Description and Proof of Correctness of `treedrawing`

poypoyan

17 Feb 2022

Abstract

This note contains an overview of my Github repo `treedrawing` [1], which is a naive algorithm that neatly draws rooted trees. Then a proof of correctness of the algorithm is presented.

1 Definitions

We start by introducing some standard and non-standard terminologies:

Definition 1.1. *A **rooted tree** is a triple $T = (V, E, r)$ where (V, E) is a tree (a graph without cycles), and $r \in V$ is called the **root**.*

Remark. We adopt terminologies from computer science: we call V the set of **nodes** (instead of vertices), and E the set of **branches** (instead of edges).

Definition 1.2. *For nodes $x, y \in V$ connected by a branch, x is the **parent** of y and y is a **child** of x if x has closer path to the root than y.*

Definition 1.3. *A **leaf** is a node that does not have a child.*

Definition 1.4. *A non-root node is **minor** iff it is the only child of its parent and it has exactly one child node. A node is **major** iff it is the root or it is not minor.*

Definition 1.5. *The **sub-rooted tree** by a major node $x \in V$ is a rooted tree $T' = (V', E', x)$ consisting of x itself (as root) and all of its the 'descendant' nodes.*

Remark. (1) The sub-rooted tree by the root node is the whole rooted tree. (2) The sub-rooted tree by the leaf node is the trivial branch-less tree with itself as the only node.

Definition 1.6. *The **weight** of a major node is the number of major nodes of the sub-rooted tree by that node. The weight of a minor node is 0.*

Remark. The weight of every leaf node is 1.

We show the converse of the last sentence in Definition 1.6: all nodes of weight 0 are minor. This is equivalent to the statement that all major nodes have weights of at least 1. This is indeed the case because the major node itself is contained in the sub-rooted tree by itself. In conclusion, a node is minor iff its weight is 0.

To aid in understanding the concepts, an example rooted tree is provided in Figure 1.

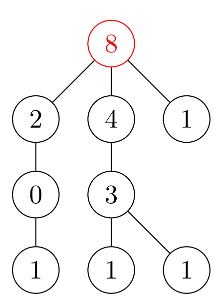

Figure 1: Example rooted tree with red-colored node as the root. The number in node indicates its weight.

2 Algorithm Description

The algorithm consists of three main functions that are executed in order: (1) `analyzeNode`, which records relevant information for each node recursively, (2) `setInitCoord`, which sets initial coordinates to each node recursively, and (3) `fixCoord`, which fixes coordinates of nodes so that there are no nodes with same location.

2.1 Function `analyzeNode`

The main input [1] of the algorithm is any data with a structure of rooted tree. Note that nodes in the input data must be *labelled*. What the `analyzeNode` function mainly does is to perform tree traversal on the data and record the following information:

- `connecNodes` is an array of arrays of natural numbers. An index `idx` in `connecNodes[idx]` corresponds to a node (with `idx = 0` being the root). Now `connecNodes[idx]` itself is the array of indices that correspond to the *children* of the `idx` node. If the `idx` node is a leaf, then `connecNodes[idx]` is the empty array.
- `labelDict` is a function that aligns node *labels* (strings) to indices for `connecNodes`, `weightNodes`, and `majorNodes`.
- `weightNodes` is an array of natural numbers. Like in `connecNodes`, an index `idx` in `weightNodes[idx]` corresponds to a node (with `idx = 0` being the root). Now `weightNodes[idx]` itself is the weight of the `idx` node.
- `majorNodes` is an array of objects. Like in `connecNodes`, an index `idx` in `majorNodes[idx]` corresponds to a node (with `idx = 0` being the root). Now for major nodes, `majorNodes[idx]` itself is the array of major nodes along the (unique) path from the root to the `idx` node (`idx` itself excluded). Since the root has no parent, `majorNodes[0]` is always the empty array. For minor nodes, `majorNodes[idx]` is the *null* object.

2.2 Function `setInitCoord`

As the name says, this function sets initial coordinates for each node. The root's coordinate is always set to $(0, 0)$; the major nodes are always set to coordinates of upright square lattice (see Figures 2 and 3). Aside from the above outputs of `analyzeNode` except `labelDict`, the relevant inputs are `outDir`, the direction of 'growth' of rooted tree, and `dist`, the distance between consecutive horizontal/vertical points in the upright square lattice. Without loss of generality, we set `outDir = 'D'` (downward direction) for the following discussion.

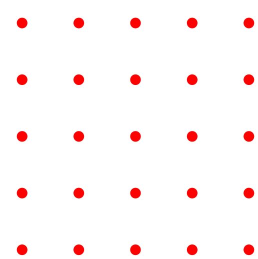

Figure 2: Upright square lattice. The middlemost point would be the origin.

For each major node x with one child node, minor nodes are traversed until encountering another major node y. The coordinate for y is set to one `dist` directly *below* (since `outDir = 'D'`) the coordinate of x. On the other hand, the coordinates of minor nodes between x and y are 'squeezed' between the coordinates of x and y at equal distance to each other. Now y either has no child or has multiple children. The former means unwinding the recursion. For the latter, y's children are all one `dist` below y. Now the x-coordinate of the children depends on their weights: the child with highest weight is placed directly below y, then the child with next highest weight is placed one `dist` away from the closest child. The placement of children swings (e.g. center, left, right, left, ...) and goes *away* from the child with highest weight. All of this is visualized in Figure 3.

The reasoning behind placement of children is aesthetics: the central path from root should have the most major nodes, and the sub-rooted trees that branch away from the central path should be minimal.

[1]Throughout this paper, we only mention the 'important' inputs. Other inputs are instead mentioned in `example.py` comments.

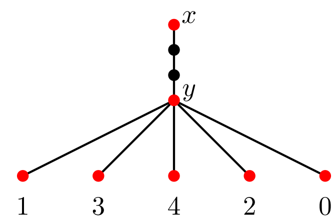

Figure 3: Initial coordinates; branches are drawn for clarity. Major nodes are red and minor nodes are black. Example weights of bottom nodes are also indicated.

2.3 Function `fixCoord`

The problem with the previous function is that several major nodes will have the same coordinates. This is what `fixCoord` fixes. The idea is to add coordinates of each node one-by-one in the set `subSet` until a coordinate to add is already in `subSet`, implying that a duplicate is found.

Let's say that nodes x and y ($x \neq y$) have the same coordinate. Using `majorNodes`, the two major nodes that have the same parent and splits the path from root to x and the path from root to y are determined. The indices of that major node for x and for y are stored at `splitNodeA` and `splitNodeB` (let's say respectively). If for example `weight[splitNodeA] > weight[splitNodeB]`, then the whole sub-rooted tree by `splitNodeB` must move *away* from the central path from root (and vice versa). This also means that the coordinate of y must move. How many `dist` to move? The whole sub-rooted tree by `splitNodeB` must move away until the coordinate of y is not equal to the coordinates of x and all siblings of x.

After the movement, `subSet` is cleared and the one-by-one adding of coordinates restarts. On the other hand, if no duplicate coordinate is found for all major nodes, then the function is done.

The main output of the algorithm is the final coordinates of each node. On the other hand, the drawing of branches is actually not an output! This means that the drawing of branches is a separate code to be made by programmer, although I have the simple 'straight line' branches (as seen in Figure 3) in mind.

3 Proof of Correctness

We prove that if the drawing of the branch is straight line, then no two branches *intersect*. Visually, this means that the intersection in Figure 4 should **never** occur in the output of the algorithm.

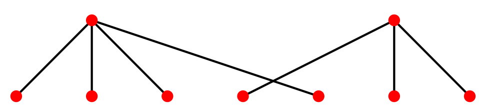

Figure 4: Intersecting branches.

Proof. To be continued. □

References

[1] poypoyan. treedrawing: naive rooted tree drawing algorithm. `https://github.com/poypoyan/treedrawing`.