Chapter 3

Tree Drawing

3.1 Rooted Trees

Rooted trees are at the center of many problems and applications in computer science. Information systems, multimedia documents databases, or virtual reality scene descriptions are only a few examples in which they are used. Their widespread use is most probably the result of the fact that they capture and reflect the way humans often organize information. A visual representation of these structures is often a major tool to help the user find his/her way in exploring data; hence the importance of graph drawing and exploration in information visualization.

3.1.1 Tree definition

We first briefly review some basic concepts about trees. A *tree* is a connected acyclic graph. Trees can be divided into rooted trees and free trees. A *rooted* tree T has a specific vertex $r \in T$ which is the *root* of the tree T. In reverse, *free trees* does not have any prespecified vertex for root.

Trees can also be divided into binary trees and multiway trees. ? *binary tree ?* of n nodes, $n > 0$, either is empty, if $n = 0$, or consists of a root node u and two binary trees $?_1$ and $?_2$ of n_1 and n_2 nodes, respectively, such that $n = 1 + n_1 + n_2$. We say that $?_1$ is the *left subtree* of $?$, and $?_2$ is the right subtree of $?$. ? *multiway tree ?* of n internal nodes, $n > 0$, either is empty, if $n = 0$, or consists of a root node u, an integer $d_u > 1$, which is the degree of u, and multiway trees $?_1, ..., T_{du}$ of $n_1, ..., n_{du}$ respectively, such that $n = 1 + n_1 + ... + n_{du}$.

If $u_1, ..., u_d$. are the roots of $?_1, ..., T_{dn}$. respectively, then we say that u is the *parent* of $u_1, ..., u_{dn}$, and $u_1, ..., u_{dn}$ are the *children* of u and the *siblings* of each other. Every node in a tree is at a specific *level* that can be defined by using the following node-numbering scheme. Number the root node 0, and number every other node to be one more than its parent; then the number of a node u is that node's level.

3.1.2 Layering

A tree is drawn to give us an intuitive understanding of the relationships appearing among the data during the solution of a problem. Tree drawings are common in books, articles, and reports. There are many different ways to draw a tree but they are not all equally appropriate. Several aesthetic rules have been proposed in an attempt to define a well-shaped drawing of a tree. The aesthetic rules 1 through 5 described in the following are presented by Wetherell and Shannon [WS79]; and rule 6 is presented by Tilford [RT81].

Aesthetic Rules

1 Trees impose a distance on the nodes; no node should be closer to the root than any of its ancestors.

2 Nodes at the same level of the tree should lie along a straight line, and the straight lines corresponding to the levels should be parallel.

3 The relative order of nodes on any level should be the same as in the level order traversal of the tree.
4 For a binary tree, a left child should be positioned to the left of its parent and a right child to the right.
5 A parent should be centred over its children.
6 A subtree of a given tree should be drawn the same way regardless of where it occurs in the tree.

The basic task in drawing a tree is to assign a pair of coordinates (x, y) to each node of the tree. Since we physically draw trees vertically, the y-coordinates of nodes are easy to determine from their levels. The most difficult task is to decide the x-coordinates of the nodes. An easy method to do this is to assign at each node a number proportional to its rank in the *inorder* traversal (Algorithm 3.1) of the tree, like the example tree in Figure 3.1.

The Visit(v) function of Algorithm 3.1 is equivalent to the numbering of node v. Starting from the root node, the first step of INORDER_TRAVERSAL(r) is to recursively call itself for the left child of node r (node 5). Again at node 5, it will be called for its left child (node 1). Node 1 has not a left child, so the first step of the algorithm fails and continues to step 2, which is the numbering of the node. Since it is the first node numbered, it gets value 1. Then it continues to step 3, which is to call itself for the right child of node 1, which exists (node 3), and so on.

Algorithm 3.1 *Inorder Traversal*
Input: The root node r of binary tree T
Output: An inorder numbering of the nodes of T

```
INORDER_TRAVERSAL(v)
    1    INORDER_TRAVERSAL(v ? LeftChild)
    2    Visit(v)
    3    INORDER_TRAVERSAL(v ? RightChild)
```

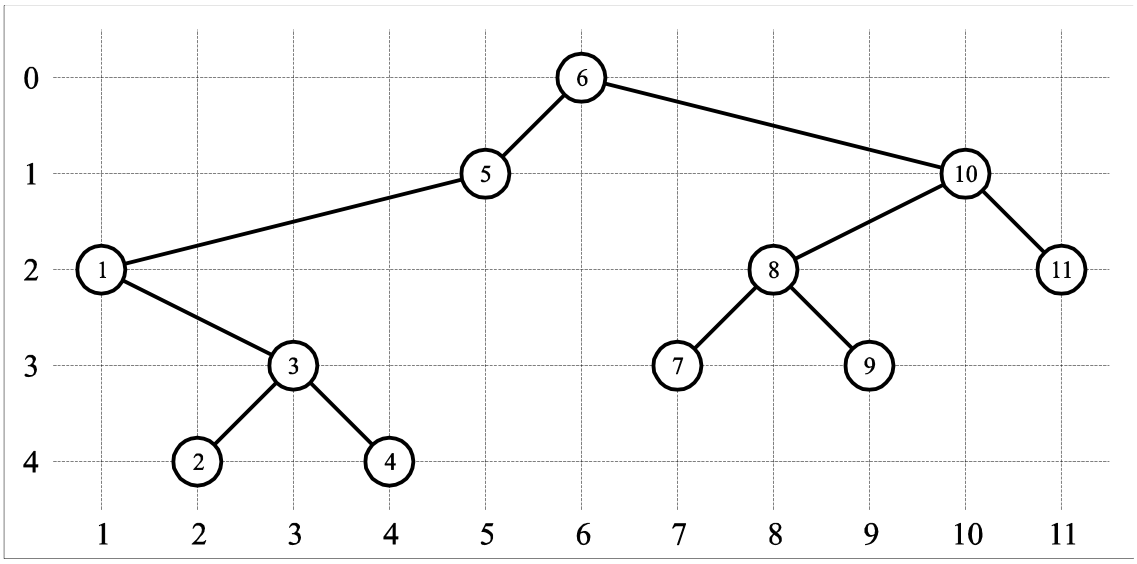

Figure 3.1: Layered drawing of a binary tree with x-coordinates assigned with an inorder traversal (each node is labelled with its inorder number).

While this simplistic approach satisfies basic aesthetic rules (Aesthetic rules 1 through 4 above), the tree drawings it generates are not well structured since they do not satisfy other aesthetic rules; a parent vertex is not necessarily centred over its children and the drawing is much wider than necessary.

Reingold and Tilford [RT81] presented a *divide and conquer* approach to determine the position of nodes. The algorithm of Reingold and Tilford (RT algorithm) takes a modular approach to the positioning of nodes. The relative positions of the nodes in a sub-tree are calculated independently of the rest of the tree. After the relative positions of two sub-trees have been calculated, they can be joined as siblings in a larger tree by placing them together as close as possible and centering the parent node above them. Imagine that the two sub-trees of a binary node have been drawn and cut out of paper along their contours. Then, starting with the two sub-trees superimposed at their roots, move them apart until a minimal agreed-upon distance between the trees is obtained at each level. This can be done gradually and can be described as shown in Figure 3.2. Initially, their roots are separated by some agreed-upon minimum distance; then, at the next level, they are pushed apart until the minimum separation is established. This process is continued at successively lower levels until the last level of the shorter sub-tree is reached. When the process is complete, the position of the sub-trees is fixed relative to their parent, which is centered over them.

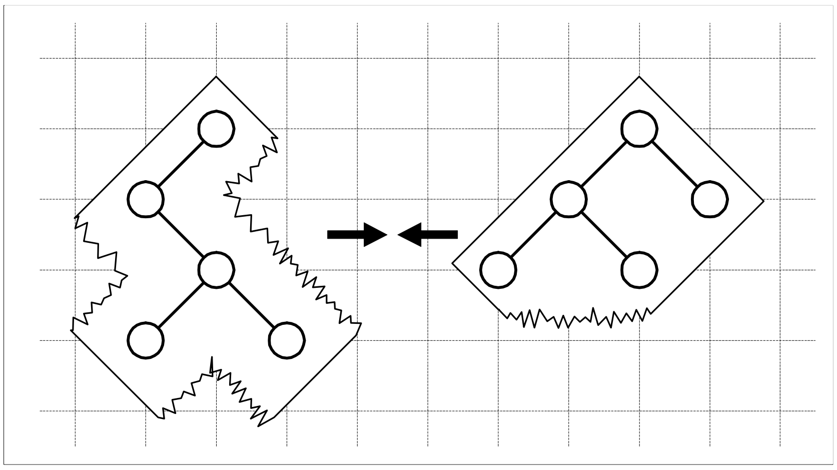

Figure 3.2: Conquer step of the algorithm.

Concisely the steps of the algorithm are presented below:

Algorithm 3.2 *Layered-Binary-Tree-Draw*
Input: A binary tree *T*
Output: A layered drawing of *T*

- *Base*
  If T has only one vertex, the drawing is trivial.
- *Divide*
  Recursively apply the algorithm to draw the left and right subtrees of tree T.
- *Conquer*
  Move the drawings of subtrees until their horizontal distance equals 2. At the end, place the root r of T vertically one level above and horizontally half way between its children. If there is only one child, place the root at horizontal distance 1 from the child.

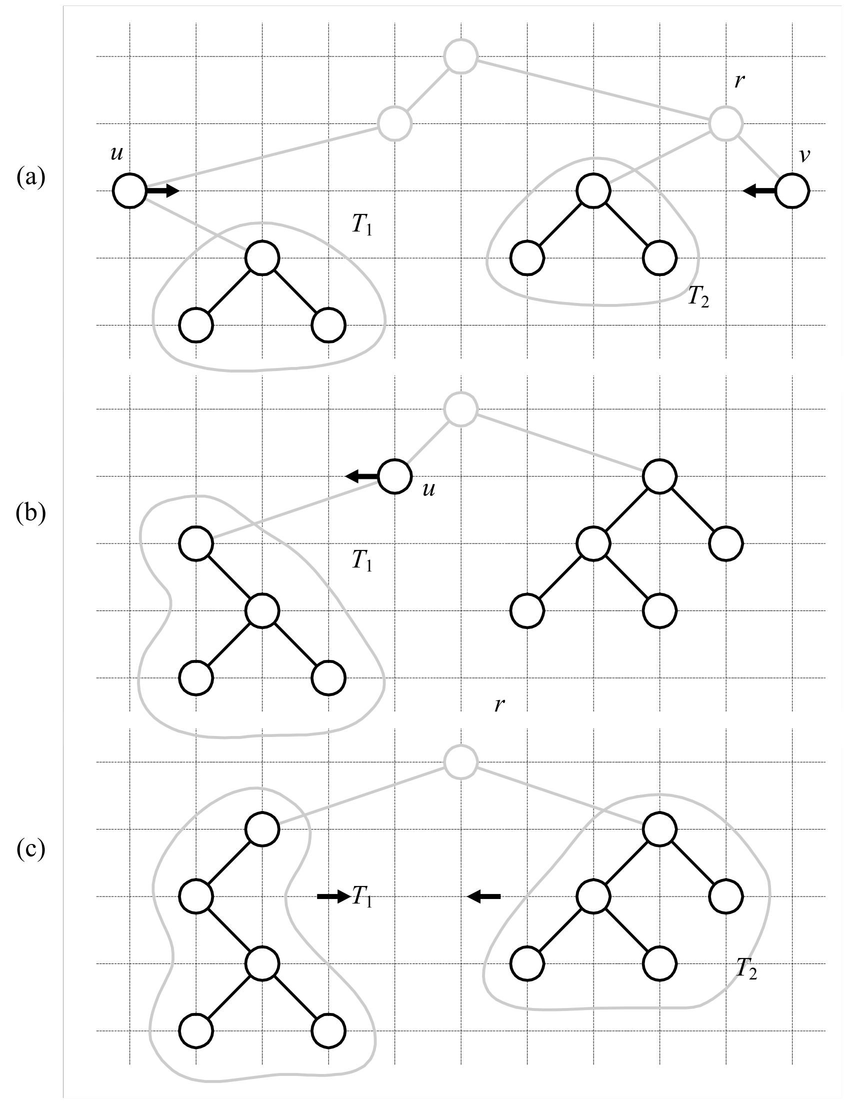

Figure 3.3: Various steps of the RT algorithm. (a) Node *u* is placed at distance 1 from subtree T_1 because it has only one child (the root of T_1). Node *v* is placed at distance 2 from subtree T_2 and the node *r* (parent of node *v* and the root of T_2) is placed halfway between its children. (b) Node *u* is placed at distance 1 from subtree T_1 because it has only one child (the root of T_1). (c) Subtrees T_1 and T_2 are placed at distance 2 and the parent is placed halfway between their roots (*r*'s children) resulting in the tree shown at Figure 3.4.

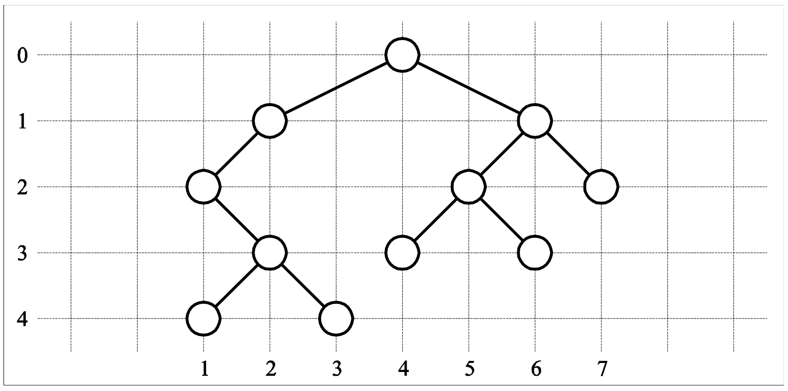

Figure 3.4: Drawing of the same binary tree after the RT algorithm. Note that the width of the tree is now 6 against 10 (in Figure 3.1)

Note that at any level two subtrees can never be moved closer; they can only be moved apart. Also note that once a subtree is laid out, its shape is fixed. The RT algorithm satisfies all six aesthetic rules presented above. Using the RT algorithm, the tree shown in Figure 3.1 is now redrawn as shown in Figure 3.4.

The above algorithm can be implemented in two traversals of the input binary tree which has an O(N) complexity, where N is the number of nodes of the tree to be drawn. The first traversal (postorder) sets the child nodes positions relative to their parent. For each vertex v, recursively computes the horizontal displacement of the left and right children of v with respect to v. The second traversal (preorder) fixes absolute positions by accumulating the displacements on the path from each vertex to the root for the x-coordinate, and by considering the depth of each vertex for the y-coordinate.

The crucial idea of the algorithm is to keep track of the *contour* of the sub-trees by special pointers, called threads, such that whenever two sub-trees are joined, only the top part of the trees down to the lowest level of the smaller tree need to be taken into account. The nodes are positioned on a fixed grid and are considered to have zero width.

In the postorder traversal part of the recursion is the merging of the contours of the two subtrees. The *left* contour of a binary tree T with height h is the sequence of vertices $v_0, ..., v_h$ such that v_i is the leftmost vertex of T with depth i. Similarly we can define the *right* subtree. The construction of the contour of the resulting tree can be done in the following way: Suppose that we have two subtrees T_1 and T_2 and the rooted vertex r. T_1 and T_2 are the left and right subtrees of r respectively. Every subtree has a unique left and right contour and the computation of the left and right contour of the resulting tree can be derived by the initial contours of the two subtrees. During the construction, we can have one of the following three cases:

1 If both subtrees have the same height h, then the left contour of the resulting tree will be the left contour of T_1 (left subtree) plus the rooted vertex r, and respectively the right contour will be the right contour of T_2 (right subtree) plus the vertex r.

2 If the height of the left subtree is less than the height of the right subtree, then the contour of the resulting tree will be derived as follow: The right contour of the resulting tree will be the right contour of the right subtree plus the rooted vertex r. The left contour can be the result of the concatenation of two portions plus the rooted vertex r. Let the height of the left contour of the left subtree be h, and its bottommost vertex be u. Also, let the w vertex belong to the left contour of the right subtree and its depth is h+1. Then, the left contour will consist of two portions: (plus the rooted vertex r) the left contour of the left subtree, and the portion of the left contour of the right subtree from the vertex w until its bottommost vertex. This case is illustrated in Figure 3.5.

3 The case in which the left subtree has greater height than the right subtree is analogous to the previous one.

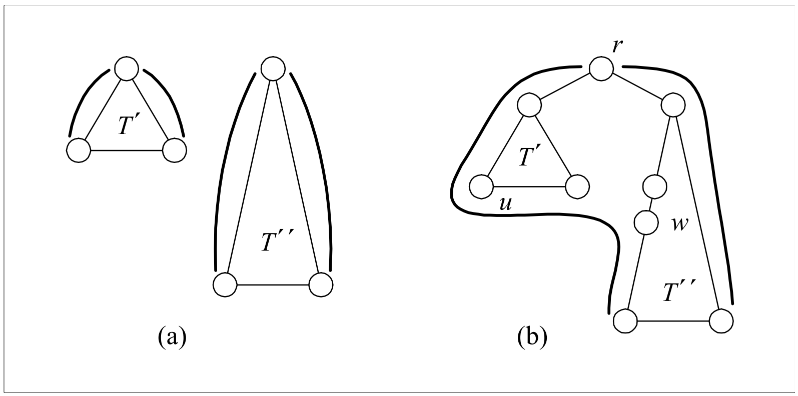

Figure 3.5: Construction of the contour lists. The left subtree T' is shorter than the right T'': (a) contour lists of T', T''; (b) contour lists of $T(r)$.

Incidentally, the modular approach taken by the RT algorithm is the reason that it fails to fulfil the need of tree drawings that occupy as little width as possible without violating the six aesthetic rules. As we can see in the Figure 3.6, the drawing of the tree constructed by the RT algorithm has width 14. But as shown in Figure 3.7, we can draw the same tree in a narrower manner (width 13). This drawing also fulfils all six aesthetic rules while occupying less space. The local horizontal compaction at each conquer step of the RT algorithm does not always compute a drawing of minimal width. This problem can be solved in polynomial time using linear programming, but it is NP-hard if there is a need for a grid drawing with integer values for the coordinates.

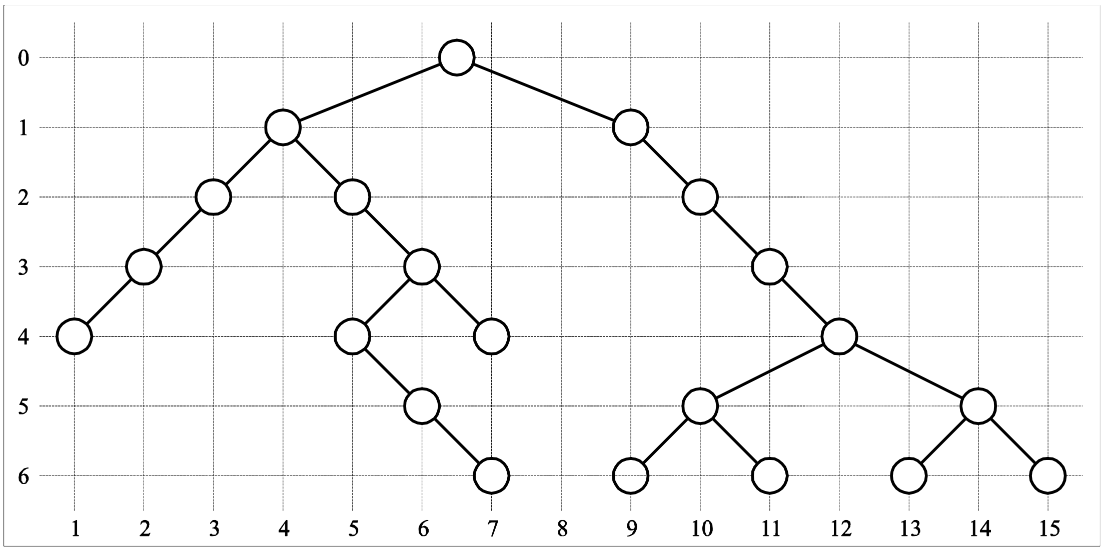

Figure 3.6: Example tree derived from the RT algorithm with non-optimal area occupation.

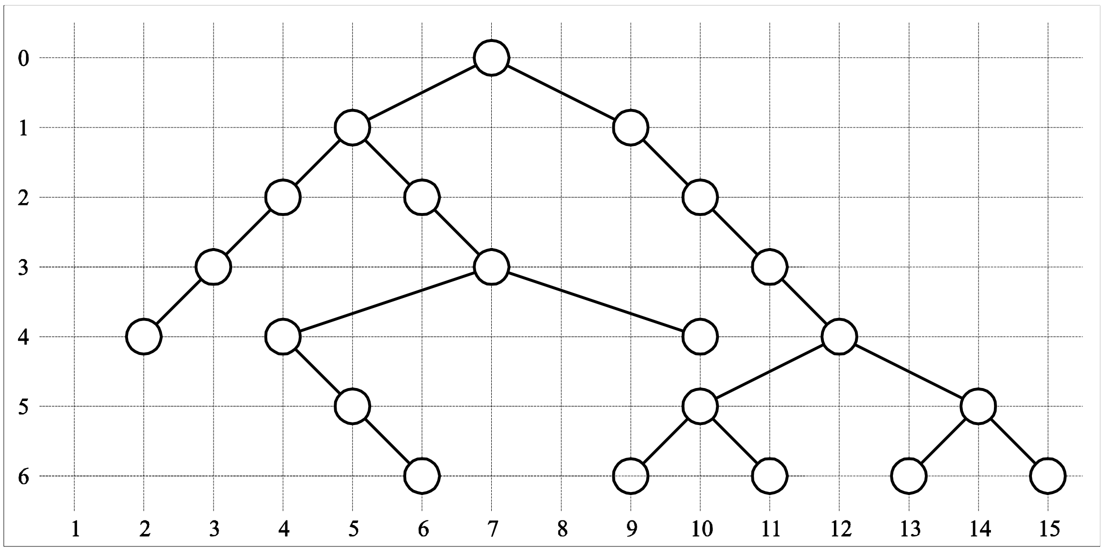

Figure 3.7: A narrower drawing of the same tree with Figure 3.6.

The properties for the Layered-Binary-Tree-Draw algorithm are summarized in the theorem below.

Theorem 3.1: Layered-Binary-Tree-Draw algorithm constructs a drawing of a binary tree T with n vertices in linear time such that is:

- Layered (the y-coordinate of each vertex is equal to minus its depth)
- Planar, straight-line and strictly downward.
- Occupies $O(n^2)$ area
- Two vertices are at horizontal and vertical distance at least 1
- Isomorphic subtrees have congruent drawing up to a translation
- Parent vertex is centered with respect to its children

Although the RT algorithm only draws binary trees, it can be straightforward extended to draw multiway trees (Algorithm 3.3). There is only a small imbalance problem with the x-coordinate of a parent vertex in case it has more than two children and we result in imbalanced layered drawings because the algorithm works from the left-to-right order for all the children. So, as we can see in Figure 3.8, the resulting drawing after we apply the algorithm in the particular rooted tree, is imbalanced.

Algorithm 3.3 *Layered-Tree-Draw*
Input: A tree T with subtrees $T_1, T_2, ..., T_m$
Output: A layered drawing of T

- *Base*
  If T has only one vertex, the drawing is trivial.
- *Divide*
  Recursively apply the algorithm to draw every subtree T_i.
- *Conquer*
  Move the drawings of subtrees T_i, T_{i-1} until their horizontal distance equals 2. At the end, place the root vertically one level above and horizontally half way between the roots of T_1 and T_m. If there is only one child, place the root at horizontal distance 1 from the child.

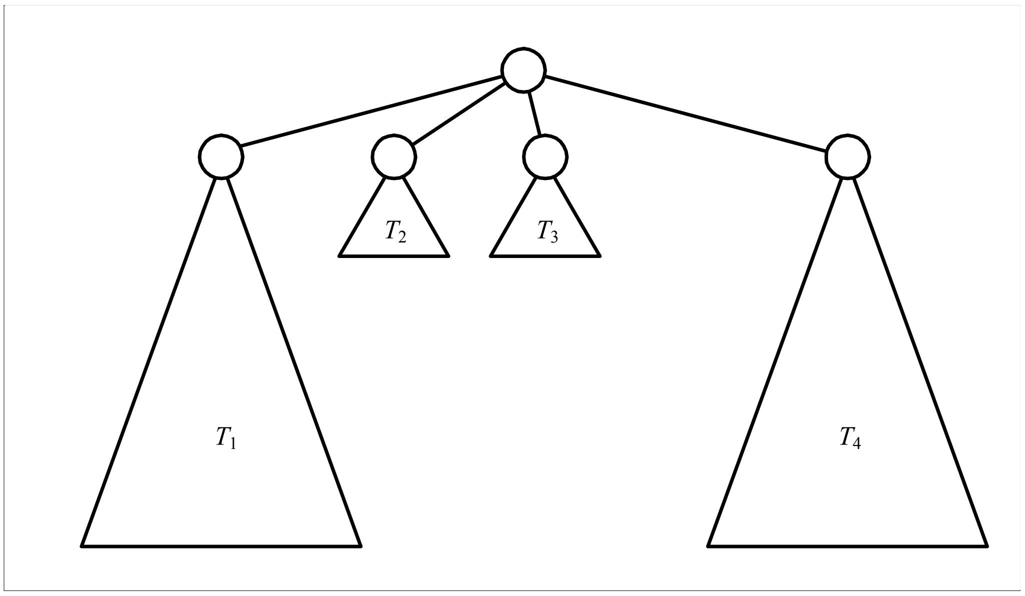

Figure 3.8: Imbalanced layered drawing of a tree.

The properties for the Layered-Tree-Draw algorithm are now extended and summarized in the theorem below.

Theorem 3.2: Layered-Tree-Draw algorithm constructs a drawing of a tree *T* with *n* vertices in linear time such that is:

- Layered (the *y*-coordinate of each vertex is equal to minus its depth)
- Planar, straight-line and strictly downward.
- Occupies $O(n^2)$ area
- Two vertices are at horizontal and vertical distance at least 1
- Isomorphic subtrees have congruent drawing up to a translation
- Axially isomorphic subtrees have congruent drawings, up to a translation and a reflection in *y*-axis

3.1.2 Radial Drawing

Radial drawing is an alternative way to draw rooted and free trees (trees with no specified root). In radial drawing, the root (or the node chosen to represent the root) of the tree is placed at the center, and all the descendant nodes on concentric rings around the root as shown in the example tree of Figure 3.9. Vertices of depth *i* are placed on circle C_i, an as the *i* increases, so does the radius *?*(*i*) of each circle C_i. Radial drawings would appear as if you were looking down onto a tree with the branches radiating from the center. An important consideration would be that the branches of the tree do not overlap.

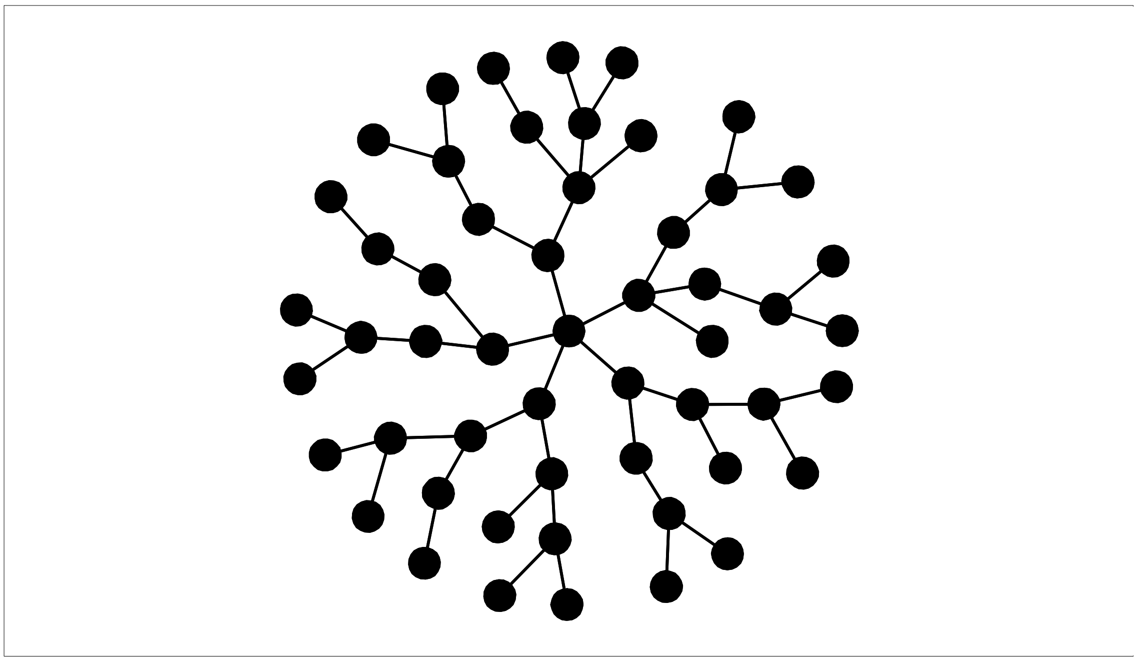

Figure 3.9: A radial drawn tree.

To ensure that the edges will not overlap, the subtree rooted at a vertex *v* is drawn bounded by an area called *annulus wedge*, because of its shape. An example of an annulus wedge is shown in Figure 3.10. If the angle of the wedge is greater than a certain limit, then edge crossing may occur because an edge with endpoints within the wedge can extend outside and intersect with other edges, as shown in Figure 3.11. To guarantee planarity, vertices must be restricted to a convex subset of the annulus wedge.

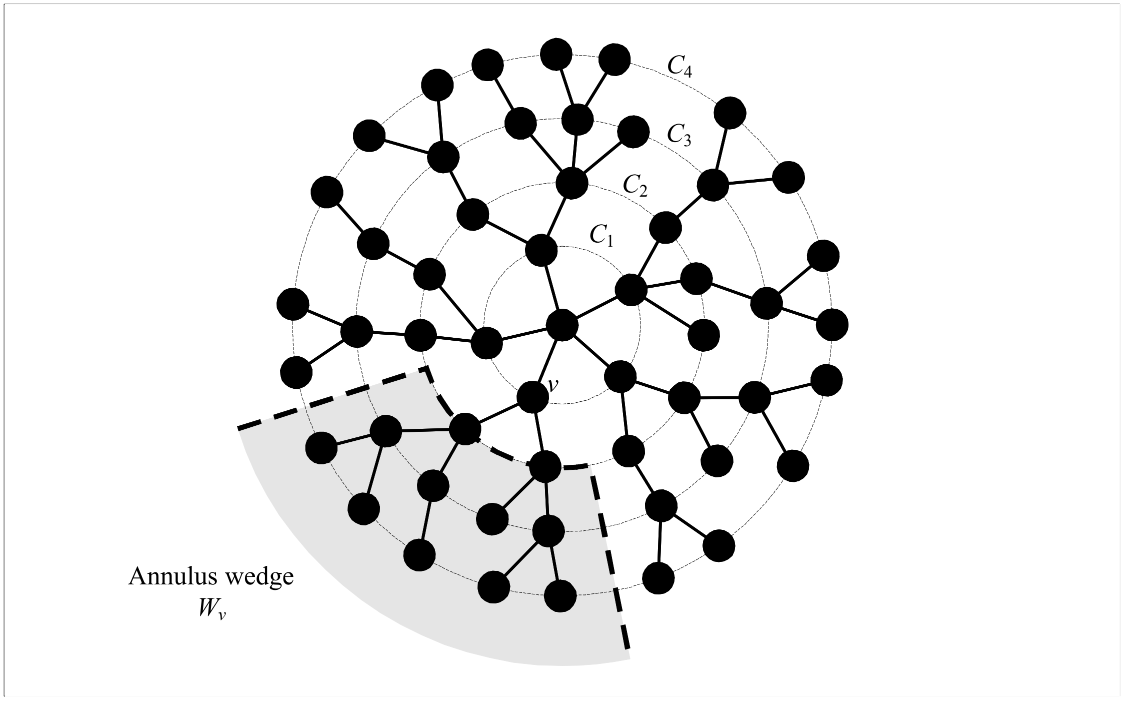

Figure 3.10: The annulus wedge of a subtree, and the concentric ring around the root of the same tree with Figure 3.8.

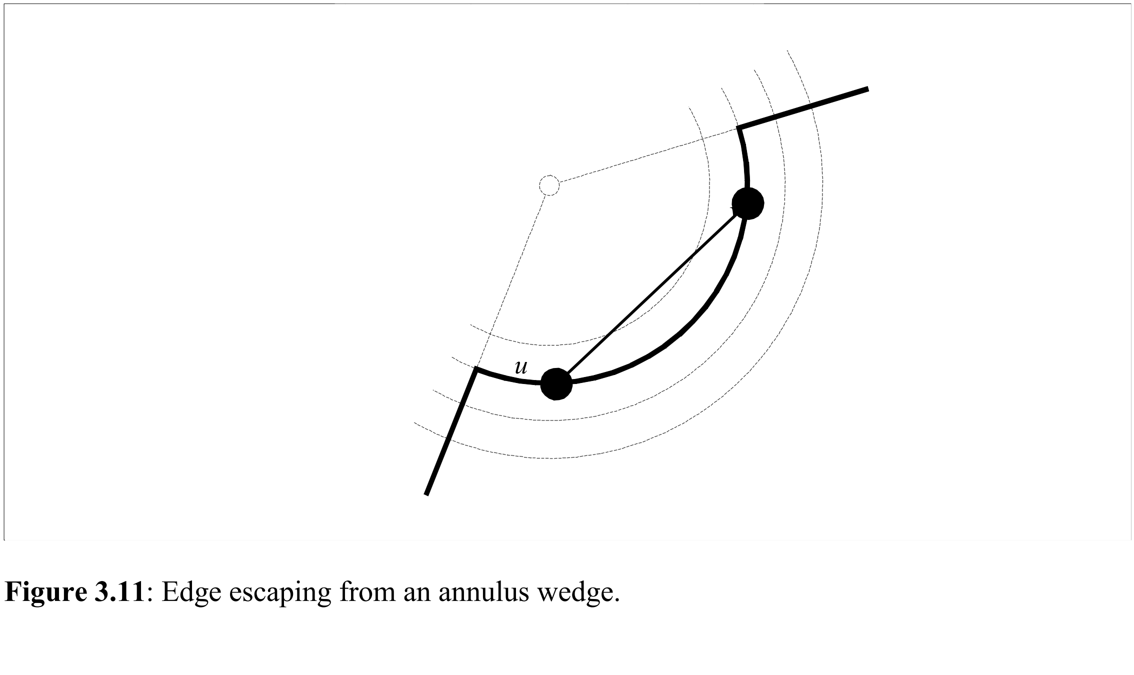

Figure 3.11: Edge escaping from an annulus wedge.

Suppose that we have a subtree rooted at vertex v which is drawn in annulus wedge W_v. Let $l(v)$ be the number of leaves in the subtree. As shown in Figure 3.12, v lies on C_i, and the tangent to C_i through v intersects C_{i+1} at points a and b. The unbounded segment F_v formed by the line segment ab and the rays from the origin through a and b is convex, and the descendants of *v* will be drawn inside this area. The children of *v* will be arranged on C_{i+1} according to the number of leaves in their respective subtrees. Specifically, the angle $ß_u$ of the wedge W_u of each child is

$$b_u = \min\left(\frac{\ell(u) b_v}{\ell(v)}, t \right)$$

where *t* is the angle formed by the region F_u. Note that $\cos(t/2) = \frac{r(i)}{r(i+1)}$, where *?*(*i*) is the radius of circle C_i.

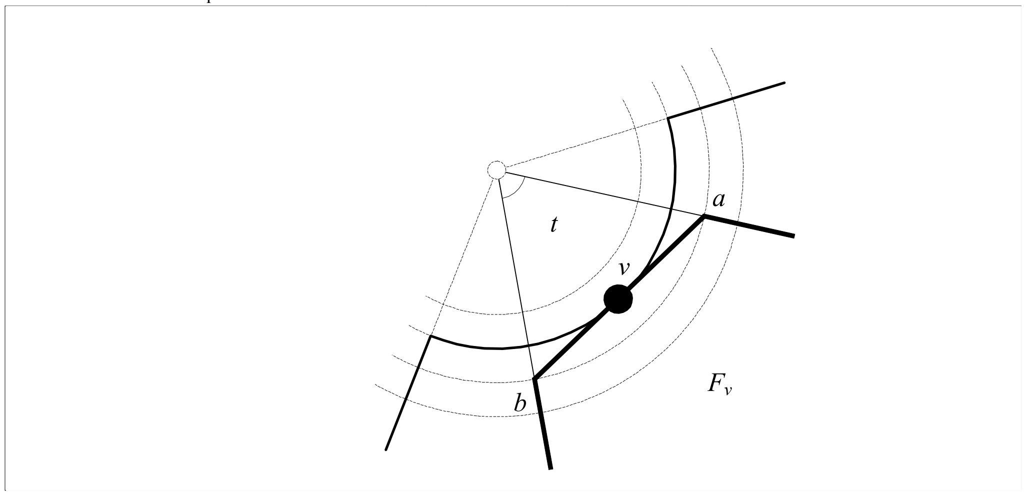

Figure 3.12: Convex subset of the annulus wedge.

For a free tree, the root is selected such that the height of the resulting rooted tree is the minimum possible. A simple pruning algorithm can be used to find in linear time the center of the tree:

Algorithm 3.4 *Tree-Pruning*

Input: A tree *T*

Output: The root of tree *T* such that the height of *T* is the minimum possible.

```
1. If the tree has at most two vertices, the center(s) have been found
2. Remove all the leaves, and goto 1
```

If the number of nodes is odd there is a unique center, else for even number of nodes, the center corresponds to the center of the line segment which joins the two nodes.

3.1.2 HV-Drawing

The drawing of a rooted binary tree using the *hv-drawing* convention, is a planar grid drawing in which tree nodes are represented as points (of integer coordinates) in the plane and tree edges as non-overlapping vertical or horizontal line segments. Moreover, each node is placed immediately to the right or immediately below its parent and the drawings of subtrees rooted at nodes with the same parent are non-overlapping. Figure 3.13 shows an example of an hv-drawing representation of a binary tree.

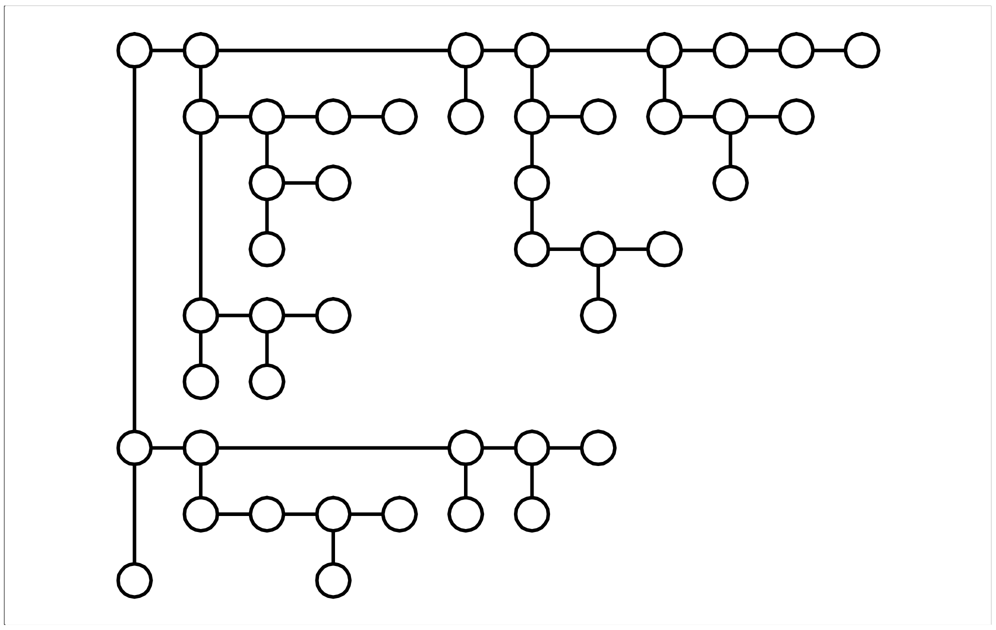

Figure 3.13: An hv-drawing of a binary tree.

Different hv-drawings of the same tree can be of different quality. The quality (or cost) is a function of the drawing. The most commonly used cost function is the area of the enclosing rectangle of the drawing. For a general binary tree, it is possible to construct an hv-drawing that is optimal with respect to one of several cost measures, including area and perimeter, in $O(n^2)$ time. We can compute an optimal hv-drawing of a tree with n nodes with respect to a cost function $w(x, y)$ which is non-decreasing in both parameters x, y where x and y are the width and the height of the enclosing rectangle of the drawing, respectively. Algorithm 3.4 is a general divide-and-conquer algorithm for constructing hv-drawings.

Algorithm 3.4 *HV-Tree-Draw*
Input: A rooted binary tree T
Output: An hv-drawing of T

- *Base*
  If T has only one vertex, the drawing is trivial.
- *Divide*
  Recursively construct hv-drawings for both left and right subtrees.
- *Conquer*
  Perform either a horizontal combination, as shown in Figure 3.14.a or a vertical combination, as shown in Figure 3.14.b.

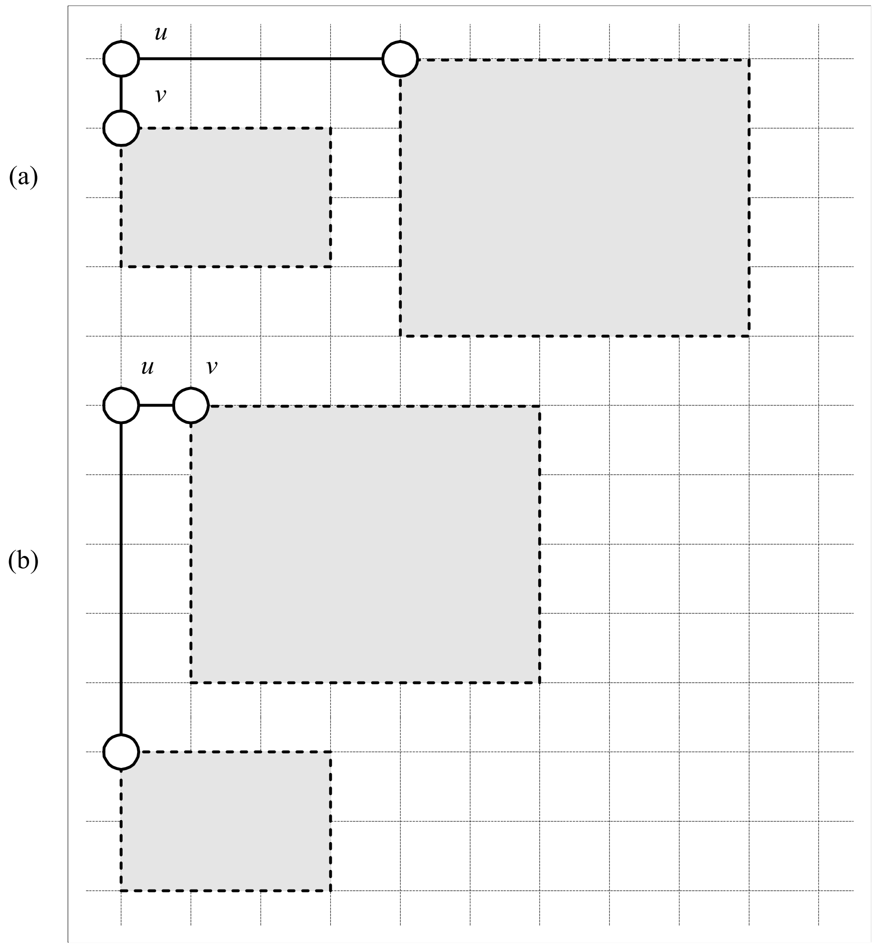

Figure 3.14: (a) horizontal combination: child node *v* is placed immediately below parent node *u*; (b) vertical combination: child node *v* is placed immediately to the right of parent node *u*. In either case, child nodes are placed in such way that the corresponding subtrees are non-overlapping.

At the conquer step of Algorithm 3.4, we have to options to draw the subtrees of a node *u* as shown in Figure 3.14. In *horizontal combination* a child of a node *u* is horizontally aligned with and to the right of *u*, while the other child is vertically aligned with and immediately below *u*, as shown in Figure 3.14.a. In *vertical combination* a child of *u* is vertically aligned with and below *u*, while the other child is horizontally aligned with and immediately to the right of *u*, as shown in Figure 3.14.b.

It is also easy to verify that if every subtree is placed in the left of every other subtree in the horizontal combination, then the width of the final hv-drawing will be at most *n*-1, where n is the number of all vertices of all the subtrees. The same can be noted for the vertical combination too.

During the construction of the hv-drawing we may choose to perform only horizontal combinations which will lead to a non-optimal area of the drawing. A better way is to use both horizontal and vertical combinations. We can choose horizontal combinations for subtrees rooted at vertices of odd depth, and vertical combinations for the others. This will lead to a balanced drawing which has area O(*n*) and aspect ratio O(1) (the shape of the occupying area tends to be square).

There is a simple specialization of the above algorithm which is called *Right-Heavy-HV-Tree-Draw* (Algorithm 3.5). In this approach, at the conquer step we perform only horizontal combinations and place the largest subtree to the right of the smallest subtree. Figure 3.15

shows an example of an hv-drawing of a binary tree, constructed by Algorithm *Right-Heavy-HV-Tree-Draw.* For a binary tree *T* with *n* vertices, the height of the drawing of *T* constructed by the *Right-Heavy-HV-Tree-Draw* is at most O (log*n*).

Algorithm 3.5 *Right-Heavy-HV-Tree-Draw*
Input: A binary tree *T*
Output: An hv-drawing of *T*

- *Base*
  If T has only one vertex, the drawing is trivial.
- *Divide*
  Recursively construct hv-drawings for both left and right subtrees.
- *Conquer*
  Perform a horizontal combination by placing the subtree with the largest number of vertices to the right of the other one.

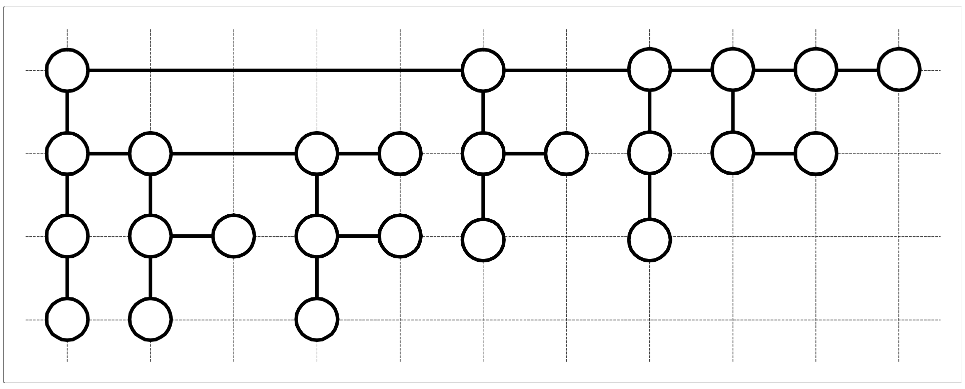

Figure 3.15: An example hv-drawing constructed by Algorithm 3.5 Right-Heavy-HV-Tree-Draw.

Theorem 3.3: Right-Heavy-HV-Tree-Draw algorithm constructs a drawing of a tree *T* with *n* vertices in linear time that is:

- Downward, planar, grid, straight-line, and orthogonal, in other words, an hv-drawing.
- Occupies O(*n*log*n*) area
- Its width is at most *n*-1
- Its height is at most log*n*
- Axially isomorphic subtrees have congruent drawings, up to a translation and a reflection in *y*-axis

In general, the biggest problem in constructing an hv-drawing for a tree, is how many times we will apply the horizontal or the vertical combination, corresponding in the resulting area of the hv-drawing. Because of the imbalance of the aspect ration of the trees constructed by the algorithm Right-Heavy-HV-Tree-Draw, a good approach is to use horizontal combination for subtrees rooted at vertices of odd depth, and vertical combinations for subtrees rooted at vertices of even depth. The resulting drawing of this method occupies an O(*n*) area.

Algorithm 3.5 Right-Heavy-HV-Tree-Draw can be easily extended from binary trees to general rooted trees as shown in Figure 3.16. In this case, slanted lines are allowed to connect vertices of different level.

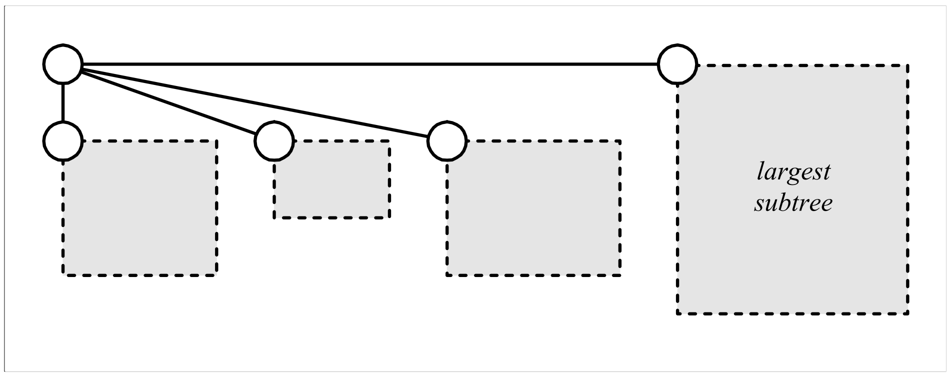

Figure 3.16: Extended version of Algorithm 3.5 Right-Heavy-HV-Tree-Draw to draw general rooted trees.

Theorem 3.4: There exists an algorithm which constructs a drawing of a tree *T* with *n* vertices in linear time that is:

- Downward, planar, grid and straight-line
- Occupies O(*n*log*n*) area
- Its width is at most *n*-1
- Its height is at most log*n*
- Axially isomorphic subtrees have congruent drawings, up to a translation and a reflection in *y*-axis