Straight-line Drawings of Binary Trees with Linear Area and Good Aspect Ratio*

Ashim Garg Adrian Rusu

Department of Computer Science and Engineering
University at Buffalo
Buffalo, NY 14260
{agarg,adirusu}@cse.buffalo.edu

Abstract

Trees are usually drawn planar, i.e. without any crossings. In this paper we investigate the area requirement of (non-upward) planar straight-line drawings of binary trees. Let T be a binary tree with n vertices. We show that T admits a planar straight-line grid drawing with area $O(n)$ and with any prespecified aspect ratio in the range $[1, n^\alpha]$, where α is a constant such that $0 \leq \alpha < 1$. We also show that such a drawing can be constructed in $O(n \log n)$ time.

1 Introduction

Trees are very common data-structures , which can be used to model information in a variety of applications such as Software Engineering (hierarchies of object-oriented programs), Business Administration (organization charts), and Web-site Design (structure of a Web-site).

A *drawing* Γ of a rooted tree T maps each node of T to a distinct point in the plane and each edge (u, v) of T to a simple Jordan curve with endpoints u and v. Γ is a *straight-line* drawing if each edge is drawn as a single straight-line segment. Γ is a *polyline* drawing if each edge is drawn as a connected sequence of one or more line-segments, where the meeting point of consecutive line-segments is called a *bend*. Γ is an *orthogonal* drawing if each edge is a chain of alternating horizontal and vertical segments. Γ is a *grid* drawing if all the vertices and bends along the edges have integer coordinates. Γ is a *planar* drawings if edges do not intersect each other in the drawing. Γ is an *upward* drawing if a parent is always assigned either the same or higher y-coordinate than its children. In this paper, we concentrate on grid drawings. So we will assume that the plane is covered by a rectangular grid. The *area* of a drawing Γ is the number of grid points contained in the smallest rectangle with sides parallel to the axes, that covers the entire drawing. We say that R is the *enclosing rectangle* of Γ. The *width* and *height* of Γ are the *width* and *height* of R, respectively. The *enclosing rectangle* of Γ is the smallest rectangle, with sides parallel to the X and Y axes, respectively, that covers the entire drawing. The *aspect ratio* of Γ is the ratio of the longer and shoter sides of its enclosing rectangle. The *degree* of a node of T is the number of edges incident with it. T is a binary tree if each node has at most two children. T is an *ordered* tree if the children of each node are assigned a left-to-right order. We denote by $T[v]$, the *subtree* of T rooted at a node v of T. $T(v)$ consists of v and all the descendents of v. Γ has the *subtree separation* [1] property if for any two vertex-disjoint subtrees $T[u]$ and $T[v]$ of T, the enclosing rectangles of the drawings of $T[u]$ and $T[v]$ do not overlap with each other. It is desirable for the drawing to have the subtree separation property [1] since the users can easily detect the subtrees in such a drawing, and also, it allows for focus+context style [8] rendering of the drawing so that if the tree has too many nodes to fit in the given drawing area, then subtrees close to focus can be shown in detail, whereas those further away from the focus can be contracted and simly shown as filled-in rectangles.

*Research supported by NSF CAREER Award IIS-9985136 and NSF CISE Research Infrastructure Award No. 0101244.

2 Our Result

It is desirable to construct straight-line drawings because straight-line edges are easier to follow than polyline edges [1] and hence they are more aesthetically pleasing than polyline drawings. It is also desirable to construct planar drawings because they are visually more appealing than non-planar drawings, i.e., drawings with edge-crossings. Grid drawings guarantee at least unit distance separation between nodes, and the integer coordinates of nodes allow such drawings to be rendered without any distortions due to truncation and rounding-off errors. Several display surfaces, such as computer screen and plain paper, have predefined aspect ratio. Besides, the drawing area available to the user in an application may also have a certain fixed aspect ratio. Therefore, it is desirable to give user control over the aspect ratio of a drawing, so that they can draw a tree in the drawing area available to them. Also, as mentioned before, it is also desirable for the drawing to exhibit the subtree separartion property. Finally, it is important to minimize the area of a drawing, so that the users can draw a tree in as small drawing area as possible.

In this paper, we investigate the problem of constructing planar straight-line grid drawings of binary trees with small area. Clearly, any planar grid drawing of a binary tree with n nodes requires at least n grid-points to place the nodes, and therefore, has area $\Omega(n)$. A long-standing fundamental question, therefore, has been that whether this is a tight bound, i.e., given a binary tree T with n nodes, can we construct a planar straight-line grid drawing of T with area $O(n)$?

In this paper, we answer this question in affirmative, by giving an algorithm that constructs a planar straight-line drawing of a binary tree with n nodes with $O(n)$ area in $O(n \log n)$ time. More over, the drawing can be parameterized for its aspect ratio, i.e., for any constant α, where $0 \leq \alpha < 1$, the algorithm can construct a drawing with any user-specified aspect ratio in the range $[1, n^{\alpha}]$. (We define the aspect ratio of a drawing to be the ratio of its longer and shorter dimensions. So its aspect ratio will always be atleast 1.) The drawing also exhibits the subtree separation property.

3 Previous Results

Research on drawing trees has a rich history [4]. Previously, the best-known bound on area of planar straight-line grid drawings of binary trees was $O(n \log\log n)$, which was shown in [1] and [9]. This bound is very close to $O(n)$, but still it does not settle the question whether binary trees can be drawn in this fashion in $O(n)$ area. Thus, our result is significant from a theoretical view-point. In fact, we already know of one category of drawings, namely, planar upward orthogonal grid drawings, for which $n \log\log n$ is a tight bound [5], i.e., any binary tree can be drawn in this fashion in area $O(n \log\log n)$, and there exists a family of binary trees that require area $\Omega(n \log\log n)$ in any such drawing. So, a natural question arises, if $n \log\log n$ is a tight bound for planar straight-line grid drawings also. Of course, our result imply that this is not the case. Besides, our drawing technique and proofs are significantly different from those of [1] and [9].

We now summarize some other known results on planar grid drawings of binary trees. Let T be a binary tree. [5] presents an algorithm for constructing an upward polyline drawing of T in $O(n)$ area. [6] and[11] present algorithms for constructing an orthogonal drawing of T in $O(n)$ area. [7] gives an algorithm for constructing upward layered drawing in $O(n^2)$ area. [2] gives an algorithm for constructing strictly upward drawing drawing of T in $O(n \log n)$ area. If T is a Fibonacci tree, (AVL tree, balanced tree, respectively), then [2, 10] ([3],[2], respectively) give algorithms for constructing an strictly upward drawing of T in $O(n)$ area.

4 Preliminaries

In this section we give definitions and some known results that will be used throughout the paper.

Throughout this paper, by the term *drawing*, we will mean a planar straight-line grid drawing. We will assume that the plane is covered by an infinite rectangular grid. A *horizontal channel* (*vertical channel*) is an infinite line parallel to X- and Y- axes, respectively, passing through the grid-points.

Let T be a tree, with one distinguished vertex v, which has at most one child. v is called the *link* vertex of T. A *partial tree* of T is a connected subgraph of T. If T is ordered, then the *leftmost path* p of T is the maximal path consisting of nodes that are leftmost children, except the first one, which is the root of T.

The last vertex of p is called the *leftmost* vertex of T. The *sibling* of a node v of T is the node that has the same parent as v. We denote by $n(T)$, the number of vertices in T.

Let R be a rectangle. Let J be the longer dimension of R, and I be its shorter dimension. Aspect ratio of R, denoted by $ar(R)$ is equal to J/I. Thus, $ar(R) \geq 1$. R has a *good* aspect ratio, if its aspect ratio is in the range $[1, n^\alpha]$, where α is a constant, such that $0 \leq \alpha < 1$. Throughout this paper, we will assume that a rectangle has sides parallel to X and Y axes, respectively.

Let Γ be a drawing of T. We denote by $R(\Gamma)$ the rectangle enclosing a drawing Γ of T. By *bottom* (*top*, *left*, and *right*, respectively) of Γ, we will mean the *bottom boundary* (*top*, *left*, and *right* boundaries, respectively) of its enclosing rectangle. Similarly, by *top-left* corner (*top-right*, *bottom-left*, and *bottom-right* corners, respectively) of Γ, we mean the *top-left* corner (*top-right*, *bottom-left*, and *bottom-right* corners, respectively) of its enclosing rectangle. The *aspect ratio* of Γ is same as the aspect ratio of its enclosing rectangle.

Let r be the root of T. Let u^* be the link vertex of T. Suppose T has n nodes. Γ is a *feasible* drawing of T, if it has the following properties:

- **Property 1**: r is placed at the top-left corner of Γ,
- **Property 2**: if $u^* \neq r$, then u^* is placed at the bottom of Γ, and
- **Property 3**: if $u^* = r$, then no other node or edge of T is placed on or crosses the vertical and horizontal channels occupied by r.

We denote by E the *empty drawing*, i.e., drawing of an empty tree, i.e., tree with zero vertices.

The following theorem is a small modification of Theorem 6 of [11] (Theorem 6 requires that $\alpha \leq x \leq 1-\alpha$, whereas here we just require that $0 < x \leq 1-\alpha$).

Theorem 1 ([11]) *Let n_0 and α be two constants, where $n_0 \geq 1$ and $0 < \alpha < 1/2$. Let $m_1, m_2, \ldots, m_{n_0}$ be a set of positive numbers. Let $G(n)$, and $g(n)$ be two functions such that*

- *$\forall n > n_0$, and $\forall x$, where $0 < x \leq 1-\alpha$, $G(xn) \leq x(G(n) - g(n)\sqrt{G(n)})$, and*
- *$\forall n \leq n_0$, $G(n) \geq m_n$.*

Then, if $g(n) \leq cn^\beta$, where $c > 0$ and β are constants, and $0 \leq \beta < 1/2$, then $G(n) = O(n)$. More specifically, $G(n) = \gamma n - \delta\sqrt{n}g(n)$, where γ and δ are constants that satisfy the following conditions:

- *$\forall n \leq n_0$, $\gamma - \delta\sqrt{n}cn^\beta \geq m_n$, and*
- *$\delta(\mu^{1/2-\beta} - 1) > \sqrt{\gamma}$, where $\mu = 1/(1-\alpha)$.*

The following theorem is based on Lemma 2 of [11].

Theorem 2 (Separation Theorem [11]) *In any tree T with degree d, there is an edge e whose removal splits T into two trees T_1 and T_2, such that for some x, $1/d \leq x \leq (d-1)/d$, $n(T_1) \leq x \cdot n(T)$ and $n(T_2) \leq (1-x) \cdot n(T)$. Edge e is called a* separator *of T. More over, e can be found in $O(n(T))$ time.*

The following corollary of Theorem 2 deals with binary trees:

Corollary 1 *The removal of a separator edge e of a binary tree T splits T into two trees T_1 and T_2, such that for some x, $1/3 \leq x \leq 2/3$, $n(T_1) \leq x \cdot n(T)$, and $n(T_2) \leq (1-x) \cdot n(T)$. More over, e can be found in $O(n(T))$ time.*

Let v be a node of tree T located at grid position (i, j) in Γ. Let Γ be a drawing of T. Assume that the root r of T is located at the grid position $(0, 0)$ in Γ. We define the following operations on Γ (see Figure 1):

- *rotate operation*: rotate Γ counterclockwise by δ degrees.
  After a rotation by δ degrees of Γ, node v is located at grid position $(i\cos\delta - j\sin\delta, i\sin\delta + j\cos\delta)$.
  In particular, after a rotation by $90°$ of Γ, node v is located at grid position $(-j, i)$.
- *flip operation*: flip Γ vertically or horizontally.
  After a horizontal flip of Γ, node v is located at grid position $(-i, j)$.
  After a vertical flip of Γ, node v is located at grid position $(i, -j)$.
- *stretch to right* operation: after this operation, vertex v will move to the grid position (ki, j), where $k \geq 1$ is an integer, called the *stretch factor*.

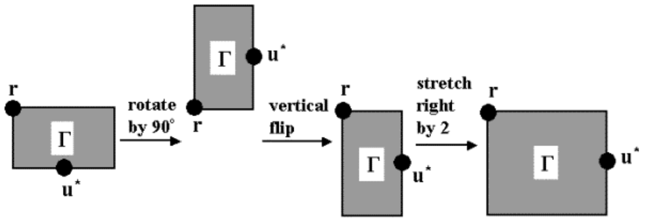


Figure 1: Rotating a drawing Γ first by $90°$, followed by flipping it vertically, followed by stretching it to right with the stretch factor of 2. Note that initially vertex u^* was located on the bottom boundary of Γ and after the flip operation u^* is located on the right boundary of Γ.

4.1 Our Tree Drawing Algorithm

Let T be a binary tree with a link vertex u^*. Let n be the number of nodes in T. Let A and ϵ be two numbers such that $1/2 < \epsilon < 1$, and A is in the range $[1, n^\epsilon]$. Let R be a large enough rectangle with aspect ratio A, width W and height H (we will determine later on, how large R should be). Notice that because $1 \leq A < n^\epsilon$, R has a good aspect ratio.

Our tree drawing algorithm, called *DrawTree*, uses the divide-and-conquer paradigm to recursively construct a feasible drawing of T within R. In each recursive step, it splits T into at most five partial trees by removing at most two vertices and the edges incident on these vertices from it. Each partial tree has at most $(2/3)n$ nodes. Correspondingly, it splits R into at most five smaller rectangles, and assign each smaller rectangle to a partial tree. It recursively draws each partial tree into its assigned rectangle, such that the drawings constructed of the partial trees are feasible drawings. Finally, it draws the two vertices and the edges within R, arranging the drawings of the five partial trees such that there are no edge crossings, and the drawing of the entire tree T fits within R.

The assignment of a rectangle R_k to a partial tree T_k, and the drawing of T_k within R_k is done based on the relative size of T_k compared to the size of T, i.e., on the ratio $x = n_k/n$, where n_k is the number of nodes in T_k: Assume that $W \leq H$ (the case when $H < W$ is analogous). Let $W' = W - 2$, and $H' = H - 4n^{1-\epsilon} - 4(\log_2 n + 3) - 2$.

- If T_k has a very small size, i.e., if $n_k < W'$, then R_k is a rectangle with width W' and height $\log_2 n_k + 3$. T_k is drawn within R_k using an algorithm called u^*-*HV-Draw*.
- Otherwise, if T_k has a moderate size, i.e., if $xH'/W' < {n_k}^{-\epsilon}$, then R_k is a rectangle with width W' and height W'/n_k^ϵ. T_k is drawn recursively within R_k by calling Algorithm *DrawTree* for it.
- Otherwise, T_k has a large enough size, i.e., $xH'/W' \geq n_k^{-\epsilon}$. Then R_k is a rectangle with width W' and height xH'. T_k is drawn recursively within R_k by calling Algorithm *DrawTree* for it.

This assignment and drawing strategy ensures that the total size of the rectangles assigned to the partial trees is slightly less than the size of R, so that after drawing the partial trees within their respective rectangles, we can put together their drawings, along with the drawings of the removed vertices and the edges, to get a drawing of T that fits within R. It also ensures that, except for the case when T_k has very small size, R_k has a good aspect ratio. This property is crucial for proving that T can be drawn in $O(n)$ area, as shown in the proof of lemma 4. Also, notice that if $W \leq H$ ($H < W$), then all the partial trees will get rectangles with equal width (height), which is equal to $W - 2$ ($H - 2$), and possibly different heights (widths).

Algorithm u^*-*HV-Draw* is a modification of the classical HV-Drawing algorithm [2], and can be used to construct a feasible drawing of a tree T' with n' nodes within a rectangle with dimensions n' and $\log_2 n' + 3$, where users can specify whether the longer dimension is width or height:

Lemma 1 *Let T be a n-node binary tree with a link vertex u^*. Algorithm u^*-*HV-Draw *can be used to construct in $O(n)$ time, a feasible drawing of T within a rectangle with width n and $\log_2 n + 3$, as well as within a rectangle with width $\log_2 n + 3$ and height n.*

Details of Algorithm u^*-*HV-Draw* are given in the Appendix. (Lemma 1 is Lemma 6 in the Appendix.)

To split T into the partial trees, *DrawTree* first orders the children of each node, such that u^* becomes the leftmost vertex of T. This can be easily done in $O(n)$ time by making all the nodes, except the root, of the path from u^* to root as left children of their parents, making the siblings of these nodes as the right children of their parents, if u^* has a child, then making it the right child of u^*, and randomly making the other nodes as the right or left children of their parents such that each node has at most one right child and at most one left child.

Next, it finds a separator edge (u, v) of T, where v is the parent of u. Based on whether, or not, (u, v) is in the leftmost path of T, we get two cases:

- *Case 1: The separator edge (u, v) is not in the leftmost path of T.* We get four subcases: (a) In the general case, T has the form as shown in Figure 2(a). In this figure,
  - r is the root of T,
  - T_2 is the tree rooted at v,
  - c is the sibling of u, and T_1 is the tree rooted at c,
  - T_α is the tree rooted at u,
  - w is the parent of u,
  - a is the last common vertex of the path $r \rightsquigarrow v$ and the leftmost path of T,
  - f is the right child of a,
  - T_β is the maximal tree rooted at f that contains w but not u,
  - T_B is the tree consisting of the trees T_α and T_β, and the edge (w, u),
  - e is the parent of a, and d is the left child of a,
  - T_A is the maximal tree rooted at r that contains e but not a, and
  - T_C is the tree rooted at d.

  In addition to this general case, we get three special cases: (b) when $T_A \neq \emptyset$ and $T_C = \emptyset$ (see Figure 2(b)), (c) when $T_A = \emptyset$ and $T_C \neq \emptyset$(see Figure 2(c)), and (d) when $T_A = \emptyset$ and $T_C = \emptyset$ (see Figure 2(d)). Note that, since (u, v) is a separator edge, from Corollary 1, T_2 has at least $(1/3)n$ nodes and so $T_B \neq \emptyset$. Some of these subcases have further subsubcases. In all the cases, we designate e as the link vertex of T_A, w as the link vertex of T_β, and u^* as the link vertex of T_C. We randomly choose a leaf of T_1, and a leaf of T_2, and designate them as the link vertices of T_1 and T_2, respectively.
- *Case 2: The separator edge (u, v) is in the leftmost path of T.* In this case, T has the general form in Figure 3(a), where
  - r is the root of T,
  - c is the right child of u,
  - T_B is the tree rooted at c,
  - e is the parent of u,
  - T_A is the maximal tree rooted at r that contains e but not u. vertex, and
  - $T_C = T_2$ is the tree rooted at v.

  In addition to the general case, we get the following three special cases: (b) when $T_A = \emptyset$ and $T_B \neq \emptyset$ (see Figure 3(b)), (c) when $T_A \neq \emptyset$ and $T_B = \emptyset$ (see Figure 3(c)), and (d) when $T_A = \emptyset$ and $T_B = \emptyset$ (see Figure 3(d)). Again, note that, since (u, v) is a separator edge, by Corollary 1, $T_C \neq \emptyset$. Some of these subcases have further subsubcases. In all these cases, we designate e as the link vertex of T_A, and u^* as the link vertex of T_C. We randomly choose a leaf of T_B and designate it as the link vertex of T_B.

In each case, the at most five partial subtrees that T gets split into are T_A, T_C, T_β, T_1, and T_2. a and u are the at most two vertices whose removal splits T into these partial trees. Note that the edges that also get removed when we remove a and u are (a, e), (d, a), (f, a), (u, w), (v, u), (c, u).

We give below a pseudocode for Algorithm *DrawTree*. In this pseudocode, it assigns rectangles to T_A, T_C, T_β, T_1, and T_2 by calling Algorithm *AssignRectangle*. It calls itself recursively to draw T_A and T_C, and calls Algorithm *DrawTreeB* to draw tree T_B. In the end, it calls Algorithm *Comp* to put together the drawings of T_A, T_B and T_C, and to draw a and its incident edges, to construct a drawing of T. If $W \leq H$, then *Comp* stacks the drawings of T_A, T_B and T_C one above the other, and places a appropriately such that there are no edge crossings when drawing edges incident on a, as shown in the second column of Figures 6

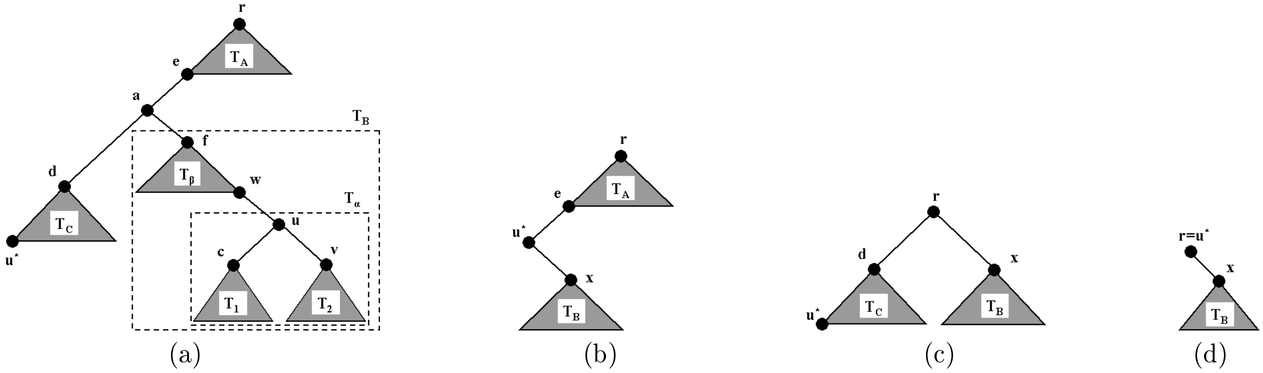


Figure 2: Case 1 (Separator edge (u, v) does not belong to the leftmost path of T): (a) General form, i.e., $T_A \neq \emptyset$ and $T_C \neq \emptyset$; (b) $T_A \neq \emptyset$ and $T_C = \emptyset$; (c) $T_A = \emptyset$ and and $T_C \neq \emptyset$; (d) $T_A = \emptyset$ and $T_C = \emptyset$.

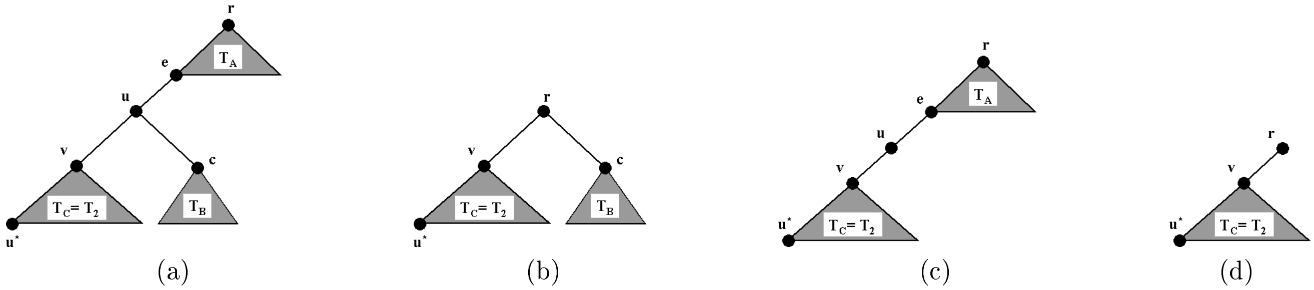


Figure 3: Case 2 (Separator edge (u, v) belongs to the leftmost path of T): (a) General form, i.e., $T_A \neq \emptyset$, $T_B \neq \emptyset$; (b) $T_A = \emptyset$ and $T_B \neq \emptyset$. (c) $T_A \neq \emptyset$ and $T_B = \emptyset$; (d) $T_A = \emptyset$, $T_B = \emptyset$.

and 7. If $H < W$, then *Comp* puts the drawings of T_A, T_B and T_C side-by-side, and places a appropriately such that there are no edge crossings when drawing edges incident on a, as shown in the third column of Figures 6 and 7. Details of Algorithm *Comp* are given in the Appendix.

DrawTreeB calls *DrawTree* to draw T_β, and calls Algorithm *DrawTreeα* to draw T_α. In the end, it calls Algorithm *CompB* to put together the drawings of T_α, and T_β and to draw the edge (u, w), to construct a feasible drawing of T_B. From Corollary 1, $T_\alpha \neq \emptyset$. Hence, we have two cases: (a) $T_\beta \neq \emptyset$, and (b) $T_\beta = \emptyset$ (see Figure 5). When $T_\beta \neq \emptyset$, then if $W \leq H$ ($H < W$), *CompB* puts the drawings of T_β above (to the left of) T_α at unit distance from it, such that their left (top) boundaries are aligned, as shown in the second (third) column of Figure 5. Then it draws the edge that connects w with the root of T_α. When $T_\beta = \emptyset$, then *CompB* simply returns the drawing of T_α as the drawing of T_B.

DrawTreeα calls *DrawTree* to draw T_1 and T_2. In the end, it calls Algorithm *Compα* to put together the drawings of T_1, and T_2, and to draw u and its incident edges, to construct a drawing of T_α. From Corollary 1, $T_2 \neq \emptyset$. We have three cases (see Figure 4): (a) $u \notin$ the left path of T and $T_1 \neq \emptyset$, (b) $u \notin$ the left path of T and $T_1 = \emptyset$, and (c) $u \in$ the left path of T. In Case (a), if $W \leq H$ ($H < W$), as shown in the second (third) column of Figure 4(a), *Compα* puts the drawing Γ_1 of T_1 above (to the left of) the drawing Γ_2 of T_2 at unit distance from it. It places u one unit to the left of and one unit above Γ_1. Then it draws edges (u, c) and (u, v). In Case (b), it just places u one unit to left and one unit above Γ_1 (see Figure 4(b)). Then it draws edge (u, v). In Case (c), it simly returns Γ_2 as the drawing of T_α.

Algorithm DrawTree

Input: A binary tree T with the link vertex u^*, a constant ϵ, where $1/2 < \epsilon < 1$, and a rectangle R with aspect ratio A, where $1 \leq A \leq n^\epsilon$. Let W and H be the width and height, respectively, of R

Output: A feasible drawing Γ of T within R.

if $n(T) = 0$ **then** Return E (Base case 1) ; (base return Empty Drawing)
else if $n(T) = 1$ (base case 2) **then** {
  Let Γ be the drawing containing exactly one vertex;
  Return Γ;
}
Order the children of each node, as its left and right children, such that u^* becomes the leftmost vertex of T. This can be easily done in $O(n)$ time by making all the nodes, except the root, of the path from u^* to root as left children of their parents, making the siblings of these nodes as the right children of their parents, if u^* has a child, then making it the right child of u^*, and randomly making the other nodes as the right or left children of their parents such that each node has at most one right child and at most one left child.
Find a separator edge (u, v) of T.
if $(u, v) \notin$ leftmost path of T **then** {
  Identify trees T_A, T_B, T_C, T_α, T_β, T_1, and T_2 as in Figure 2;
  Designate vertex e as the link vertex of T_A, w as the link vertex of T_β, and u^* as the link vertex of T_C (See Figure 2). Randomly choose a leaf of T_1, and a leaf of T_2, and designate them as the link vertices of T_1 and T_2, repectively.
}
else {
  Identify trees T_A, T_B, and T_C as in Figure 3;
  Designate e as the link vertex of T_A, and u^* as the link vertex of T_C (See Figure 3). We randomly choose a leaf of T_B and designate it as the link vertex of T_B.
}
Let $R(T_A)$ =AssignRectangle(T_A, T, R, ϵ);
Let $R(T_C)$ =AssignRectangle(T_C, T, R, ϵ);
if $(u, v) \notin$ leftmost path of T **then** {
  Let $R(T_\alpha)$ =AssignRectangle(T_α, T, R, ϵ);
  Let $R(T_\beta)$ =AssignRectangle(T_β, T, R, ϵ);
  Let $R(T_1)$ =AssignRectangle(T_1, T, R, ϵ);
  Let $R(T_2)$ =AssignRectangle(T_2, T, R, ϵ);
}
else Let $R(T_B)$ =AssignRectangle(T_B, T, R, ϵ);
if $(u, v) \notin$ leftmost path of T and $T_A \neq \emptyset$ and $T_C = \emptyset$, and $r \neq e$ **then** Change the orientation of $R(T_A)$ (i.e. the width of $R(T_A)$ becomes its height and the height of $R(T_A)$ becomes its width) ;
if $(u, v) \notin$ leftmost path of T and $T_\beta \neq \emptyset$ and $H < W$ **then** Change the orientation of $R(T_\beta)$ (i.e. the width of $R(T_\beta)$ becomes its height and the height of $R(T_\beta)$ becomes its width);
Let Γ_A =DrawTree($T_A, R(T_A), \epsilon$);
Let W and H be the width and height, respectively, of R;
if $(u, v) \notin$ leftmost path of T **then** Let Γ_B =DrawTreeB($T_B, R(T_\alpha), R(T_\beta), R(T_1), R(T_2), W, H, \epsilon$);
else Let Γ_B =DrawTree($T_B, R(T_B), \epsilon$);
Let Γ_C =DrawTree($T_C, R(T_C), \epsilon$);
Let Γ =Comp($\Gamma_A, \Gamma_B, \Gamma_C$, R); (see the Appendix for description of Algorithm $Comp$)
Return Γ;

end Algorithm.

Algorithm AssignRectangle

Input: T_k, T, $R(T)$, and ϵ, where $1/2 < \epsilon < 1$. Let I and J be the dimensions of R, where $J \geq I$.
Output: A rectangle $R(T_k)$ whose one dimension is equal to $I-2$ and the other dimension is either equal to $\log_2 n(T_k) + 3$, or is at least $(I-2)/n(T_k)^\epsilon$.

Let $I_k = I' = I - 2$;
Let $J' = J - 4n^{1-\epsilon} - 4(\log_2 n(T) + 3) - 2$;
Let $x = n(T_k)/n(T)$;
if $n(T_k) < I'$ **then** $J_k = \log_2 n(T_k) + 3$;
else if $xJ'/I' < n(T_k)^{-\epsilon}$ **then** $J_k = I'/n(T_k)^\epsilon$;
else $J_k = xJ'$;
Let $R(T_k)$ be a rectangle with dimensions I_k and J_k, where if I is the height of R then I_k is the height of R_k and J_k is its width, otherwise I_k is the width of R_k and J_k is its height;
Return $R(T_k)$;

end Algorithm.

Algorithm DrawTreeB

Input: T_B, $R(T_\alpha)$, $R(T_\beta)$, $R(T_1)$, $R(T_2)$, W, H, and ϵ, where $1/2 < \epsilon < 1$.
Output: A feasible drawing Γ_B of T_B.

Let Γ_α =DrawTree$\alpha(T_\alpha, R(T_1), R(T_2), W, H, \epsilon)$;
Let Γ_β =DrawTree$(T_\beta, R(T_\beta), \epsilon)$;
Let Γ_B =CompB$(\Gamma_\alpha, \Gamma_\beta, W, H)$;
Return Γ_B;

end Algorithm.

Algorithm DrawTreeα

Input: T_α, $R(T_1)$, $R(T_2)$, W, H, and ϵ, where $1/2 < \epsilon < 1$.
Output: A feasible drawing Γ_α of T_α.

Let Γ_1 =DrawTree(T_1, $R(T_1)$, ϵ);
Let Γ_2 =DrawTree(T_2, $R(T_2)$, ϵ);
Let Γ_α =Comp$\alpha(\Gamma_1, \Gamma_2, W, H)$;
Return Γ_α;

end Algorithm.

Algorithm Compα

Input: Γ_1, Γ_2, W, and H.
Output: A feasible drawing Γ_α of T_α.

if $u \notin$ the leftmost path of T and $T_1 \neq \emptyset$ (see Figure 4(a)) **then** {
  if $W \leq H$ **then** {
    place Γ_1 above Γ_2 separated by unit distance such that the left boundaries of Γ_1 and Γ_2 are aligned;
    Place u at one unit to the left of and one unit above Γ_1;

}

else ($H < W$) **then**{

place Γ_1 to the left of Γ_2 separated by unit distance such that the top boundaries of Γ_1 and Γ_2 are aligned;

Place u at one unit to the left of and one unit above Γ_1;

}

Draw edges (u, c) and (u, v);

Let Γ_α be the drawing thus obtained;

}

else if $u \notin$ the leftmost path of T and $T_1 = \emptyset$ (see Figure 4(b)) **then** {

Place u at one unit to the left of and one unit above Γ_2;

Draw edge (u, v);

Let Γ_α be the drawing thus obtained;

}

else ($u \in$ the leftmost path of T (see Figure 4(c))) Let $\Gamma_\alpha = \Gamma_2$;

Return Γ_α;

end Algorithm.

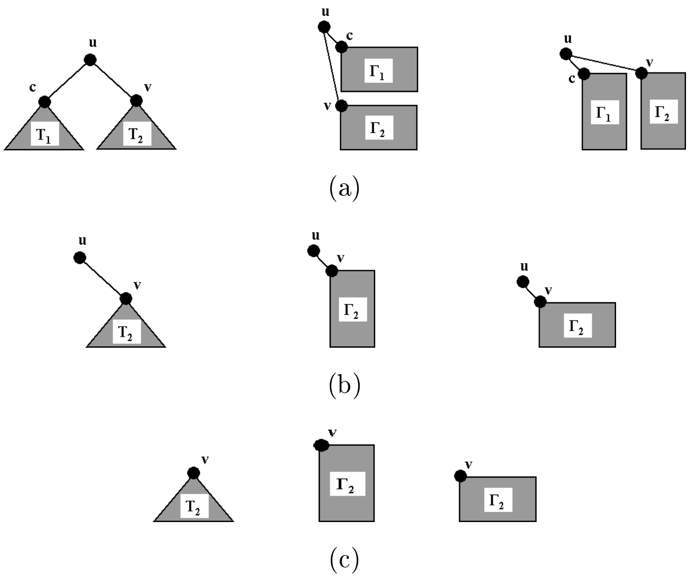


Figure 4: Drawing T_α, when: (a) u does not belong to the leftmost path of T and $T_1 \neq \emptyset$, (b) u does not belong to the leftmost path of T and $T_1 = \emptyset$, and (c) $u \in$ leftmost path of T. The first column shows the three cases. The second column shows the drawings when $W \leq H$. The third column shows the drawings when $H < W$. Here, Γ_1 is the drawing of T_1 and Γ_2 is the drawing of T_1. For convinience, we have shown Γ_1, and Γ_2 as having same the width in the second column and the same height in the third column. In actuality, their heights and widths may be different.

Let

$$y = \begin{cases} u & \text{if } u \text{ does not belong to the leftmost path of } T \\ v & \text{otherwise} \end{cases}$$

Algorithm CompB

Input: Γ_α, Γ_β, W, and H.
Output: A feasible drawing Γ_B of T_B.

if $T_\beta \neq \emptyset$ (see Figure 5(a)) **then** {
    if $W \leq H$ **then** place Γ_β above Γ_α separated by unit distance such that the left boundaries of Γ_β and Γ_α are aligned;
    else $H < W$) **then** {
        First rotate by $90°$ and then flip Γ_β vertically;
        place Γ_β to the left of Γ_α separated by unit distance such that the top boundaries of Γ_β and Γ_α are aligned;
    }
    Draw edge (w, y);
Let Γ_B be the drawing obtained;
}
else ($T_\beta = \emptyset$ (see Figure 5(b))) Let $\Gamma_B = \Gamma_\alpha$;
Return Γ_B;

end Algorithm.

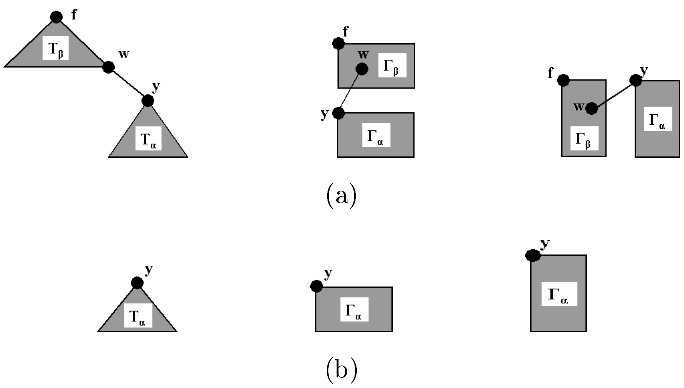


Figure 5: Drawing T_B when: (a) $T_\beta \neq \emptyset$, and (b) $T_\beta = \emptyset$. Here, Γ_β and Γ_α are the drawings of T_β and T_α, respectively. Node y shown here is either node u or v. The first column shows the two cases. The second column shows the drawings when $W \leq H$. The third column shows the drawings when $H < W$. For convinience, we have shown Γ_β, and Γ_α as having same the width in the second column and the same height in the third column. In actuality, their height and widths may be different. For convinience, we have shown w as being somewhere inside Γ_β. Actually, since Γ_beta is a feasible drawing, either $w = f$, or w will be at the bottom of Γ_β.

Recall that

Lemma 2 (Planarity) *Let T be a binary tree with a link vertex u^* given as input to Algorithm* DrawTree*. Algorithm* DrawTree *will construct a feasible drawing Γ of T.*

Proof: Proof is by induction over the size of T, i.e., over the number of nodes n in T.

When T has at most one vertex, this is trivially true (this corresponds to Base Cases 1 and 2 of Algorithm *DrawTree*).

If T gets drawn in Base Case 3 of Algorithm *DrawTree*, then it is drawn using Algorithm *u^*-HV-Draw*. From Lemma 1, this drawing is a feasible drawing.

In the general recursive step, we have two cases: Case 1 (when the separator edge (u, v) is not in the leftmost path), and Case 2 (when the separator edge (u, v) is in the leftmost path).

Consider Case 1. Assume that we have already constructed feasible drawings Γ_A, Γ_β, Γ_1, Γ_2, and Γ_C

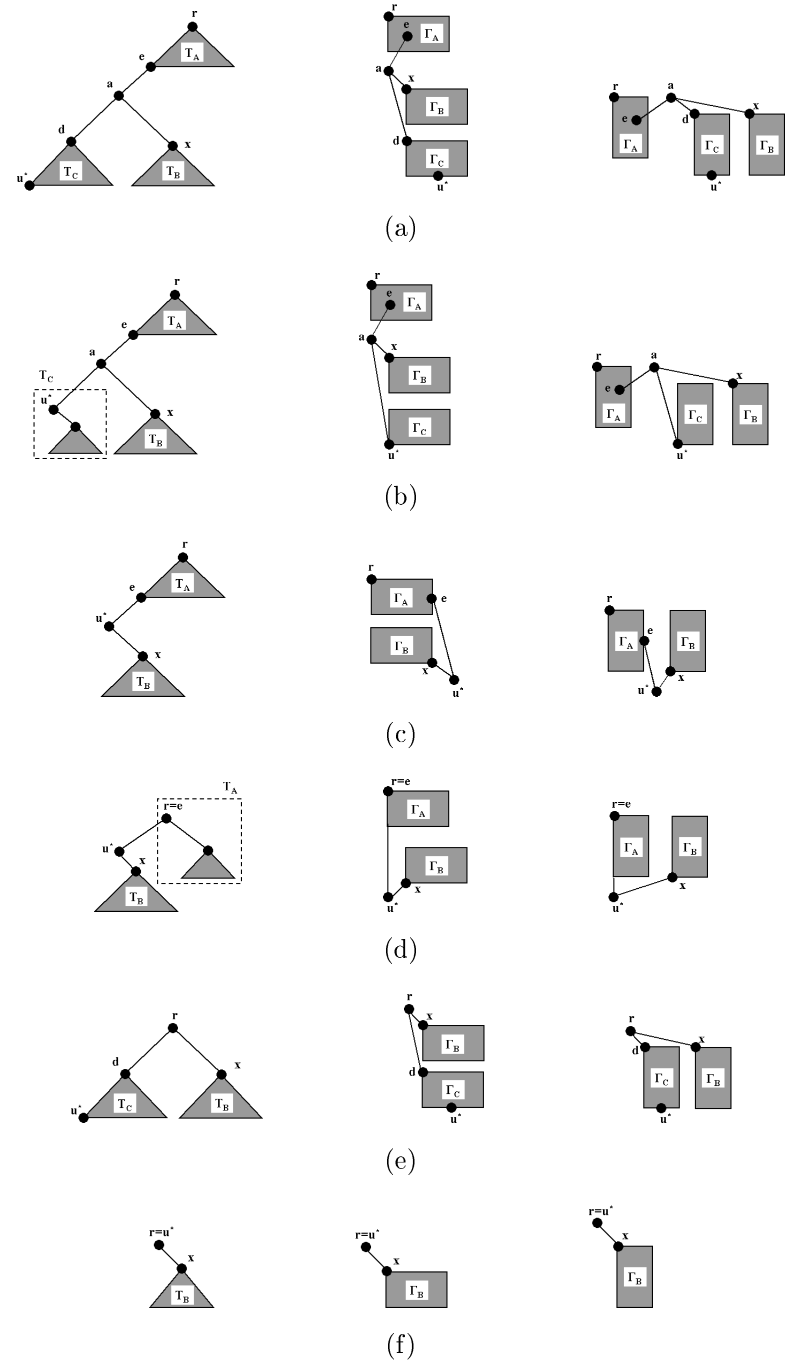
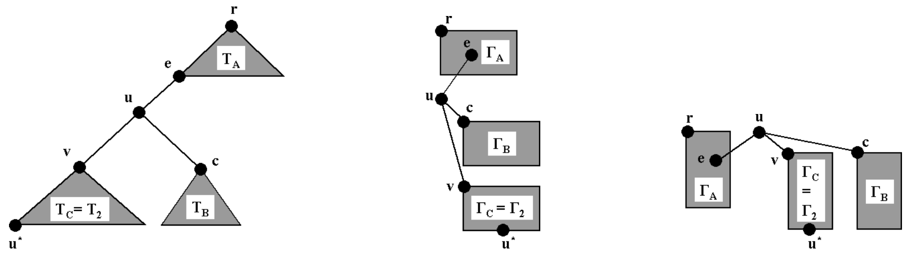
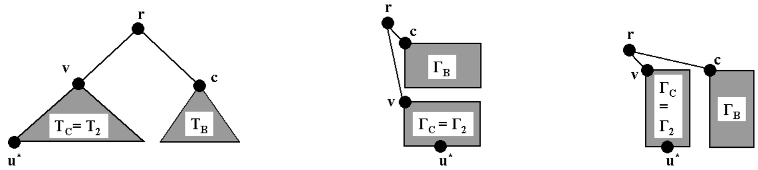


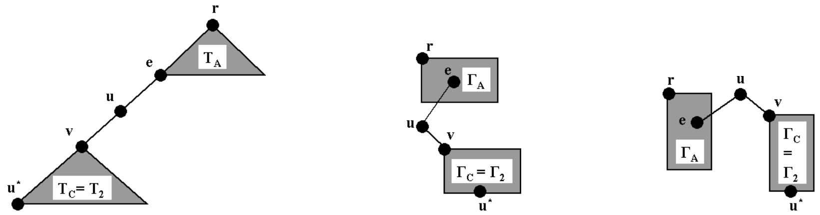
Figure 6: Drawing T in all the subsubcases of Case 1 (when the separator (u, v) is not in the leftmost path of T): (a) $T_A \neq \emptyset$, $T_C \neq \emptyset$, $d \neq u^*$. (b) $T_A \neq \emptyset$, $T_C \neq \emptyset$, $d = u^*$. (c) $T_A \neq \emptyset$, $T_C = \emptyset$, $r \neq e$. (d) $T_A \neq \emptyset$, $T_C = \emptyset$, $r = e$ or $T_A = \emptyset$, $T_C \neq \emptyset$, $d = u^*$. (e) $T_A = \emptyset$, $T_C \neq \emptyset$, $d \neq u^*$. (f) $T_A = \emptyset$, $T_C = \emptyset$. Node x shown here is either node f, or u or v. Here, the first column shows the subcases. The second column shows the drawings when $W \leq H$. The third column shows the drawings when $H < W$. Here, Γ_A, Γ_B and Γ_C are the drawings of T_A, T_B and T_C, respectively. For convinience, we have shown Γ_A, Γ_B and Γ_C as having same the width in the second column and the same height in the third column. Note that when $W \leq H$ ($H < W$), the assigned rectangles of T_A, T_B and T_C will have the same width, but the widths (height) of their actual drawings Γ_A, Γ_B and Γ_C may be different from each other. For convinience, we have shown e as being somewhere inside Γ_A. Actually, since Γ_A is a feasible drawing, either $e = r$, or e will be at the bottom of Γ_A.



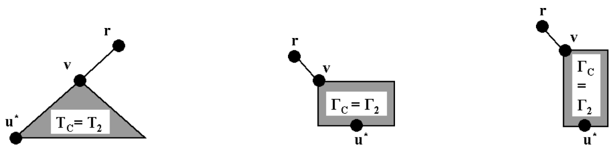
(a)



(b)



(c)



(d)

Figure 7: Drawing T in all the subsubcases of Case 2 (when the separator (u, v) is in the leftmost path of T): (a) $T_A \neq \emptyset$, $T_B \neq \emptyset$. (b) $T_A = \emptyset$, $T_B \neq \emptyset$. (c) $T_A \neq \emptyset$, $T_B = \emptyset$. (d) $T_A = \emptyset$, $T_B = \emptyset$. Here, the first column shows the subcases. The second column shows the drawings when $W \leq H$. The third column shows the drawings when $H < W$. Here, Γ_A, Γ_B and Γ_C are the drawings of T_A, T_B and T_C, respectively. For convinience, we have shown Γ_A, Γ_B and Γ_C as having same the width in the second column and the same height in the third column. Note that when $W \leq H$ $(H < W)$, the assigned rectangles of T_A, T_B and T_C will have the same width, but the widths (height) of their actual drawings Γ_A, Γ_B and Γ_C may be different from each other. For convinience, we have shown e as being somewhere inside Γ_A. Actually, since Γ_A is a feasible drawing, either $e = r$, or e will be at the bottom of Γ_A.

of the partial trees T_A, T_β, T_1, T_2 and T_C, respectively, of T. Recall that a feasible drawing has Properties 1, 2, and 3, as given in the definition of feasible drawing in Section 4.

Because Γ_1 and Γ_2 are feasible drawings, in all the cases, as shown in Figure 4, Algorithm *Compα* can draw edges (u, c) and (u, v) without any edge crossings, and as a result, Drawing Γ_α constructed by Algorithm *DrawTreeα* is a feasible drawing.

Because Γ_β and Γ_α are feasible drawings, in all the cases, as shown in Figure 5, Algorithm *CompB* can draw the edge (w, y) without any edge crossings, and as a result, Drawing Γ_B constructed by Algorithm *DrawTreeB* is also a feasible drawing. Note that in Figure 5, we have shown w to be somewhere inside Γ_β, but because of Properties 1 and 2, it either the same as the root of T_β, or is at the bottom of Γ_β.

Because Γ_A, Γ_B, and Γ_C are feasible drawings, in all the cases, as shown in Figures 6 and 7, Algorithm *Comp* can draw the edge (e, a), (a, d), (a, x), (u, v), (e, u), and (u, c) without any edge crossings, and as a result, Drawing Γ of T constructed by Algorithm *DrawTree* is also a feasible drawing. Note that in Figures 6 and 7, we have shown e to be somewhere inside Γ_A, but because of Properties 2 and 3, it is either the same as the root of T, or is at the bottom of Γ_A.

Now Consider Case 2. Assume that we have already constructed feasible drawings Γ_A, Γ_B, and Γ_C of T_A, T_B, and T_C, respectively. Using the same reasoning as above, we can prove that Algorithm *Comp* will construct a feasible drawing of T in this case also. □

Lemma 3 (Time Complexity) *Let T be a binary tree with a link vertex u^* given as input to Algorithm* DrawTree. *Algorithm DrawTree will construct a drawing Γ of T in $O(n \log n)$ time.*

Proof: By Corrollary 1, in each recursive step, each of the trees T'_A, T'_β, T'_1, T'_2 and T'_C obtained by splitting a tree T' with n' nodes and a link vertex u', contains at most $(2/3)n'$ nodes. Hence, the depth of the recursion of Algorithm *DrawTree* is $O(\log n)$. In each recursive step, it can make u' the left most vertex of T, assign rectangles to T'_A, T'_β, T'_1, T'_2 and T'_C, and combine their drawings to obtain a drawing of T' in $O(n')$ time. Hence, the overall running time of *DrawTree* is $O(n \log n)$. □

Lemma 4 (Area) *Let T be a binary tree with a link vertex u^* given as input to Algorithm* DrawTree. *Algorithm* DrawTree *can construct a drawing of T within a rectangle with $O(n)$ area.*

Proof:

The proof is similar to the proof of Theorem 5 of [11]. Let R be the rectangle assigned to T. Let $D(n)$ be some function such that the area of R should be at least $D(n)$ for the algorithm to succeed, i.e., to draw T entirely within R.

Let H be the height and W be the width of R. We have two cases: $W \leq H$ and $H < W$.

Consider the case when $W \leq H$ (the case $H < W$ is analogous).

In each subsubcase of Case 1 (see Figure 2) T is divided into at most five subtrees: T_A, T_C, T_β, T_1, and T_2. Similary, in each subsubcase of Case 2 (see Figures 3), T is divided into at most three subtrees: T_A, T_B and T_C. In each of these subsubcases, for Algorithm *DrawTree* to succeed, it is necessary that the area of the rectangle R_k that is assigned to each tree T_k (where T_k is T_A, T_B, T_C, T_β, T_1, or T_2), should be at least $D(n_k)$, where n_k is the number of nodes in T_k. Let H_k be the height and W_k be the width of R_k.

The algorithm assigns R_k to T_k, and draws T_k inside R_k as follows: Let $x = n_k/n$. Let $W' = W - 2$ and $H' = H - 4n^{1-\epsilon} - 4(\log_2 n + 3) - 2$.

1. If T_k has a very small size, i.e., if $n_k < W'$ then $H_k = \log_2 n_k + 3$ and $W_k = W'$. T_k is drawn in R_k using Algorithm u^*-HV-Draw. Since $H_k = \log_2 n_k + 3$ and $W_k = W' > n_k$, from Lemma 1, Algorithm u^*-HV-Draw will succeed in drawing T_k within R_k.
2. Otherwise, if T_k has a moderate size, i.e., if $xH'/W' < n_k^{-\epsilon}$, then $W_k = W'$ and $H_k = W'/n_k^\epsilon$. In this case, the area of R_k is equal to $W' \cdot W'/n_k^\epsilon > W'xH' = xW'H'$ (since $xH'/W' < n_k^{-\epsilon}$). T_k is drawn recursively in R_k. Thus, Algorithm *DrawTree* can successfully draw T_k within R_k, if $D(n_k) \leq xW'H'$.
3. Otherwise, if T_k has a large enough size, i.e., if $xH'/W' \geq n^{-\epsilon}$, $W_k = W - 2$ and $H_k = xH'$. In this case, area of R_k is equal to $xH'W'$. T_k is drawn recursively in R_k. Thus, for Algorithm *DrawTree* to successfully draw T_k within R_k, it is necessary that $D(n_k) \leq xW'H'$.

Thus, we get that, for Algorithm *DrawTree* to successfully draw T_k within R_k, it is necessary that $D(n_k) \leq xH'W'$, i.e., $D(xn) \leq x(H - 4n^{1-\epsilon} - 4(\log_2 n + 3) - 2)(W - 2)$. (Since $n_k = xn$, $W' = W - 2$ and $H' = H - 4n^{1-\epsilon} - 4(\log_2 n + 3) - 2$.)

The condition that $D(xn) \leq x(H - 4n^{1-\epsilon} - 4(\log_2 n + 3) - 2)(W - 2)$ is also a sufficient condition for Algorithm *DrawTree* to draw T within R. To see this, notice that in both Case 1 and Case 2, as shown in Figures 2, 4, 5, and 3, Algorithm *DrawTree* uses at most 2 extra horizontal channels and 2 vertical channels to combine drawings of the partial trees to obtain a drawing of T. At least one partial tree of T, namely the subtree rooted at v, where (u, v) is the separator of T, has a large enough size, since it contains at least $n/3$ nodes. Thus, T can have at most 4 very small and moderately sized partial trees. Each moderately sized tree T_k is assigned a rectangle R_k with width $W - 2$ and height $W'/n_k^\epsilon = (W'/n_k)n_k^{1-\epsilon} \leq (W'/n_k)n^{1-\epsilon} < n^{1-\epsilon}$ (since $n_k \leq n$ and $W' < n_k$). Each very-small partial tree is assigned a rectangle with width $W - 2$ and height $\log_2 n + 3$. Hence, if each partial tree T_k can be drawn within its assigned rectangle R_k, then drawing all the very small and moderately sized partial trees will totally require at most $(4n^{1-\epsilon} + 4(\log_2 n + 3))(W - 2)$ area. It then follows that if each large enough partial tree with xn nodes can be drawn within area $x(H - 4n^{1-\epsilon} - 4(\log_2 n + 3) - 2)(W - 2)$, i.e., within its assigned rectangle, then the entire drawing of T can fit within R. Each very-small partial tree can be drawn within its assigned rectangle using Algorithm *u^*-HV-Draw*. If $D(xn) \leq x(H - 4n^{1-\epsilon} - 4(\log_2 n + 3) - 2)(W - 2)$ then each moderately sized and large enough tree can also be drawn within its assigned rectangle. We therefore conclude the if $D(xn) \leq x(H - 4n^{1-\epsilon} - 4(\log_2 n + 3) - 2)(W - 2)$, then T can be drawn within R.

Thus, $D(xn) \leq x(H - 4n^{1-\epsilon} - 4(\log_2 n + 3) - 2)(W - 2)$ is a necessary and sufficient condition for drawing T within R.

Since we are interested in finding $D(n)$, suppose the rectangle R assigned to T has area equal to $D(n)$. Then, because W is the smaller dimension of R, $W \leq \sqrt{D(n)}$, and since $A = H/W \leq n^\epsilon$, $H \leq n^{\epsilon/2}\sqrt{D(n)}$. Substituting W and H by these bounds in $D(xn) \leq x(H - 4n^{1-\epsilon} - 4(\log_2 n + 3) - 2)(W - 2)$, and with some mathematical manipulation, we get that $D(xn) \leq x(D(n) - (4n^{1-\epsilon} + 4\log_2 n + 2n^{\epsilon/2} + 14)\sqrt{D(n)} + 28) \leq x(D(n) - (4n^{1-\epsilon} + 4\log_2 n + 2n^{\epsilon/2} + 42)\sqrt{D(n)})$. Since $1/2 < \epsilon < 1$, $1 - \epsilon$ and $\epsilon/2$ are both less than $1/2$. Hence, $g(n) = 4n^{1-\epsilon} + 4\log_2 n + 2n^{\epsilon/2} + 42 \leq cn^\beta$, for some constants $c > 0$ and β, where $0 \leq \beta < 1/2$. Hence, from Theorem 1, we get that $D(n) = O(n)$.

□

Theorem 3 (Main Theorem) *Let T be a binary tree with n nodes. Given any number A, where $1 \leq A \leq n^\alpha$, for some constant α ($0 \leq \alpha < 1$), we can construct in $O(n \log n)$ time a planar straight-line grid drawing Γ of T with area $O(n)$ and aspect ratio A.*

Proof: Let ϵ be a constant such that $1 \leq A \leq n^\epsilon$ and $1/2 < \epsilon < 1$. Assign a rectangle R with area $D(n)$ and aspect ratio A to T, where $D(n)$ is the function as defined in proof of Lemma 4. Note that from the proof of Lemma 4, $D(n) = O(n)$. Construct a drawing Γ of T in R by calling Algorithm *DrawTree* with T, R and ϵ as input. From Lemmas 2, 3, and 4, Γ will be a planar straight-line grid drawing of T contained entirely within R. □

Remark. We believe that our algorithm can be extended to draw bounded-degree trees in $O(n)$ area. We are currently in the process of formalizing the algorithm to draw bounded-degree trees.

References

[1] T. Chan, M. Goodrich, S. Rao Kosaraju, and R. Tamassia. Optimizing area and aspect ratio in straight-line orthogonal tree drawings. *Comput. Geom. Theory Appl.* to appear. A preliminary version appeared in Proceedings of Graph Drawing '96, Lecture Notes in Computer Science, volume 1190, pages 63-75, 1997.

[2] P. Crescenzi, G. Di Battista, and A. Piperno. A note on optimal area algorithms for upward drawings of binary trees. *Comput. Geom. Theory Appl.*, 2:187–200, 1992.

[3] P. Crescenzi, P. Penna, and A. Piperno. Linear-area upward drawings of AVL trees. *Comput. Geom. Theory Appl.*, 9:25–42, 1998. (special issue on Graph Drawing, edited by G. Di Battista and R. Tamassia).

[4] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing.* Prentice Hall, Upper Saddle River, NJ, 1999.

[5] A. Garg, M. T. Goodrich, and R. Tamassia. Planar upward tree drawings with optimal area. *Internat. J. Comput. Geom. Appl.*, 6:333–356, 1996.

[6] C. E. Leiserson. Area-efficient graph layouts (for VLSI). In *Proc. 21st Annu. IEEE Sympos. Found. Comput. Sci.*, pages 270–281, 1980.

[7] E. Reingold and J. Tilford. Tidier drawing of trees. *IEEE Trans. Softw. Eng.*, SE-7(2):223–228, 1981.

[8] M. Sarkar and M. H. Brown. Graphical fisheye views. *Commun. ACM*, 37(12):73–84, 1994.

[9] C.-S. Shin, S.K. Kim, S.-H. Kim, and K.-Y. Chwa. Area-efficient algorithms for straight-line tree drawings. *Comput. Geom. Theory Appl.*, 15:175–2002, 2000.

[10] L. Trevisan. A note on minimum-area upward drawing of complete and Fibonacci trees. *Inform. Process. Lett.*, 57(5):231–236, 1996.

[11] L. Valiant. Universality considerations in VLSI circuits. *IEEE Trans. Comput.*, C-30(2):135–140, 1981.

Appendix

4.2 Algorithm u^*-*HV-Draw*

The classical HV-drawing algorithm [2] can be used to construct a drawing of a binary tree T with n nodes in a rectangle with dimensions n and $\log_2 n + 1$. Moreover, users can specify whether the longer dimension is the height of the width of a rectangle. This is formally stated in the following lemma:

Lemma 5 ([2]) *Let T be a binary tree with n nodes. Let R be a rectangle with width (height) n and height (width) $\log_2 n + 1$. Then, we can construct a drawing of T within R in $O(n)$ time, such that the root of T is placed at the top-left corner of R.*

However, it is possible that HV-drawing algorithm may not construct a feasible drawing of T, if T has a link vertex u^*, because it may not place u^* at the bottom of the drawing. Here, we give Algorithm u^*-*HV-Draw*, which will construct a feasible drawing of T, if T has a link vertex u^*, within a rectangle R with width (height) n and height (width) $\log_2 n + 3$): Let W and H be the width and height, respectively, of R.

1. Order the children of each node of T such that u^* becomes the leftmost node of T. As Described in Section 4.1, in the description of Algorithm *DrawTree*, this can be easily done in $O(n)$ time.
2. Let r be the root of T. Let $p = r_1(= r)r_2 \ldots r_k(= u^*)$ be the leftmost path of T (see Figure 8). Let $c_1(= c), c_2, \ldots, c_k$ be the right children of $r_1, r_2, \ldots, r_k$. (If a vertex r_i does not have a right child, then $c_i = \emptyset$). Let $T_1, T_2, \ldots, T_k$ be the subtrees rooted at $c_1, c_2, \ldots, c_k$. (If a vertex r_i does not have a right child, then $T_i = \emptyset$.)
3. If $W \geq H$, then draw each T_i using the HV-Drawing algorithm of [2] within a rectangle of width $n(T_i)$ and height $\log_2 n(T_i) + 3$. Otherwise (i.e., if $H > W$), draw T_i using the HV-Drawing algorithm of [2] within a rectangle of width $\log_2 n(T_i) + 3$ and height $n(T_i)$. Let Γ_i be the drawing constructed of T_i. Note that by Lemma 5, c_i will be at the top-left corner of Γ_i.
4. We have three cases, based on the value of k:
   (a) $k = 1$: As shown in Figure 11(b,c), place r one unit to the left and one unit above Γ_i. Draw the edge (r, c).
   (b) $k = 2$: If $W \geq H$, then flip Γ_k vertically, and place Γ_1 to the left of Γ_k, separated by unit horizontal distance such that their bottom boundaries are aligned (see Figure 10(b)); otherwise (i.e., if $H > W$), place Γ_1 to above Γ_k, separated by unit vertical distance such that their left boundaries are aligned (see Figure 10(c)). Place r one unit to the left and one unit above Γ_1, and u^* one unit to the left and one unit below Γ_k. Draw the edges (r, c), (r_u^*), and (u^*, c_k).
   (c) $k \geq 3$: If $W \geq H$, then first flip Γ_k vertically, and then, as shown in Figure 9(a), place $\Gamma_1, \Gamma_2, \ldots, \Gamma_k$ in a left-to-right order such that their top boundaries are aligned, $\Gamma_1, \Gamma_2, \ldots, \Gamma_{k-1}$ are separated by unit horizontal distance, and Γ_{k-1} and Γ_k are separated by two units of horizontal distance. Place $r, r_2, \ldots, r_{k-1}$ at one unit above these drawings at unit vertical distance, such that $r, r_2, \ldots, r_{k-2}$ are aligned with the left boundaries of $\Gamma_1, \Gamma_2, \ldots, \Gamma_{k-2}$, respectively, and r_{k-1} is aligned with the right boundary of Γ_{k-1}. Place u^* between Γ_{k-1} and Γ_k such that it is separated by unit horizontal distance from both, and is placed at unit vertical distance below the bottom boundary of Γ_j, where Γ_j is the subtree among $\Gamma_1, \Gamma_2, \ldots, \Gamma_k$ with greatest height. R. If $H > W$, then as shown in Figure 9(b), stack the drawings of $\Gamma_1, \Gamma_2, \ldots, \Gamma_k$ one above the other such that their left boundaries are aligned. Place each r_i, where $1 \leq i \leq k - 1$ at the same height as c_i at unit horizontal distance from it towards its left. Place $u*$ to the left of c_k at unit horizontal distance and at unit vertical distance below the bottom boundary of Γ_k.

Lemma 6 *Let T be a n-node binary tree with a link vertex u^*. Algorithm* u^*-HV-Draw *can be used to construct in $O(n)$ time, a feasible drawing of T within a rectangle with width n and $\log_2 n + 3$, as well as within a rectangle with width $\log_2 n + 3$ and height n.*

Proof: Immediate from Lemma 5, and the way Algorithm u^*-*HV-Draw* constructs its drawings. □

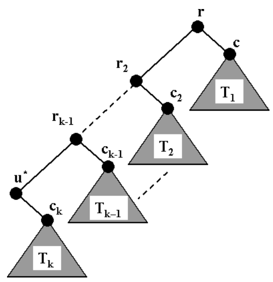
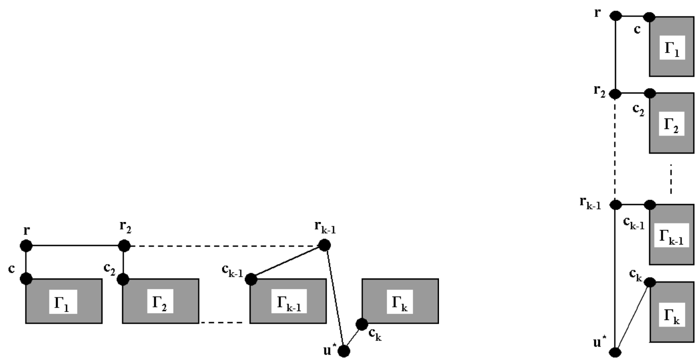


Figure 8: General form of a binary tree T with link vertex u^* and root r, where u^* is also the leftmost vertex of T.



(a) (b)

Figure 9: Drawing T of the Figure 8 when $k \geq 3$: (a) When $W \geq H$ (b) when $H > W$.

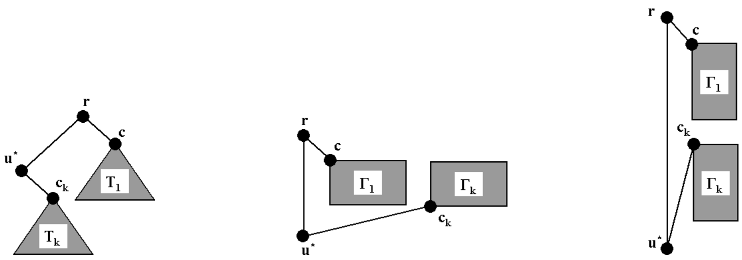
4.3 Algorithm *Comp*

We now give the pseudocode for Algorithm *Comp*. We use the same notation as in Section 4.1. Let

$$x = \begin{cases} f & \text{if } T_\beta \neq \emptyset \\ y & \text{otherwise} \end{cases}$$

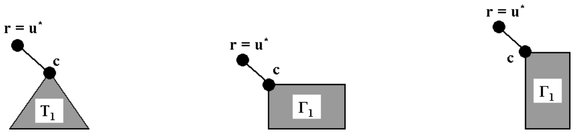
Let D and D' be two feasible drawings of some trees H and H'. The operation $D \ominus_1 D'$ places D to the left of D' separated from it by unit horizontal distance. The operation $D \ominus_2 D'$ places D to the left of D' separated from it by two units of horizontal distance. The operation $D \oplus_1 D'$ places D above D' separated from it by unit vertical distance. The operation $D \oplus_2 D'$ places D above D' separated from it by two units of vertical distance.

If the separator edge (u, v) does not belong to the leftmost path of T then algorithm Comp constructs Γ as illustrated in Figure 6, otherwise Comp constructs Γ as illustrated in Figure 7.



(a) (b) (c)

Figure 10: Drawing T of the Figure 8 when $k = 2$: (a) Tree T in this case; (b) Drawing T when $W \geq H$; (c) Drawing T when $H > W$.



(a) (b) (c)

Figure 11: Drawing T of the Figure 8 when $k = 1$: (a) Tree T in this case; (b) Drawing T when $W \geq H$; (c) Drawing T when $H > W$.

Algorithm Comp

Input: Γ_A, Γ_B, Γ_C, R.
Output: A feasible drawing Γ of T.

Let W and H be the width and height, respectively, of R;
Let (i, j) be the coordinates of the top-left corner of R;
if (u, v) does not belong to the leftmost path of T (see Figure 6) **then** $\{$
  if $T_A \neq \emptyset$ and $T_C \neq \emptyset$ (see Figure 6(a),(b)) **then** $\{$
    if $d = u^*$ **then** Flip Γ_C vertically;
    if $W \leq H$ **then** Let $\Gamma' = \Gamma_A \oplus_2 (\Gamma_B \oplus_1 \Gamma_C)$ such that the left boundaries of Γ_B and Γ_C are aligned and are placed one unit to the right of the left boundary of Γ_A;
    else ($H < W$) $\{$
      Let $\Gamma' = \Gamma_A \ominus_2 (\Gamma_C \ominus_1 \Gamma_B)$ such that the top boundaries of Γ_B and Γ_C are aligned and are placed one unit below the top boundary of Γ_A;
      Move Γ_C down such that u^* becomes the bottommost vertex of Γ';
    $\}$
    Place Γ' such that r is at grid location (i, j);
    if $W \leq H$ **then** Place a at grid location $(i, j - (height(R(T_A)) + 1))$;
    else ($H < W$) Place a at grid location $(i + width(R(T_A)) + 1, j)$;
    Connect a with e, x, and d;
  $\}$
  else if $T_A \neq \emptyset$, $T_C \neq \emptyset$ and $r \neq e$ (see Figure 6(c)) **then** $\{$
    if $W \leq H$ **then** $\{$
      Rotate Γ_A by $90°$;
      Then Flip Γ_A vertically;
      Flip Γ_B horizontally;
      Then Flip Γ_B vertically;
      Let $\Gamma' = \Gamma_A \oplus_1 \Gamma_B$ such that the left boundaries of Γ_A and Γ_B are aligned;
      stretch Γ_A to the right such that e becomes the rightmost vertex of Γ';
      Place Γ' such that r is at grid location (i, j);
      Place u^* at the bottom-right corner of $R(T)$;
    $\}$
    else ($H < W$) $\{$
      Rotate Γ_A by $90°$, and then Flip vertically;
      Flip Γ_B vertically;
      Let $\Gamma' = \Gamma_A \ominus_2 \Gamma_B$ such that the top boundaries of Γ_A and Γ_B are aligned;
      Place Γ' such that r is at grid location (i, j);

Place u^* at grid location $(i + width(R(T_A)) + 1, j - height(R))$;
}
Connect u^* with e and x;
}
else if $T_A \neq \emptyset$, $T_C = \emptyset$ and $r = e$ or $T_A = \emptyset$, $T_C \neq \emptyset$, and $d = u^*$ (see Figure 6(d)) **then** {
if $W \leq H$ **then** {
Flip Γ_B vertically;
Let $\Gamma' = \Gamma_A \oplus_1 \Gamma_B$ such that the left boundary of Γ_B is one unit to the right of the left boundary of Γ_A;
Place Γ' such that r is at grid location (i, j);
}
else ($J(R(T))$ is the width of $R(T)$) {
Flip Γ_B vertically;
Let $\Gamma' = \Gamma_A \ominus_1 \Gamma_B$ such that the bottom boundaries of Γ_A and Γ_B are aligned;
Place Γ' such that r is at grid location (i, j);
}
Place u^* at the bottom-left corner of $R(T)$;
Connect u^* with r and x;
}
else if $T_A = \emptyset$, $T_C \neq \emptyset$, and $d \neq u^*$ (see Figure 6(e)) **then** {
Place r at grid location (i, j);
if $W \leq H$ **then**
Let $\Gamma' = \Gamma_B \oplus_1 \Gamma_C$ such that the left boundaries of Γ_B and Γ_C are aligned;
Place Γ' such that x is at grid location $(i + 1, j - 1)$;
}
else ($H < W$)
Let $\Gamma' = \Gamma_C \ominus_1 \Gamma_B$ such that the top boundaries of Γ_B and Γ_C are aligned;
Place Γ' such that d is at grid location $(i + 1, j - 1)$;
Move Γ_C down such that u^* becomes the bottommost vertex of Γ';
}
Connect r with x and d;
}
else ($T_A = \emptyset$, $T_C = \emptyset$ (see Figure 6(f))) {
Place r at grid location (i, j);
Place Γ_B such that x is at grid location $(i + 1, j - 1)$;
Connect r with x;
}
}
else ((u, v) belongs to the leftmost path of T (see Figure 7)) {
if $T_A \neq \emptyset$, $T_B \neq \emptyset$ (see Figure 7)(a) **then** {
if $W \leq H$ **then** Let $\Gamma' = \Gamma_A \oplus_2 (\Gamma_B \oplus_1 \Gamma_C)$ such that the left boundaries of Γ_B and Γ_C are aligned and are placed one unit to the right of the left boundary of Γ_A;
else ($H < W$) {
Let $\Gamma' = \Gamma_A \ominus_2 (\Gamma_C \ominus_1 \Gamma_B)$ such that the top boundaries of Γ_B and Γ_C are aligned and are placed one unit below the top boundary of Γ_A;
Move Γ_C down such that u^* becomes the bottommost vertex of Γ';

}

Place Γ' such that r is at grid location (i, j);

if $W \leq H$ **then** Place u at grid location $(i, j - (height(R(T_A)) + 1))$;

else ($H < W$) Place u at grid location $(i + width(R(T_A)) + 1, j)$;

Connect u with e, c, and v;

}

else if $T_A = \emptyset$, $T_B \neq \emptyset$ (see Figure 7(b)) **then** {

Place r at grid location (i, j);

if $W \leq H$ **then** {

Let $\Gamma' = \Gamma_B \oplus_1 \Gamma_C$ such that the left boundaries of Γ_B and Γ_C are aligned;

Place Γ' such that c is at grid location $(i + 1, j - 1)$;

}

else ($H < W$) {

Let $\Gamma' = \Gamma_C \ominus_1 \Gamma_B$ such that the top boundaries of Γ_B and Γ_C are aligned;

Place Γ' such that v is at grid location $(i + 1, j - 1)$;

Move Γ_C down such that u^* becomes the bottommost vertex of Γ';

}

Connect r with c and v;

}

else if $T_A \neq \emptyset$, $T_B = \emptyset$ (see Figure 7(c)) **then** {

if $W \leq H$ **then** {

Let $\Gamma' = \Gamma_A \oplus_2 \Gamma_C$ such that the left boundary of Γ_C is one unit to the right of the left boundary of Γ_A;

Place Γ' such that r is at grid location (i, j);

Place u at grid location $(i, j - (height(R(T_A)) + 1))$;

}

else ($H < W$) {

Let $\Gamma' = \Gamma_A \ominus_2 \Gamma_C$ such that the top boundary of Γ_C is one unit below the top boundary of Γ_A;

Place Γ' such that r is at grid location (i, j);

Place u at grid location $(i + width(T_A) + 1, j)$;

Move Γ_C down such that u^* becomes the bottommost vertex of Γ';

}

Connect u with e and v;

}

else ($T_A = \emptyset$, $T_B = \emptyset$) (see Figure 7(d)) {

Place r at grid location (i, j);

Place Γ_C such that v is at grid location $(i + 1, j - 1)$;

Connect r with v;

}

}

Let Γ be the drawing obtained;

Return Γ;

end Algorithm.