t] - 1 \leq [n/t] \leq \lfloor((t - 1)/t)n\rfloor$, since $t \geq 2$. Hence, any subtree other than $T_i$ has size $< \lfloor((t - 1)/t)n\rfloor$. This proves that $T_i$ is unique. \[\square\]

**Lemma 2.** Let $T(V,E)$ be a free tree and let $v$ be a vertex of $V$ whose removal disconnects $T$ into $k$ subtrees $T_1 \cdots T_k$. Let $t \geq 2$ be a constant and let $h$ be the number of subtrees among $T_1 \cdots T_k$ with size equal to $\lfloor((t - 1)/t)n\rfloor$. Then $0 \leq h \leq 2$ and $t > 2 \Rightarrow h \leq 1$.

**Proof.** The following inequality must hold: $h \lfloor((t - 1)/t)n\rfloor + 1 \leq n$. The most favorable scenario is when $\lfloor((t - 1)/t)n\rfloor$ is minimum, i.e., for $t = 2$. In this case the inequality yields $\lfloor n/2\rfloor \leq (n - 1)/h$. Assuming $h > 2$ implies $\lfloor n/2\rfloor < (n - 1)/2$, that is impossible. Hence, it must be $h \leq 2$.

Let us now assume $h = 2$. Since $\lfloor((t - 1)/t)n\rfloor = \lfloor((t - 1)/t)n\rfloor - \alpha$, $0 \leq \alpha < 1$, we must have $\lfloor((t - 1)/t)n\rfloor \leq (n - 1)/2 + \alpha < (n + 1)/2$. By means of simple manipulations it is easy to see that this is equivalent to require $t < 2n/(n - 1)$. The case $n = 2$ is impossible, because we have a vertex and at least two non empty trees, and for $n > 2$ it holds $2n/(n - 1) \leq 3$. Thus, $h = 2$ is possible only if $t < 3$, proving that $t > 2 \Rightarrow h \leq 1$. \[\square\]

**Theorem 2.** Let $T(V,E)$ be a free tree and let $v$ be a vertex of $V$ whose removal disconnects $T$ into $k$ subtrees $T_1 \cdots T_k$. Let $t \geq 2$ be a constant. Then:

1. if $v$ is not a $t$-divider, all the $t$-dividers of $T$ are in the maximum size subtree among $T_1 \cdots T_k$;
2. if $v$ is a $t$-divider, let $h$ be the number of subtrees among $T_1 \cdots T_k$ of size equal to $\lfloor((t - 1)/t)n\rfloor$. Then:
   (a) $h = 2 \Rightarrow T$ contains no other $t$-divider;
   (b) $h = 1 \Rightarrow$ all the other $t$-dividers of $T$, if any, are in the unique subtree of size $\lfloor((t - 1)/t)n\rfloor$;
   (c) $h = 0 \Rightarrow$ all the other $t$-dividers of $T$, if any, are in subtrees of size $\geq \lfloor n/t \rfloor$.

**Proof.** Let us first consider the case where $v$ is not a $t$-divider. In view of Lemma 1 there exists a unique $T_i$ among $T_1 \cdots T_k$ with more than $\lfloor((t - 1)/t)n\rfloor$ vertices. Let $w$ be any $t$-divider of $T$. Then $w$ must belong to $T_i$ if $w \in T_j$, $j \neq i$, its removal from $T$ would generate the subtree containing $T_i$, which has size $> \lfloor((t - 1)/t)n\rfloor$, contradicting the fact that $w$ is a $t$-divider.

Let us now assume that $v$ is a $t$-divider. In view of Lemma 2, $0 \leq h \leq 2$. If $h = 1$, i.e., there exists at least a tree $T_i$ of size equal to $\lfloor((t - 1)/t)n\rfloor$, then each subtree other than $T_i$ does not contain $t$-dividers, because the removal of any of its vertices would generate a tree - including both $v$ and $T_i$ - of size at least $\text{size}(T_i) + 1 > \lfloor((t - 1)/t)n\rfloor$.\[\square\]
Lemma 3. For any free tree $T(V, E)$, any constant $t \geq 2$, and any two $t$-dividers $d_i$ and $d_j$, all the vertices on the path between $d_i$ and $d_j$ are also $t$-dividers.

Proof. Let us consider any vertex $u$ along the path from $d_i$ to $d_j$. By definition of $t$-divider, the subtree of $T$ containing both $u$ and $d_j$ induced by the removal of $d_i$ has size $\leq \lfloor \frac{(t-1)/t}{n} \rfloor$. The same holds for the subtree of $T$ containing both $u$ and $d_i$ induced by the removal of $d_j$. This immediately implies that $u$ is a $t$-divider. (See also Fig. 4.)

Corollary 2. For any free tree $T(V, E)$ and any constant $t \geq 2$, the subgraph induced by the $t$-dividers of $T$ is connected.

Finding $t$-dividers. An algorithm for finding a $t$-divider of a tree can be immediately derived from case 1 of Theorem 2. The algorithm starts from any vertex $v$ of the tree and checks if $v$ is a $t$-divider. If so, it stops and returns $v$. Otherwise, it iterates on the subtree of maximum size $T_v$ obtained by the removal of $v$; the new iteration starts from the neighbor of $v$ in $T_v$. From now on we call $d$ the $t$-divider found by the algorithm and $T_d$ the last subtree containing $d$ considered during its execution. Hence, $\text{size}(T_d) \geq \lfloor \frac{(t-1)/t}{n} \rfloor$.

In order to generate all the $t$-dividers of the tree, in view of Theorem 2 and Corollary 2, this procedure can be extended by performing a breadth-first search from $t$-divider $d$, terminating each branch of the breadth-first recursion whenever a non-$t$-divider is found. Both finding a $t$-divider and enumerating all the $t$-dividers of a $n$-vertex tree require $O(n)$ time.

$t$-divided forests. The concept of $t$-divider can be easily extended from trees to forests. The following lemma is useful for this purpose:
Lemma 4. Let $\mathcal{F}$ be a forest of $T_1 \cdots T_h$ free trees and let $t \geq 2$ be a constant. If we denote by $f$ the size of the forest, i.e., $f = \sum_{i=1}^{h} \text{size}(T_i)$, then at most one tree has size $\geq [(t - 1)/t]f$.

Proof. Let us suppose that there exists a tree $T_i$ such that $\text{size}(T_i) > [(t - 1)/t]f$. Then:

$$0 \leq \sum_{j=1, j \neq i}^{h} \text{size}(T_j) = f - \text{size}(T_i) \leq f - \left\lfloor \frac{t-1}{t}f \right\rfloor - 1 \leq \left\lceil \frac{f}{t} \right\rceil - 1 $$

since $t \geq 2$. This implies the uniqueness of $T_i$, if it exists. □

We call a $f$-vertex forest $\mathcal{F}$ $t$-divided if and only if all its trees have size $\leq [(t - 1)/t]f$. Given a forest $\mathcal{F}$, in view of Lemma 4 only two cases are possible: either $\mathcal{F}$ is already $t$-divided or it can be $t$-divided by removing a single vertex, that we will call $t$-divider for $\mathcal{F}$. This vertex must be searched in the maximum size tree of $\mathcal{F}$, say $T_i$, and any $t$-divider of $T_i$ is also a $t$-divider for $\mathcal{F}$. We remark that it may also exist a vertex $v \in T_i$ that is a $t$-divider for $\mathcal{F}$ but not for $T_i$.

4. Naïve decomposition approaches

In order to grow a hierarchy tree out of a graph, a simple top-down strategy works as follows: starting from the root of the hierarchy tree, which is a contraction of the whole graph, the vertices of the graph are partitioned by means of a clustering subroutine 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 with the same graph, depending on the clustering algorithm used as subroutine.

In this section we devise two clustering algorithms for partitioning a $n$-vertex tree $T$. Both algorithms are extremely simple and hinge upon the concept of $t$-divider. Their analyses suggest that it is easy to guarantee either logarithmic depth or bounded degree for the hierarchy tree, but not both, except for special classes of trees. This motivates the design of the more sophisticated scheduling-based algorithm presented in Section 5.

Though the definition of $t$-divider holds both on free and on rooted trees, in the course of the presentation we assume that $T$ has been rooted at a vertex $r$. We focus on a generic step during the top-down construction of the hierarchy tree $HT$ and we call $c$ the node of $HT$ considered at that step. Unless otherwise stated, we assume that the subgraph to be clustered $S(c)$ is a tree named $T_c$ and having size $n_c$. 

4.1. Hierarchy trees of logarithmic depth

A straightforward application of the concept of \( t \)-divider leads to the following algorithm:

\[ \text{Algorithm SimpleClustering}(T_c, t). \]

After a \( t \)-divider of \( T_c \) has been found, consider the \( k \) subtrees \( T_1 \cdots T_k \) obtained from \( T_c \) by removing the \( t \)-divider \( d \) and create \( k \) children \( c_1 \cdots c_k \) of node \( c \) in the hierarchy tree: \( \forall i \in [1, k] \) child \( c_i \) covers the vertices in subtree \( T_i \). The \( t \)-divider \( d \) is added back to the cluster having minimum cardinality. (See Fig. 5b.)

As the cardinality of each new cluster is upper bounded by \( \lfloor \frac{(t-1)t}{t} \rfloor n_c \), the hierarchy tree \( HT \) computed by recursively applying algorithm \text{SimpleClustering} has depth \( O(\log n) \). Moreover, let \( u \) be any vertex of the original tree \( T \) and let \( c_u \) be the singleton of \( HT \) associated with vertex \( u \): during the construction of \( HT \) \( u \) is visited in total as many times as the depth of node \( c_u \) in \( HT \), thus giving worst-case running time \( O(n \log n) \) to build the entire hierarchy tree.

It is also worth observing that the subgraph induced by the vertices covered by every cluster in the hierarchy tree is connected. From now on, where there is no ambiguity we refer to this property as connectivity of clusters. As observed in Section 2, hierarchy trees whose clusters are connected are always valid. The stated properties can be resumed as follows:

\[ \text{Remark 1.} \] Let \( T(V, E) \) be a \( n \)-vertex free tree and let \( \Delta \) be its maximum degree. Algorithm \text{SimpleClustering} computes a valid hierarchy tree of \( T \) having depth \( O(\log n) \) and degree \( \leq \Delta \) in \( O(n \log n) \) worst-case running time.
Even if algorithm SimpleClustering can be implemented using off the shelf data structures and generates hierarchy trees with small depth, the structure of the returned hierarchy tree $HT$ may be irregular and may depend too much on the input tree: namely, the degree of the internal nodes of $HT$ may be too large, since it depends on the degree of the $t$-divider found by the algorithm at each step. For instance, the hierarchy tree shown in Fig. 6a is obtained running algorithm SimpleClustering on an 11-vertex star centered at vertex 0.

4.2. Hierarchy trees of bounded degree

In order to generate hierarchy trees with bounded degree $g \geq 2$ we can refine the naïve approach as follows:

**Algorithm** ConnectedClustering($T_c, g, t$). After a $t$-divider of $T_c$ has been found, consider the $k$ subtrees $T_1 \cdots T_k$ obtained from $T_c$ by removing the $t$-divider $d$ and check the value $k$. If $k \leq g$, work exactly as in algorithm SimpleClustering. Otherwise, sort the subtrees in non-increasing order by size: w.l.o.g. let $T_1 \cdots T_g \cdots T_k$ be the sorted sequence. Create $g$ children $c_1 \cdots c_g$ of node $c$ in the hierarchy tree: $\forall i \in [1, g – 1], c_i$ covers the vertices in subtree $T_i$ and $c_g$ covers the vertices of $T' = \{d\} \cup \bigcup_{h=g}^{k} T_h$. (See Fig. 5c.)

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 $k \leq g$). Nothing is guaranteed about the size of $T'$: $HT$ may be therefore very unbalanced up to reach linear height (see Fig. 6b).
Let us now analyze the running time of algorithm ConnectedClustering on the \( n_c \)-vertex tree \( T_c \). W.l.o.g. we assume that in the original tree \( T \) the children of each node are sorted by non-increasing size of the subtree of which they are the root (this can be easily achieved in a \( O(n \log n) \) time preprocessing step). Under this hypothesis, during the execution of algorithm ConnectedClustering on cluster \( c \), no sorting procedure is required and \( O(n_c) \) time is sufficient to perform all the remaining operations. Since the height of \( HT \) can be \( \Theta(n) \) and building the children of any cluster \( c \) requires \( O(n_c) \) time, the total time to grow the entire hierarchy tree is \( O(n^2) \). The previous considerations can be resumed as follows:

**Remark 2.** Let \( T(V,E) \) be a \( n \)-vertex free tree and let \( g \) be the maximum degree required for a hierarchy tree of \( T \). Algorithm ConnectedClustering computes a valid hierarchy tree of \( T \) having depth \( O(n) \) and degree \( \leq g \) in \( O(n^2) \) worst-case running time.

The performances of algorithm ConnectedClustering turn out to be better if the maximum degree \( \Delta \) of tree \( T \) is bounded by a constant. This follows from the fact that the maximum size subtree \( T_1 \), which is not included in \( T' \) since \( g \geq 2 \), has size at least \( \lfloor n_c / \Delta \rfloor \). Hence, \( T' \) contains at most \( \lceil ((\Delta - 1) / \Delta)n_c \rceil \) vertices, which is a fraction of \( n_c \) when \( \Delta \) is constant.

**Remark 3.** Let \( T(V,E) \) be a \( n \)-vertex free tree with constant maximum degree \( \Delta \) and let \( g \) be the maximum degree required for a hierarchy tree of \( T \). Algorithm ConnectedClustering computes a valid hierarchy tree of \( T \) having depth \( O(\log n) \) and degree \( \leq g \) in \( O(n \log n) \) worst-case running time.

### 5. An almost optimal decomposition algorithm

In this section we present a clustering algorithm aimed at overcoming the drawbacks of the naive decomposition strategies discussed in Section 4. The new algorithm, called BalancedClustering, is based on a reduction to a classical partitioning problem concerned with the scheduling of a set of jobs on \( p \) identical machines, known as *minimum multiprocessor scheduling on parallel machines*. This optimization problem is formally stated as follows:

**Instance:** number \( p \) of processors, set of jobs \( J = \{1 \cdots k\} \), each with its own length \( l_j \), \( 1 \leq j \leq k \).

**Solution:** a \( p \)-processor schedule for \( J \), i.e., a function \( \sigma : J \rightarrow [1, p] \) assigning each job to a processor.

**Measure:** schedule makespan, i.e., time necessary to complete the execution of the jobs, that can be expressed as \( \max_{q \in [1, p]} \sum_{j \in J : \sigma(j) = q} l_j \).

Minimum multiprocessor scheduling on parallel machines has both an on-line and an off-line version and is NP-complete even when \( p = 2 \) [12]. A simple approximation algorithm for it consists of considering the jobs one by one and assigning a job to
the machine currently having the smallest load [13]. From now on we will refer to this subroutine as GrahamSchedule. In the off-line case, very good solutions (i.e., approximation ratio $\frac{4}{3} - \frac{1}{3p}$) can be obtained if the jobs are previously sorted in non-increasing order by length, so as to consider longer jobs first.

Let us now come back to the tree partitioning problem. We assume that the degree of the hierarchy tree should be limited by a constant $g \geq 2$ that algorithm BalancedClustering receives as input. Under this assumption, we exploit the idea that the balancing 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.

Since we admit the existence of non-connected clusters, in the rest of this section we assume that the input of the algorithm is a $f$-vertex forest $F$ instead of a single tree. We call $F_1 \cdots F_h$ the $h$ trees in the forest and we assume that they are rooted at vertices $r_1 \cdots r_h$, respectively. The pseudo-code of algorithm BalancedClustering is given in Fig. 7.

The algorithm works on $t$-divided forests. We recall that if $F$ contains a tree of size $\frac{\left\lfloor (t-1)/t \right\rfloor f}{t}$, it can be $t$-divided by removing a single vertex (line 7) as described in Section 3. The trees in the $t$-divided forest are then grouped into clusters so as to guarantee balanced cluster sizes. This is done by exploiting a reduction to the minimum multiprocessor scheduling problem: the number $g$ of clusters to be built represents the number of machines $p$, the trees coincide with the jobs, and the size of a tree is the length of the corresponding job. In view of this reduction, algorithm BalancedClustering uses as a subroutine procedure GrahamSchedule$(X,a,b)$: $X$ is a set of trees that must be grouped into clusters and interval $[a,b]$, $1 \leq a \leq b \leq g$, is a suitably chosen range of clusters with which the trees must be assigned.

In more detail, in the case where $F$ has been $t$-divided by removing a vertex $d$, the algorithm distinguishes between upper and lower trees: the roots of lower trees

...
are the children of the $t$-divider $d$, which is instead added back to the unique upper tree obtained by deleting it (see Fig. 8a). In this situation, the algorithm does not mix upper and lower trees in a same cluster and gives $g_u$ clusters to upper trees and $g_l = g - g_u + 1$ clusters to lower trees: note that $g_l + g_u = g + 1$ since the cluster which contains the $t$-divider, say $c_{gu}$, is allowed to include trees of both types. Number $g_u$ is chosen proportionally to the total size of the upper trees. If $F$ is already $t$-divided at the beginning of the execution, each tree is considered upper and the forest of lower trees $F_l$ remains empty.

5.1. Analysis

Lemma 5. Algorithm BalancedClustering builds valid hierarchy trees.

Proof. It is sufficient to prove that at each step during the construction of the hierarchy tree each tree $F_i$ in the forest $F$ to be partitioned has a representative vertex $r_i$ satisfying the short distance property: the representatives are all connected to a vertex $v$ of the original tree $T$ such that $v \not\in F$, i.e., $v \not\in c$ (see Fig. 8a). Proving that algorithm BalancedClustering produces clusters which maintain such an invariant implies that the hierarchy tree is valid according to Corollary 1.

Observe that the short distance property trivially holds when forest $F$ consists of a single tree: in this case for any disconnected cluster built by the algorithm we can choose $v = d$, since all the trees covered by that cluster contain a neighbor of the $t$-divider $d$.

The proof proceeds by induction on the depth $l$ in the hierarchy tree of the clusters built at a generic step. The short distance property clearly holds in the base step ($l = 1$) due to the observation above. If $l > 1$, the parent of the new clusters has depth $l - 1$.
and by inductive hypothesis satisfies the property, as illustrated in Fig. 8a. As far as $\mathcal{F}$ is partitioned, the invariant is maintained both by clusters containing upper trees and by clusters consisting only of lower trees: this follows from the inductive hypothesis in the former case and can be easily proved choosing $v = d$ in the latter.

It is worth remarking the importance of the distinction between upper and lower trees for obtaining a valid decomposition. Indeed, Fig. 8b shows that arbitrarily mixing upper and lower trees in the same cluster may produce non valid partitions: in the example, where the upper tree $F_2$ and the lower tree $T_2$ are covered by cluster $c_1$ and all the other trees are covered by $c_2$, a non-valid view is obtained as proved by the existence of cycle $<c_1, c_2, c', c_1>$.

In the following we study the running time of algorithm BalancedClustering and the structural properties of the hierarchy trees that it builds. Some preliminary lemmas will be useful at this aim.

**Lemma 6.** If $\mathcal{F}$ is not $t$-divided at the beginning of the execution of algorithm BalancedClustering, then $\text{size}(\mathcal{F}_u) \leq |f/t| \leq \lfloor (t-1)/t \rfloor \leq \text{size}(\mathcal{F}_l)$ after the assignments in lines 8 and 9.

**Proof.** As far as the algorithm for finding $t$-dividers is concerned, $d$ is the first $t$-divider of $\mathcal{F}$ encountered along the path from $r_1$ to $d$. Hence, the size of the subtree of $F_1$ rooted at $d$ is $> \lfloor (t-1)/t \rfloor$. This implies that $\text{size}(\mathcal{F}_u) \leq |f/t|$ after the assignment statement in line 8 of Figure 7. The inequality $\lfloor (t-1)/t \rfloor \leq \text{size}(\mathcal{F}_l)$ immediately follows since $\mathcal{F}_l = \mathcal{F} \setminus \mathcal{F}_u$.

**Corollary 3.** If $\mathcal{F}$ is not $t$-divided at the beginning of the execution of algorithm BalancedClustering, then $1 \leq g_u < g_i \leq g$.

**Proof.** Recall that $g_i = g - g_u + 1$ and that $g_u$ is chosen proportionally to $\text{size}(\mathcal{F}_u)$ (see line 11 in Fig. 7). Lemma 6 completes the proof.

The next two lemmas prove that algorithm BalancedClustering finds balanced partitions. Both the use of $t$-dividers and the choice of the scheduling subroutine are crucial in the proofs.

**Lemma 7.** Let $\mathcal{F}$ be a $f$-vertex forest and let $t \geq 2$ and $g \geq 2$ be two constants. Let $c_1 \cdots c_g$ be the clusters built by algorithm BalancedClustering $(\mathcal{F}, g, t)$ and let $s_1 \cdots s_g$ be their cardinalities. If $\mathcal{F}$ is not $t$-divided at the beginning of the execution of the algorithm, then, $\forall i, j$ such that $i, j \in [1, g_u - 1]$ or $i, j \in [g_u, g]$, $|s_i - s_j| \leq \lfloor (t-1)/t \rfloor$.

**Proof.** Let us first consider a generic pair of clusters built by algorithm BalancedClustering, say $c_i$ and $c_j$, such that $g_u \leq i < j \leq g$. Consider a generic step $k$ during the execution of procedure GrahamSchedule in line 13 of algorithm BalancedClustering. As far as the algorithm is concerned, both clusters $c_i$ and $c_j$ are used by
the second call to procedure GrahamSchedule, yet $c_i$ may be already partially filled with some upper trees (this happens if $i = g_u$).

Let $s_i^k$ and $s_j^k$ be the cardinalities of clusters $c_i$ and $c_j$, respectively, at step $k$ of procedure GrahamSchedule. Let $T_2 \cdots T_q$ be the lower trees of the $t$-divided forest as in Fig. 8a, let $T_k$ be the tree to be scheduled at step $k$, and let $n_k$ be its size. If $T_k$ is added to a cluster other than $c_i$ and $c_j$, then $s_i^{k+1} = s_i^k$ and $s_j^{k+1} = s_j^k$. Otherwise three cases are possible:

- $s_i^k = s_j^k$: $T_k$ can be added either to $c_i$ or to $c_j$. In any case $|s_i^{k+1} - s_j^{k+1}| = n_k \leq \cdots \leq n_1$ since trees are sorted by size in non-increasing order. See also Fig. 9a.
- $s_i^k < s_j^k$: as far as algorithm GrahamSchedule is concerned, $T_k$ is added to cluster $c_i$, therefore, obtaining $s_i^{k+1} = s_i^k + n_k$ and $s_j^{k+1} = s_j^k$. If $s_i^{k+1} \geq s_j^{k+1}$ then it is easy to prove that $|s_i^{k+1} - s_j^{k+1}| \leq n_k$ (see Fig. 9b). Otherwise $|s_i^{k+1} - s_j^{k+1}| < |s_i^k - s_j^k|$ (see Fig. 9c).
- $s_i^k > s_j^k$: this case is symmetric to the previous one.

When all the lower trees of $F$ have been assigned to the clusters, the cardinalities of $c_i$ and $c_j$ differ by the maximum between $n_1$ and the initial content of $c_i$, that is bounded by $\lfloor f/t \rfloor$ due to Lemma 6. Moreover, since $F$ has been $t$-divided, $n_1 \leq \lfloor ((t-1)/t)f \rfloor$. Hence, at the end of the algorithm execution $|s_i - s_j| \leq \lfloor ((t-1)/t)f \rfloor$.

To conclude the proof, observe that the case $1 \leq i, j < g_u$ is trivial, because the total size of the upper trees is $\leq \lfloor ((t-1)/t)f \rfloor$ due to Lemma 6. □

**Lemma 8.** Let $F$ be a $f$-vertex forest and let $t \geq 2$ and $g \geq 2$ be two constants. Let $c_1 \cdots c_g$ be the clusters built by algorithm BalancedClustering $(F, g, t)$ and let $s_1 \cdots s_g$ be their cardinalities. If $F$ is already $t$-divided at the beginning of the execution of the algorithm, then $\forall i, j \in [1, g]$ $|s_i - s_j| \leq \lfloor ((t-1)/t)f \rfloor$.

**Proof.** The proof is very similar to the proof of Lemma 7 and we omit the details. The main differences are the following: the size of each tree of $F$ is at most $\lfloor ((t-1)/t)f \rfloor$ by hypothesis, $g_u = g$ and $g_l = 0$. The clusters, initially empty, are therefore completely filled in by the first call of procedure GrahamSchedule in line 12. □

We are now ready to discuss the performances of algorithm BalancedClustering:
Theorem 3. Let $T(V,E)$ be a $n$-vertex free tree and let $g$ be the required maximum degree of a hierarchy tree of $T$. Algorithm BalancedClustering computes a valid hierarchy tree of $T$ having depth $O(\log n)$ and degree $\leq g$ in $O(n \log n)$ worst-case running time.

Proof. Let $HT$ be the hierarchy tree grown by recursively applying algorithm BalancedClustering. Lemma 5 guarantees that $HT$ is valid. As far as the algorithm works, it should be also clear that the degree of $HT$ is bounded by $g$. Let us now consider a cluster $c$ of $HT$ and let $n_c$ be its cardinality. Let $c_1 \cdots c_g$ be the children of $c$ in $HT$, with cardinalities $s_1 \cdots s_g$, respectively. Let $m$ be the index of a maximum cardinality child of $c$. Our aim is to prove that $s_m$ is a constant fraction of $n_c$: in particular, we show that the specific constant depends on the value of $t$.

We first consider the case where $\mathcal{F} = S(c)$ is not $t$-divided at the beginning of the execution of algorithm BalancedClustering. The claim easily holds if $m < g_i$: Lemma 6 implies that the size of each cluster filled in with only upper trees is at most $\lceil f/t \rceil$. If $m \geq g_i$, let $r$ be an integer in $[g_i,g]$ such that $r \neq m$ (such an index exists due to Corollary 3). Due to Lemma 7:

$$s_m - s_r \leq \left\lceil \frac{t-1}{t} n_c \right\rceil.$$ 

Moreover, since the algorithm partitions the vertices covered by $c$, the following equality holds:

$$\sum_{i=1}^{g} s_i = n_c.$$ 

Summing up the left and right sides of the above inequalities, respectively, it is easy to obtain

$$2s_m + \sum_{i=1, i \neq r, m}^{g} s_i \leq \left\lceil \frac{2t-1}{t} n_c \right\rceil$$

and therefore $s_m \leq \lfloor (2t-1)/2t \rfloor n_c$. A very similar reasoning, with the help of Lemma 8, holds if $\mathcal{F}$ is already $t$-divided at the beginning of the execution of the algorithm. Hence, since the cardinality of each cluster of the hierarchy tree is a constant fraction of the cardinality of its parent, the depth of $HT$ is clearly $O(\log n)$.

To conclude, we discuss the running time of algorithm BalancedClustering $(\mathcal{F},g,t)$. We denote by $f$ and $h$ the number of vertices and the number of trees of forest $\mathcal{F}$, respectively. Since each tree is non-empty, $h \leq f$ and lines 2 to 11 in Fig. 7 can be easily implemented in $O(f)$ time. If we assume that the children of each node of tree $T$ are sorted by non-increasing size of the subtree of which they are the root, each call to procedure GrahamSchedule considers the trees in the given ordering and, for each tree, decides to which cluster it must be added by selecting the cluster currently having the smallest cardinality (this choice can be done in $O(1)$ time since $g$ is a constant). Hence, lines 12 and 13 in Fig. 7 can be implemented in $O(f)$ time, as well.
In conclusion, during the recursive application of algorithm BalancedClustering to build the entire hierarchy tree, each vertex of tree $T$ is visited in total as many times as its depth in $HT$, thus giving total running time $O(n \log n)$. 

The hierarchy tree shown in Fig. 6c is obtained running algorithm Balanced-Clustering on an 11-vertex star. The example shows that algorithm Balanced-Clustering is able to effectively balance the cardinalities of clusters. This good result is obtained in spite of losing the property of connectivity. However, it is worth remarking that the disconnectivity of clusters remains “weak”, because for each pair of disconnected components in the same cluster there exist two representative vertices whose distance on the original tree is equal to 2 (see Corollary 1 and the invariant property discussed in the proof of Lemma 5).

6. Concluding remarks

In this paper we have considered the problem of computing hierarchical decompositions of trees. We have introduced the concept of $t$-divider, that generalizes the well-known concepts of centroids and separators, and we have designed new tree partitioning algorithms hinging upon $t$-dividers. All the algorithms work by vertex deletion and aim at optimizing different features of the hierarchy tree: among them, depth, maximum degree, and balancing deserve special attention.

We have shown that bounded degree of the hierarchy tree is difficult to achieve if small depth and balanced cluster size have to be guaranteed: this is especially true when the maximum degree of the tree to be clustered is not constant and if disconnected clusters are not allowed. We refer to Remarks 1–3 in Section 4 for a detailed description of the performances of the naïve partitioning approaches. We have then proved that all the above criteria can be optimized simultaneously if one is willing to give up internally connected clusters. This idea, together with a reduction to a classical scheduling problem, allowed us to design a partitioning algorithm that, given a $n$-vertex tree $T$, computes in $O(n \log n)$ time a balanced hierarchy tree of $T$ having bounded degree and logarithmic depth (see Lemmas 7 and 8, and Theorem 3 in Section 5 for details). We remark that $\Omega(n)$ is a trivial lower bound on the construction of any hierarchy tree of $T$.

All our algorithms also guarantee to build valid hierarchy trees, i.e., guarantee that any contraction of the original tree on its hierarchy tree is itself a tree. This is especially relevant in graph visualization applications, where hierarchical decomposition are commonly used. Building valid hierarchy trees may be difficult when disconnected clusters are allowed, and most of the partitioning algorithms from the literature that find disconnected clusters do not guarantee this property.

In our opinion it would be interesting to extend this work towards two main directions. First, a generalization of the concepts and ideas presented throughout the paper to vertex- and edge-weighted trees would be a valuable improvement. This would make it possible to apply the $t$-divider based algorithms to the block cut-vertex tree of a graph, assigning the weight of a tree vertex with the size of the corresponding biconnected
component. We remark that computing a partition of the block cut-vertex tree is a useful preliminary step in many graph decomposition algorithms.

Furthermore, an extensive experimental study of tree decomposition algorithms could be useful to point out benefits and disadvantages of partitioning strategies based either on vertex or on edge deletion. Some preliminary results along this line are reported in [10], showing that the use of centroids, that is a common option for partitioning algorithms based on vertex deletion, does not usually yield the best solution in practice. The best choice for \( t \) in most tests is \( t = 3 \), i.e., partitioning using separators yields the best hierarchy trees with respect to balancing, depth, and degree.

Acknowledgements

The authors wish to thank three anonymous referees for their careful review of the paper and for their constructive comments.

References


