An Indented Level-Based Tree Drawing Algorithm for Text Visualization

Xi He and Ying Zhu
Department of Computer Science
Georgia State University
Atlanta - 30303, USA
Email: xhe8@student.gsu.edu, yzhu@gsu.edu

***Abstract*—Level-based tree drawing is a common algorithm that produces intuitive and clear presentations of hierarchically structured information. However, new applications often introduces new aesthetic requirements that call for new tree drawing methods. In this paper, we propose an indented level-based tree drawing algorithm for visualizing parse trees of English language. This algorithm displays a tree with an aspect ratio that fits the aspect ratio of the newer computer displays, while presenting the words in a way that is easy to read. We discuss the design of the algorithm and its application in text visualization for linguistic analysis and language learning. An efficient and practical implementation of the algorithm is also presented.**

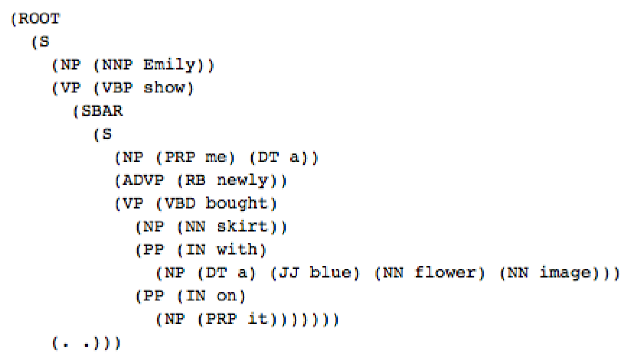
I. Introduction

Tree drawing is one of the most researched areas in data visualization. They are widely used in biology, business, chemistry, software engineering, artificial intelligence, Web site design, data analysis, education, and social networks.

In graph theory, a tree is an undirected graph without simple cycles. A typical tree consists of nodes and edges. Each node represents an entity. Each edge represents the relationship between entities. A tree with a root node is called a rooted tree.

A tree drawing algorithm consists of a set of rules for placing the nodes and drawing the edges. Some tree drawing rules are introduced to address the characteristics of the structure of the data. Some tree drawing rules are aesthetic rules for the efficient use of space and clarity of presentation. The most important aesthetics of tree drawings include area, aspect ratio, subtree separation, closest leaf, and farthest leaf, etc. A new application may introduce new aesthetic rules that lead to new tree drawing algorithms.

We are developing a text analysis and visualization program for linguistic studies and language learning. One of the main features is the visualization of the parse tree for each sentence. A parse tree is a rooted tree showing the syntactic structure of a sentence or a string (Figure 1). Visualizing the parse trees can help researchers or students analyze the structure of the sentence and its complexity. The typical drawing of a parse tree is a top-down, level-based tree (Fig. 2). This type of drawing is intuitive and clear. But the drawback is that the aspect ratio of the tree does not fit the aspect ratio of newer computer displays. The tree grows vertically. The height of the parse tree visualization is usually larger than its width, particularly when the sentence is structurally sophisticated. However, the standard aspect ratio of computer displays after 2012 is 16:9, with the width larger than the height. With a dual monitor setup, the display is even wider. Therefore a traditional parse tree visualization does not make optimal use of the screen space. When multiple parse trees need to be displayed in a sequence, this problem becomes even more obvious.



```
(ROOT
  (S
    (NP (NNP Emily))
    (VP (VBP show)
      (SBAR
        (S
          (NP (PRP me) (DT a))
          (ADVP (RB newly))
          (VP (VBD bought)
            (NP (NN skirt))
            (PP (IN with)
              (NP (DT a) (JJ blue) (NN flower) (NN image)))
            (PP (IN on)
              (NP (PRP it)))))))
    (. .)))
```

Fig. 1: The syntactic structure of the sentence “Emily show me a newly bought skirt with a blue flower image on it.”

Therefore, in our application and many similar cases, it is more desirable to display the parse tree horizontally with the root node on the left for the optimal use of space. In addition, the leaf nodes (i.e. the words) should be placed in such a way that they can still be read from left to right as a sentence. This becomes a new aesthetic rule for tree drawing. Simply drawing a tree horizontally with a level-based algorithm is not user friendly because readers have to read the sentence vertically (see Figure 3). A simple modification of the existing level-based tree algorithm to create an indented display of words also does not work because line crossings make the tree difficult to read (see Figure 4).

To address this issue, we propose a new indented level-based tree drawing algorithm that preserves the grammatical structure of the parse tree but also allows users to read the sentence from left to right. The resulting tree visualization fits the aspect ratio of most computer displays better than the traditional parse tree visualization. We also analyze the time complexity of this algorithm and discuss the implementation of this algorithm using JavaScript and d3.js.

The rest of the paper is organized as follows: Section II

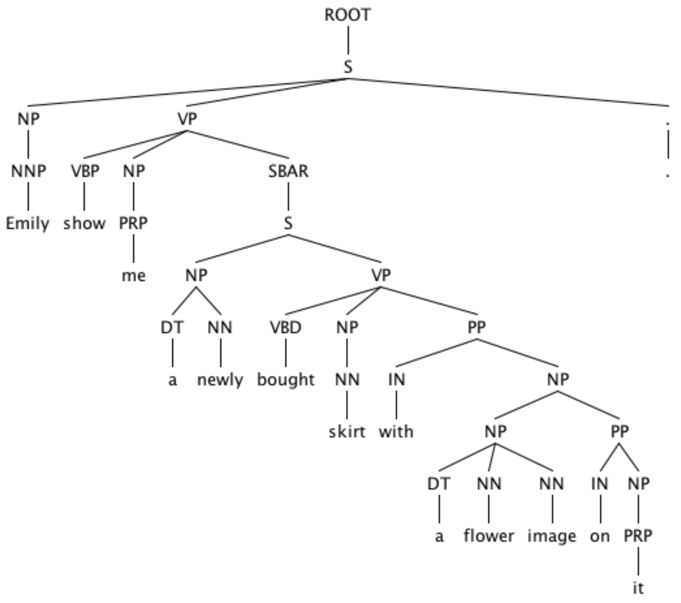


Fig. 2: The traditional level-based parsed tree of the sentence "Emily show me a newly bought skirt with a blue flower image on it."

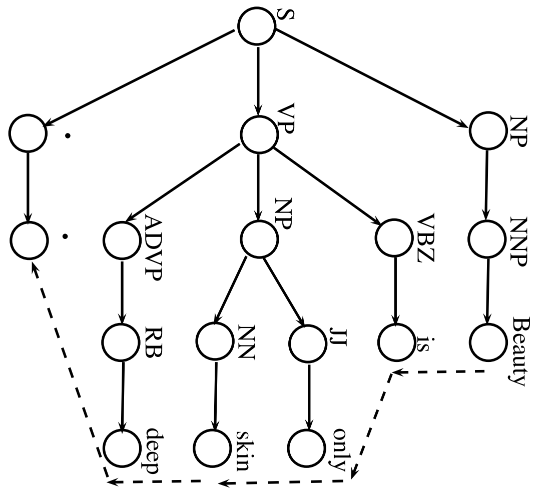


Fig. 3: The visualization of a parsing tree for the sentence "Beauty is only skin deep.". Note that the words are displayed vertically and the sentence is hard to read.

briefly reviews the algorithms for level-based tree drawing and non-layered tree drawing. Then we propose the new tree drawing algorithm and define the related tree drawing problem in Section III. In Section IV, we briefly review the Reingold and Tilford algorithm, which is the basis of our algorithm. We present the design details of our algorithm in Section V and analyze its time complexity in Section VI. We also discuss the implementation and usage of our algorithm in Section VII. Finally, we summarize the paper in Section VIII.

II. Related Work

A. Level-Based Tree Drawing Algorithms

The level-based tree drawing (also called layered tree drawing) is the most popular tree drawing algorithm. The five main aesthetic rules for level-based tree drawing include :

Aesthetic rule #1: Nodes of a tree at the same height should lie along a straight line, and the straight lines defining the levels should be parallel.

Aesthetic rule #2: A left child should be positioned to the left of its ancestor and a right child to the right.

Aesthetic rule #3: A father should be centered over its children.

Aesthetic rule #4: A tree and its mirror image should produce drawings that are reflection of one another; moreover, a subtree should be drawn the same way regardless of where it occurs in the tree.

Aesthetic rule #5: Small, interior subtrees should be spaced out evenly among larger subtrees. Small subtrees at the far left or far right should be adjacent to larger subtrees.

A tree drawing algorithm needs to calculate the position of each tree node in a way so that the resulting tree is aesthetically pleasing and conserves space.

Knuth first published an algorithm for drawing binary trees [1]. In 1979, Wetherell and Shannon [2] presented an $O(n)$ algorithm that satisfies aesthetic rules #1–3 while at the same time minimizes width. A similar algorithm was developed by Sweet [3]. In 1981, Reingold and Tilford [4] presented an algorithm that was inspired by the Wetherell and Shannon algorithm and addressed its flaws by satisfying aesthetic rule #4. Then in 1990, Walker [5] presented an improved method that satisfies aesthetic rule #5 for trees of unbounded degree. In 2002, Buchheim et al. [6] improved Walker's algorithm so that drawing trees of unbounded degree can be run in $O(n)$ instead of $O(n^2)$ in Walker's algorithm. For a recent survey on tree drawing algorithms, please refer to [7].

Our proposed algorithm falls into the category of level-based tree drawing. In this algorithm, we introduce a new aesthetic rule:

Aesthetic rule #6: A tree should be draw from left to right, with the leaf nodes indented in such a way that they can be read from left to right as a sentence.

B. Non-Layered Tree Drawing Algorithms

The level-based tree drawing algorithms assume that nodes in the tree have uniform size. However, in many practical applications, these tree nodes may have varying sizes. Non-layered tree drawing algorithms are therefore designed to generate a more vertically compact drawing which places child nodes at a fixed distance from the parent nodes. Miyadera et al. present an $O(n^2)$ algorithm [8] for non-layered trees that horizontally positions parent nodes at a fixed offset from their first child, instead of centering above the children. Many other algorithms, such as Bloesch algorithm [9], Stein and Benteler algorithm [10] and Li and Huang algorithm [11], Marriot et al. algorithm [12] and Ploeg algorithm [13], employ the similar idea. After preprocessing such as discretizing, they run the

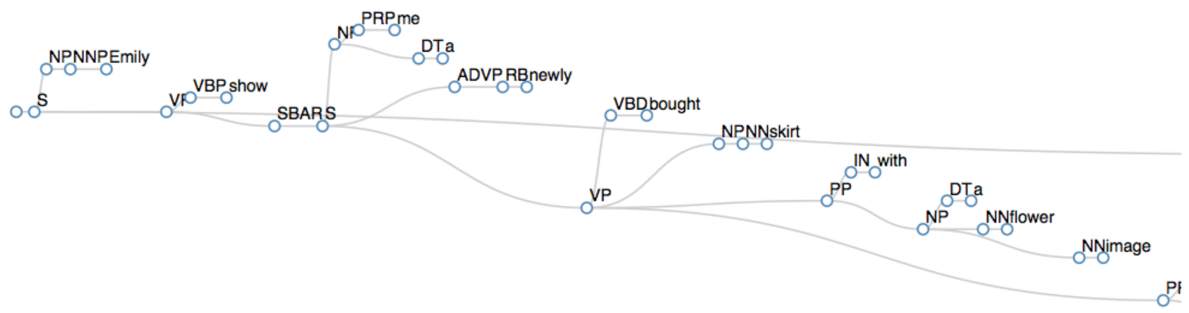


Fig. 4: The simple modification of level-based parsed tree of the sentence "Emily show me a newly bought skirt with a blue flower image on it."

extended Reingold and Tilford algorithm [4] to draw the non-layered trees.

III. Defining Indented Level-Based Tree Drawing Problem

The indented level-based tree drawing (see Fig. 5 for example) algorithm has the following properties:

1) A left child and a right child should be positioned on the right of their parent node.
2) The root node is the leftmost node of the tree. The tree is displayed horizontally.
3) The horizontal coordinates of the leaf nodes should be indented in such a way that, starting with the top leaf node, the leaf nodes are sorted horizontally from left to right. In other words, the leaf nodes can be read from left to right as a sentence.

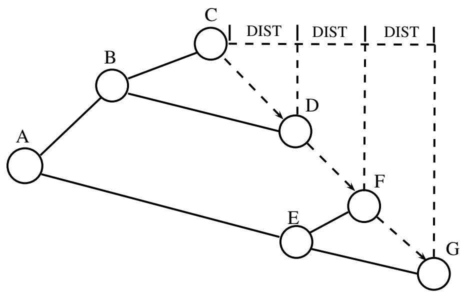


Fig. 5: An example of the indented tree drawing. The leaf nodes are sorted from left to right horizontally.

The indented tree drawing, due to its third property, will first sort leaf nodes horizontally from left to right, and then vertically from top to bottom. Therefore the tree drawing problem is then reformulated as follows: given an input tree structure, calculate the horizontal and vertical coordinates for each node of the tree so that the drawing is compact and satisfies the properties as listed above.

IV. Reingold and Tilford Algorithm

Reingold and Tilford Algorithm is one of the most important and influential algorithms in the area of tree drawings, and serves as the basis for our tree drawing algorithm. In this section, we discuss its basic ideas and the major challenges it has addressed.

Reingold and Tilford Algorithm is inspired by Wetherell and Shannon algorithm [2]. Wetherell and Shannon algorithm assigns equivalent vertical coordinates to nodes at the same level, and keeps track of the next leftmost available horizontal positions on each level when positioning nodes in an attempt to keep the width of the tree minimal. It works well in many cases, but can cause unpleasing result in some cases as the width between two sibling nodes is unnecessarily expanded. Reingold and Tilford proposed a new aesthetic (Aesthetic rule #4), stating that a subtree should be drawn the same way regardless of where it occurs in the tree. They designed a new algorithm to incorporate this new aesthetic. Instead of statically assigning the leftmost available position to nodes, the algorithm recursively computes the horizontal coordinates of the nodes and dynamically adjusts their positions. The brief description of the algorithm is as follows:

Two tree traversals are used to produce the final horizontal coordinates of nodes while their vertical coordinates can be pre-determined with their levels. The first post-order traversal assign the preliminary horizontal coordinates and modifier fields for each node. The second pre-order traversal compute the final horizontal coordinates for each node by summing its preliminary horizontal coordinates and modifier fields of all of its ancestors. In the post-order tree traversal, starting from leaf nodes (the smallest subtrees), smaller subtrees are positioned from left to right, and are combined with their parents to form greater subtrees. Parent nodes are placed in the center of their children. This process continues recursively until the root is reached.

The vital part of the algorithm is how to position sibling subtrees. For a given node, its subtrees are positioned one by one, from left to right. When positioning a new child subtree, the subtree is shifted right until the pre-defined distance between the new subtree and its left sibling subtree (or sub-forest) is reached. The process starts at the level of subtree root. The subtree is pushed towards right so that the roots of the subtree and its sibling subtree are separated by the *sibling separation* value. At the next lower level, the subtree is pushed towards right again if the leftmost node of the subtree and the rightmost node of its sibling subtree at that level is not separated by the *subtree separation* value. The

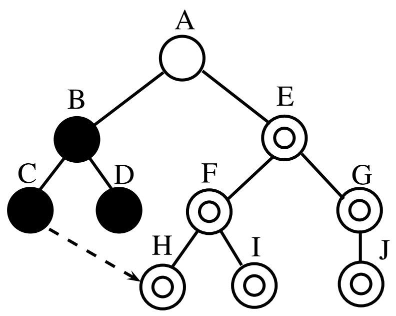


Fig. 6: Position subtree E on the right of subtree B. A traversal of right contour of subtree B and left contour of subtree E is needed to compute the proper distance to separate subtree B and subtree E.

process continues until the bottom of the shorted subtree is reached. The shifting distance of the subtree is stored in the modifier field of the subtree root. In order to run the above steps in $O(n)$, Reingold and Tilford introduced the concept of "contours". The left (right) contour of a subtree is defined as the sequence of leftmost (rightmost) nodes at each level in the subtree. A linked list data structure is used to maintain the contour. We demonstrate these concepts with an example in Fig. 6. For simplicity, we refer subtree A as the subtree rooted at node A. Subtree A consists of subtree B and subtree E. The left contour of subtree B contains node B and node C and its right contour contain node B and node D. Likewise, Nodes E, F, H and nodes E, G, J compose the left and right contour of the subtree E, respectively. When positioning subtree E, node pairs (B, E) and (D, F) are examined at different level, and subtree E will be shifted accordingly to maintain the predefined distance between nodes in node pairs.

The construction of contours is recursively performed while positioning subtrees. The parent subtree can form its left/right contour by linking its root and the left/right contour of its child subtrees. For example, the left contour of subtree A contain node A, the left contour of subtree B, and node H which is on the left contour of subtree E.

The second tree traversal, a preorder traversal, determines the final horizontal coordinates for nodes. It starts at the root of the tree, summing each node's preliminary horizontal coordinate value with the combined sum of the modifier fields of its ancestors.

V. Indented Reingold and Tilford Tree Drawing Algorithm

Our tree drawing algorithm is based on Reingold and Tilford algorithm [4] but with significant change, mainly because of the third property listed in Section III. Namely the leaf nodes need to be sorted horizontally. This property is in conflict with the Aesthetic rule #1 in the Reingold and Tilford algorithm. The underlying assumption in Reingold and Tilford algorithm is that the vertical coordinates of each node is determined by its level. In our case, this is no longer valid. Therefore, our algorithm needs to compute both horizontal and vertical coordinates recursively.

Here we still use a vertical tree for easier explanation. In the implementation, the tree is draw horizontally.

The algorithm requires three traversals of a tree. The first traversal is described in *firstWalk* procedure (See Algorithm 1). Starting from the leaves that we consider the minimal subtrees till the root of the whole tree, the algorithm recursively computes and stores the relative positions of the subtree root relative to their children (if they have any) and the relative positions of the subtrees relative to their sibling subtrees, respectively. The second tree traversal aggregates these relative positions, and compute the final position for each node. This process is described in *secondWalk* procedure (See Algorithm 2). The last tree traversal redistributes the vertical positions of nodes in order to make it look more pleasant while still maintaining the properties of the indented tree drawing (See Algorithm 3).

Algorithm 1 firstWalk procedure

```
1:  procedure FIRSTWALK(TreeNode $v$)
2:      TreeNode $w$                                  ▷ $v$'s left sibling
3:      List<TreeNode> $nodes$                        ▷ child of $v$
4:      int $mid$                                     ▷ center of child

5:      int $len$ ← $nodes$.length
        ▷ Line 6-11 horizontal coordinates
6:      if $v$ is a non-leaf node then
7:          mid=($nodes$[0].prelimX+$nodes$[$len$-1].prelimX)/2
8:          $v$.prelimX = mid;
9:      else if $w$ exists then
10:         $v$.prelimX=$w$.prelimX+SS                ▷ SS: const
11:     end if
        ▷ Line 12-19 vertical coordinates
12:     if $v$ is a non-leaf node then
13:         $v$.prelimY=$nodeY$.prelimY+$nodeY$.modY-DIST;
            ▷ DIST: const
            ▷ $nodeY$ is $v$'s child with the smallest vertical
    coordinates
14:         if $w$ exists then
15:             $v$.modY=$w$.modY+DIST*$w$.l;
                ▷ $w$.l denotes the number of its offspring leaves
16:         end if
17:     else if $w$ exists then
18:         v.prelimY=w.prelimY+DIST;
19:     end if
20:     if $w$ exists then
21:         Positioning $v$ on the right of $w$
22:     end if
23: end procedure
```

Like other positioning algorithms, our algorithm also uses two separate fields for the positioning of tree nodes. For a non-leaf leave, the *prelimX* or *prelimY* field of the node denotes its relative horizontal or vertical position to its children while the *modX* or *modY* field denotes the relative horizontal or vertical position that the subtree root node is from its sibling subtree. The position value in the *modX* or *modY* field of a node is assigned based on the entire subtree rooted at the node, and

Algorithm 2 secondWalk procedure

```
1: procedure SECONDWALK(TreeNode $v$)
2:     v.modX += v.parent.modX;
3:     v.x = v.prelimX + v.modX;
       ▷ Line 2-3 Compute final horizontal coordinates
4:     v.modY += v.parent.modY;
5:     v.y = v.prelimY + v.modY;
       ▷ Line 4-5 Compute final vertical coordinates
6: end procedure
```

will be applied to all of its offspring nodes when calculating their final coordinates. For a leaf node, only the *prelimX* or *prelimY* is needed to denote its relative horizontal or vertical position to its leftmost siblings. We assume in this paper that the coordinate system has its original point at the top-left corner. That is, if the height of a node is greater than that of another node, then the node has a smaller vertical coordinate.

In order to decrease the complexity of computation, the algorithm is designed to decouple the horizontal and vertical positioning of nodes. There are three major steps that will change the positions of the nodes. The first one is the initial assignment of both horizontal and vertical positions to nodes in accordance to the rules in the first tree traversal. The next step is the subtree positioning, which adjusts the horizontal positions of nodes. The third step is the node redistribution step in the third tree traversal, which tunes the vertical positions of nodes. We discuss these three steps in more details in the coming sub-sections.

A. Assigning initial positions to nodes

We have defined rules for the assignment of horizontal and vertical positions. For horizontal positioning, the rules state that (1) if a node is a non-leaf node, then place it in the center of its children; (2) if a node is a leaf with no sibling, assign 0 to *prelimX* field. (3) if a node is a leaf with a left sibling, then place it to the right of its left sibling at a pre-defined distance. The pseudo codes for these rules are listed in line 6-11 in Algorithm 1.

The rules for assigning vertical positions to nodes are more complicated:

Rule 1: If a node is a leaf with no left sibling, assigns 0 to its *prelimY* field.

Rule 2: If a node is a leaf with a left sibling, in order to satisfy the third property of indented tree drawings, the algorithm assigns w.prelimY+$DIST$ to the node's *prelimY* field where w is the node's left sibling and $DIST$ is the pre-defined distance.

Rule 3: If a node is a non-leaf node with no left sibling, in order to guarantee that a node is higher than any of its children, the algorithm assigns $nodeY$.prelimY+$nodeY$.modY-$DIST$ to the node's *prelimY* field where $nodeY$ is the child which has the greatest height.

Rule 4: If a node is a non-leaf node with left siblings, not only should this node be higher than any of its children, but the entire subtree rooted on this node is also moved upwards by $DIST$*w.l where w.l represents the number of leaf node in its sibling subtrees. This is so because in order to satisfy the third property of indented tree drawings, the leftmost leaf of the current subtree should be $DIST$*w.l higher than the leftmost leaf of its sibling subtree.

The pseudo codes for the above rules are listed in line 12-19 in Algorithm 1.

B. Positioning subtrees

The horizontal positioning discussed above only calculates the relative position of a node to its siblings or children. The task of positioning subtree aims at computing the relative position of the current subtree to its sibling subtree (Line 21 in Algorithm 1). Positioning a subtree on the right of its sibling subtree in the level-based tree drawing is a complicated process since the algorithm has to travel the sequence of rightmost nodes in the left subtree and the sequence of leftmost nodes in the right subtree in order to determine the minimal shifting distance that can separate two subtrees at a pre-defined distance. (We have discussed this process in Section IV.) However, positioning subtrees in our tree drawing algorithm is greatly simplified and can be achieved in $O(1)$ time.

For a given subtree, we argue that its leftmost node is its leftmost leaf. The proof is as follows: Assume the above statement is not correct. In other words, the leftmost node of a subtree is a non-leaf node. Since this is a non-leaf node, its offsprings contain at least one leaf node, and according to the second property of our tree drawing algorithm it is centered among its children. If the non-leaf node has only one leaf node, then it has the same horizontal coordinates as its leaf node. This can prove that the leftmost of a subtree is a leaf node. Otherwise, it must have a leaf node that is positioned on its left. This is a contradiction. So we prove that for any given subtree, its leftmost node is its leftmost leaf. With similar method, we can also prove that a subtree's rightmost node is its rightmost leaf.

This property makes the task of positioning subtrees quite simple. Traveling the left/right contour of right/left subtrees is unnecessary because if the rightmost leaf in the left subtree and the leftmost leaf in the right subtree are properly separated, then other nodes in these two subtrees will be well separated. In Fig. 7, for example, if node D and node H are well separated, then nodes within subtree B and nodes within subtree E are also well separated. The distance that a subtree is moved is equivalent to the distance that the subtree's leftmost leaf is moved so that the leaf is properly separated from the rightmost leaf of its left sibling subtree.

C. Redistributing nodes

In our algorithm, left sibling subtrees are gradually lifted up in order to line up the leaf nodes from left to right. In most cases this would not cause problems. However, in some cases where the left subtree contains many more nodes than the right subtree, the resulting tree drawing may look less pleasant since nodes within the right subtree are not vertically distributed evenly. In Fig. 8, the tree drawing on the left does not look good as node A and node C is unnecessarily distant.

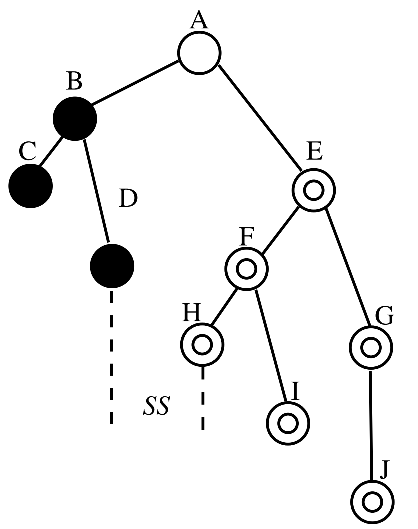


Fig. 7: Node D is the rightmost node in subtree B. Node H is the leftmost node in subtree E. The task of positioning subtree E is equivalent to separating Node D and Node H horizontally at a distance of SS (a pre-defined constant).

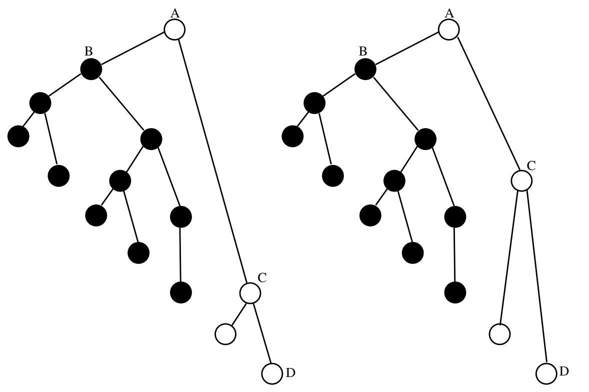


Fig. 8: The left tree drawing is less pleasant. The right tree drawing is more pleasant after node C is repositioned.

The tree drawing on the right looks more pleasant after node C is repositioned between node A and node D.

The task of node redistribution is to carry out another tree traversal and redistribute the vertical positions of nodes recursively so that the nodes within subtrees are vertically more evenly distributed.

To implement the node redistribution, the algorithm requires another pre-order tree traversal. For any visited node, the algorithm recalculates the vertical coordinate for each of its children. The calculation is based on the idea that nodes at different levels within the visited node's subtree should be vertically and evenly positioned between the visited node and the leaf. As described in Algorithm 3, the *thirdWalk* procedure computes the allocatable space between the node and its child, evenly splitting the space among different levels of the subtree rooted on the child. It then determines the new vertical coordinate of the child node.

Algorithm 3 thirdWalk procedure

```
1: procedure THIRDWALK(TreeNode $v$)
2:     for each child $u$ of $v$ do
3:         $spots \leftarrow (u$.y-$v$.y)/DIST-1;
           ▷ $spots$: # of vertical positions between $v$ and $u$.
4:         $level \leftarrow u$.level
           ▷ for a leaf node, its level is 1
           ▷ for a non-leaf node, its level is 1 plus the max of its child's level
5:         $ave \leftarrow spots/level$
           ▷ $ave$: # of vertical positions $u$ will be pulled toward its parent.
6:         $u$.y $\leftarrow u$.y-$ave$*DIST
7:     end for
8: end procedure
```

VI. Algorithm Analysis

Our algorithm is composed of three tree traversals. In the first post-order tree traversal, for each node, the initial assignment of coordinates takes $O(1)$ time, and the associated subtree positioning takes $O(1)$ time. So the total time complexity for the first tree traversal is $O(n)$. The second pre-order tree traversal takes constant time for each node to sum up the coordinates of its parent node. Therefore, its total time complexity is $O(n)$. The time complexity of the last tree pre-order tree traversal is also $O(n)$. We conclude the time complexity for the algorithm is $O(n)$.

VII. Implementation and Usage

We have implemented the indented level-based tree drawing algorithm in JavaScript and integrated it into D3.js [14], a popular JavaScript library for data visualization. The *Layouts* package in D3.js library provides efficient implementation of layout algorithms for various structures including the classic level-based tree. It also offers helper functions to facilitate the implementation of new layout algorithms.

Based on D3.js, we have added new APIs for quick construction of the indented tree drawing. We explain some of the APIs in Table I:

APIs	***Description***
d3.layout.indentedtree	Creates a new indented tree layout.
indentedtree.size	Sets the available layout size.
indentedtree.sort	Sets the sort order of sibling nodes.
indentedtree.separation	Sets separation between nodes.
indentedtree.nodes	Computes the tree layout.
indentedtree.links	Returns edge positions.

TABLE I: APIs for the indented tree drawings

With these APIs, Web application developers only need to write a few lines to draw indented trees and embed them in their Web pages. The code snippet in Listing 1 provides a showcase for drawing a simple indented tree.

Line 2 creates an instance of indented tree. Then in Line 3 the size of the tree drawing is specified. The positioning

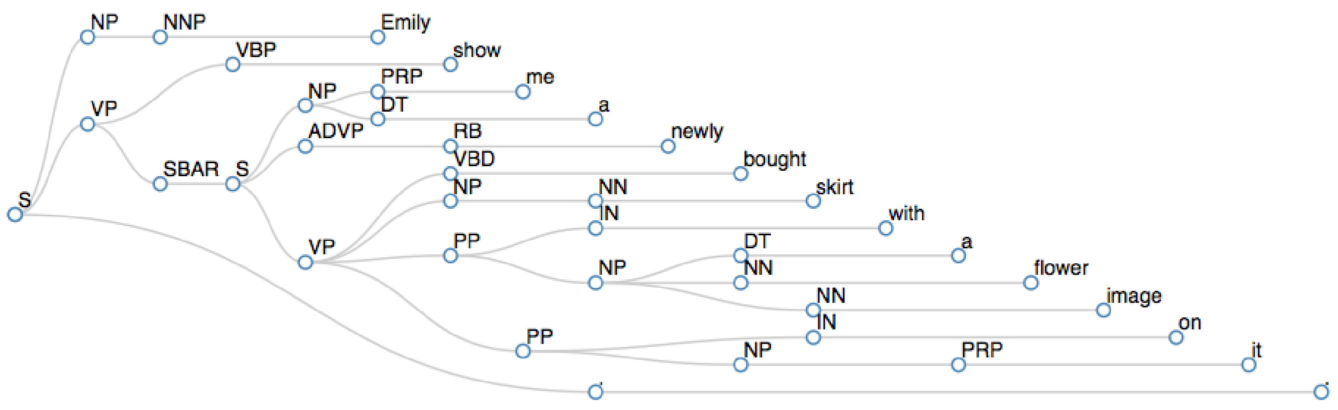


Fig. 9: The indented level-based parsed tree of the sentence "Emily show me a newly bought skirt with a blue flower image on it."

algorithm is invoked in Line 6, and tree nodes' horizontal and vertical coordinates are returned. The positions of tree edges are computed by pairing parent nodes and child nodes in Line 8. Once the position of tree nodes and edges are specified, then other D3 drawing routines can be used to draw the indented tree. Fig. 9 shows an example of the indented level-based tree drawing created with our new APIs. Comparing that with the traditional parsing tree in Fig. 1, we can see that our drawing has a wider aspect ratio that fits better with newer computer displays. It also places the words from left to right for easy reading, while clearly presenting the syntactic structure of the sentence.

```
1  // Create a tree layout
2  var tree = d3.layout.indentedtree();
3  tree.size([height, width]);
4  // Compute nodes' positions
5  // root: input data
6  var nodes = tree.nodes(root);
7  // Retrieve edges positions
8  var edges = tree.links(nodes);
9  Draw nodes and edges with D3 routines.
```

Listing 1: Sample code for drawing an indented tree

VIII. Conclusion

In this paper, we propose a new indented level-based tree drawing algorithm. This is a modified Reingold and Tilford algorithm that satisfies a new aesthetic rule – make the tree fit the wider aspect ratio of the newer computer display while preserving the order of the leaf nodes in a sentence. This is motivated by the need of visualizing parse trees in a text visualization application. The new algorithm creates a parse tree drawing that makes optimal use of the space while maintaining the word order for easy reading. The new algorithm also solves the problems of line crossings for tidy presentation.

The proposed tree drawing algorithm is a useful complement to the traditional level-based tree drawing algorithms. In addition to drawing parse trees, it can be applied to any tree structure where the leaf nodes need to be horizontally sorted.

References

[1] D. E. Knuth, "Optimum binary search trees," *Acta informatica*, vol. 1, no. 1, pp. 14–25, 1971.

[2] C. Wetherell and A. Shannon, "Tidy drawings of trees," *IEEE Transactions on Software Engineering*, no. 5, pp. 514–520, 1979.

[3] R. E. Sweet, "Empirical estimates of program entropy," Xerox Res. Cent., Tech. Rep., 1978.

[4] E. M. Reingold and J. S. Tilford, "Tidier drawings of trees," *IEEE Transactions on Software Engineering*, no. 2, pp. 223–228, 1981.

[5] J. Q. Walker, "A node-positioning algorithm for general trees," *Software: Practice and Experience*, vol. 20, no. 7, pp. 685–705, 1990.

[6] C. Buchheim, M. Jünger, and S. Leipert, "Improving walkers algorithm to run in linear time," in *Graph Drawing*. Springer, 2002, pp. 344–353.

[7] R. Tamassia, "Handbook of graph drawing and visualization," *CRC Press 2013*.

[8] Y. Miyadera, K. Anzai, H. Unno, and T. Yaku, "Depth-first layout algorithm for trees," *Information processing letters*, vol. 66, no. 4, pp. 187–194, 1998.

[9] A. Bloesch, "Aesthetic layout of generalized trees," *Software: Practice and Experience*, vol. 23, no. 8, pp. 817–827, 1993.

[10] B. Stein and F. Benteler, "On the generalized box-drawing of treessurvey and new technology," in *International Conference on Knowledge Management*, 2007, pp. 416–423.

[11] X. Li and J. Huang, "An improved generalized tree layout algorithm," in *Informatics in Control, Automation and Robotics (CAR), 2010 2nd International Asia Conference on*, vol. 2. IEEE, 2010, pp. 163–166.

[12] K. Marriott, P. Sbarski, T. van Gelder, D. Prager, and A. Bulka, "Hi-trees and their layout," *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, no. 3, pp. 290–304, 2011.

[13] A. Ploeg, "Drawing non-layered tidy trees in linear time," *Software: Practice and Experience*, 2013.

[14] M. Bostock, V. Ogievetsky, and J. Heer, "D^3 data-driven documents," *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, no. 12, pp. 2301–2309, 2011.