Visualizing Workflow Structures Using a Modified Tree Layout Algorithm

Anđelija Đorđević[1,*], Petar Rajković[1], Aleksandar Milenković[1], Dejan Aleksić[2], Đorđe Čikić[1] and Dragan Janković[1]

[1] University of Niš, Faculty of Electronic Engineering, Aleksandra Medvedeva 4, Niš, Serbia
[2] University of Niš, Department of Physics, Faculty of Sciences and Mathematics, Višegradska 33, Niš, Serbia

Abstract

This paper presents a novel adaptation of the Reingold-Tilford algorithm designed for the hierarchical visualization of workflows in product manufacturing. While originally designed for tree structures, the algorithm is reinterpreted to suit workflows, which often include multiple entry points, nodes with multiple parents, and a unique terminal node representing the single final product. The proposed Workflow algorithm modifies the tree layout algorithm to reflect these features while maintaining aesthetic coherence and clear predecessor-successor relations. The resulting visualization supports intuitive understanding, traceability, and real-time usability within manufacturing process monitoring, offering a clear view of process flow and status. This work highlights how a classic layout approach can be effectively tailored to the specific needs of industrial workflows.

Keywords

Workflow visualization, tree layout algorithm, workflow structure, hierarchical visualization

1. Introduction

Trees are fundamental data structures commonly used to represent hierarchical data. In addition to their roles in storage and processing, there is a need to visualize trees, allowing users to gain insights and manipulate the structure effectively. This need has led to the development of various algorithms for tree visualization, each approaching the task in its own way [1].

When visualizing trees, there tends to be a preference for tidy drawings. In 1979, Wetherell and Shannon [2] defined the following three aesthetics for creating tidy tree drawings.

Aesthetic 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 2: In a binary tree, each left son should be positioned left of its father and each right son right of its father.

Aesthetic 3: A parent should be centered over its children.

Wetherell and Shannon, in addition to formalizing three key aesthetics for tree representation, highlighted that edges should not intersect. The only points of intersection in the tree should occur at the nodes. Additionally, no node should be positioned closer to the root than its predecessors. Lastly, it is important to minimize the usable space, aiming for a dense arrangement of nodes.

In their work [3], Reingold and Tilford built upon Wetherell and Shannon's findings [2] and advanced Wetherell and Shannon's algorithm (the WS algorithm) to what is now known as asymmet-

*SQAMIA 2025: Workshop on Software Quality, Analysis, Monitoring, Improvement, and Applications, September 10-12, 2025, Maribor, Slovenia

[1*] Corresponding author.
[†] These authors contributed equally.
andjelija.djordjevic@elfak.ni.ac.rs (A. Đorđević); petar.rajkovic@elfak.ni.ac.rs (P. Rajković); aleksandar.milenkovic@elfak.ni.ac.rs (A. Milenković); alexa@pmf.ni.ac.rs (D. Alekić); djordje.cikic@elfak.ni.ac.rs (Đ. Čikić); dragan.jankovic@elfak.ni.ac.rs (D. Janković);
0000-0002-4760-8641 (A. Đorđević); 0000-0003-4998-2036 (P. Rajković); 0000-0003-1796-7536 (A. Milenković) ; 0000-0002-5833-6194 (D. Aleksić) ; 0009-0003-2931-3596 (Đ. Čikić) ; 0000-0003-1198-0174 (D. Janković)¶

rically portray symmetrical trees Reingold-Tilford algorithm (the RT algorithm), which serves as a foundation for tree visualization. They observed that the WS algorithm can portray symmetrical trees in an asymmetrical way, meaning the tree and its reflection are not drawn as mirror images. Furthermore, the same subtrees can be displayed differently depending on their position within the larger tree structure. These observations led Reingold and Tilford to enhance the formal aesthetics with a fourth rule defined in [3]:

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

Reingold and Tilford emphasized that the order of nodes at all levels should mirror the order found in an in-order traversal of the tree. These principles formed the foundation for the development of the RT algorithm. Since then, numerous algorithms have been created based on the RT algorithm with the goal of modifying or improving it [4, 5].

The visualization of a tree can vary depending on the specific context of a problem, which is one reason why different visualization algorithms exist. These algorithms can also be applied to visualize other issues based on tree structures, even if they are not strict tree representations. This work originated from the idea of visualizing workflows that represent how a product is created in a factory. A workflow outlines the various steps involved in producing a final product. These steps can occur sequentially or in parallel across different departments of the factory [13, 14]. Production may start in one department but can also be distributed among several departments working simultaneously. The output from one department can either be routed entirely to another, divided into two or more new departments, or combined with results from other departments before being passed on to a new one. Ultimately, all these processes lead to the creation of a single final product, with the last step in the series always resulting in just one product.

While workflows are not strictly trees in the graph-theoretical sense, they often exhibit structural similarities that make a tree-based graphical representation both intuitive and useful. Although such structures do not meet the formal definition of a tree, they can still be described using tree terminology. In this context, the graphical representation of the workflow that this paper describes does not need to have a unique root. Instead, there can be one or more nodes at the top level. Each internal node may have multiple children as well as multiple parents. At the bottom level, however, there is always exactly one node, which is referred to as an only leaf and represents the final product. This perspective allows tree visualization techniques to be adapted to workflows in a way that enhances clarity, traceability, and practical applicability in production environments.

This paper introduces essential tree terminology and aesthetic criteria for the hierarchical visualization of trees, and the paper structure is as follows. Section 2 provides an overview of related work, briefly summarizing existing approaches to visualizing trees, hierarchical structures, flows, and workflows. Section 3 briefly explains the steps involved in drawing trees using the RT algorithm, which serves as a foundation for problem-solving. In Section 4, the details of the modifications and adaptations made to this algorithm to meet the needs of workflow visualization, specifically focusing on the creation of the WF algorithm, are described. Finally, the paper concludes with a discussion and final remarks.

2. Related work

How trees are visualized can vary significantly depending on the purpose and context in which they are used. Tree layout algorithms, originally developed for strictly hierarchical structures, can often be adapted to visualize other types of data that do not formally qualify as trees but share similar characteristics. With appropriate modifications, existing tree visualization methods can be effectively repurposed to address a wide range of problems beyond pure tree structures.

In addition to the hierarchical structure of trees described in the previous section, there is a need for tree visualization methods that are not strictly hierarchical due to the requirement to display large amounts of data and to ensure scalability. Tree maps, as described in [6] and [7], are used for direct access and to visualize value changes. Cone trees were introduced in [8] and further

improved in [9] and [10] to represent and navigate large hierarchies. The Bubble Tree Drawing Algorithm [11], unlike the RT algorithm that relies on contour, employs the concept of enclosing circles to represent the space necessary for drawing a subtree. The Bubble Tree algorithm is designed for drawing general rooted trees, with a key advantage being its focus on enhancing angular resolution and aesthetic criteria.

Tree visualizations can be categorized into two types: connection and enclosure [12]. The connection type represents a natural way of drawing trees, where nodes symbolize data and edges illustrate the connections between them. Various algorithms fall under this category, including cone trees [8], hyperbolic trees [17], radial views [18, 19], balloon views [8], disk trees [20], classical hierarchical trees [3], botanical visualizations [21], and NicheWorks [22]. On the other hand, enclosure is utilized to illustrate the tree structure itself. In this approach, each node is represented by a single region, which is then subdivided to depict its children. This division continues recursively through the descendants. Common representations of this type include tree maps [7] and Venn diagrams [23]. The concept of Space Optimized Trees [12] introduces a novel method where children are accessed through polygons. Enclosure is particularly effective for representing quantitative values associated with nodes.

In addition to general-purpose methods for tree visualization, which primarily focus on representing abstract hierarchical structures, there exists a separate class of visual representations aimed at modeling and communicating operational processes. These include workflows, which, while structurally like trees in some aspects, are typically more complex and dynamic. While standard workflow modeling notations such as UML activity diagrams [24], BPMN [25], and Petri nets [26] are widely used for expressing process logic in enterprise and engineering contexts, the approach presented in this paper is specifically tailored to the needs of workers in a production environment. The visualization is integrated directly into the Manufacturing Execution System already in use, with the goal of enhancing clarity, traceability, and real-time usability. Workflow nodes are enriched with textual descriptions and dynamically colored based on execution state, offering an intuitive overview of the current process flow. This practical representation prioritizes interpretability and ease of use over formal expressiveness, making it more suitable for day-to-day operations in manufacturing settings. To achieve this, the underlying layout is based on a modified version of the RT algorithm, originally designed for tree structures, which is here adapted to accommodate the specific characteristics of production workflows while preserving visual clarity and hierarchical intuition.

3. Tree visualization algorithm

The RT algorithm served as the foundation for developing the workflow (WF) visualization algorithm. As previously mentioned, the RT algorithm was based on the WS algorithm. This section will detail the method for implementing the RT algorithm, which includes aspects of the WS algorithm. For the program's implementation of the RT algorithm, which will later be upgraded to the WF algorithm, existing implementations in [15] and [16] were utilized.

The RT algorithm starts with the premise that plotting the tree requires knowledge of each node's position. Once the positions are established, a straightforward plotting or printing routine can generate a tree diagram. To plot the nodes accurately, it is essential to know the X and Y coordinates of each node. The Y coordinate is determined by the node's depth, which corresponds to its level in the tree. Additionally, the vertical distance between nodes must be calculated to pinpoint the exact Y coordinate. On the other hand, calculating the X coordinate is vital, as it plays a crucial role in the plotting process. Algorithms for tree visualization primarily revolve around determining the X coordinate.

A postorder traversal of the tree was used to calculate the X coordinate. This process can be divided into four phases: determining the initial X coordinate, ensuring that all nodes are visible on the screen, positioning the leaves in the center, and finally, determining the final X coordinate.

3.1. Determining the initial X coordinate

When determining the initial X coordinate, the approach differs depending on whether the node is a leaf or an internal node.

3.1.1. Determining the X-coordinate of the leaves in relation to their left sibling

In the case of a leaf node, the calculation of the X coordinate is based on its sibling nodes. More precisely, the X coordinate is determined solely by the ordinal position of the node among its siblings in the parent's list of children. If the node is the first child, its initial X value is set to zero. For any other node, the X value is calculated as the sum of the X value of the left sibling and a specified distance between the nodes.

3.1.2. Determining the X coordinate of internal nodes

For internal nodes, the X coordinate is determined based on the left sibling and by centering the parent node over its children. Let's determine the X coordinate of node A, which has parent P and the oldest child C.

If node A has only one child, referred to as node C, the method for determining the X coordinate will vary depending on whether node A has left siblings. If node A is the first child of its parent P, the X coordinate will be set to the same value as the X coordinate of node C. If node A has left siblings, the X coordinate will be set to the sibling distance from the left sibling. After that, the entire subtree rooted at node A (without moving the root) needs to be centered relative to node A by modifying the MOD property of node A, as shown in the following pseudocode.

```
this.X = this.PreviousSibling().X + SIBLING_DISTANCE;
this.MOD = this.X − this.LeftMostChild().X;
```

If node A has more than one child, the temporary X coordinate for the parent will be the average of the X coordinates of the leftmost and rightmost children of node A. If node A is the oldest child of its parent P, its X coordinate will be the previously calculated temporary X coordinate.

```
this.X = (this.LeftMostChild().X + this.RightMostChild().X) / 2;
```

If node A has left siblings, the X coordinate will be set to the sum of the X coordinate of the left sibling and the sibling distance. Finally, it is necessary to center the entire subtree with the root at node A (without moving the root) relative to node A, as shown in the following pseudocode.

```
this.X = this.PreviousSibling().X + SIBLING_DISTANCE;
this.MOD = this.X − (this.LeftMostChild().X + this.RightMostChild().X) / 2;
```

It is important to explain how to move a subtree rooted at an internal node without moving the root itself. This operation is necessary whenever the root of the subtree is not the leftmost child of its parent. Depending on the tree structure, this can be a costly operation. If every node of the subtree were to be passed through in the subtree and repeatedly update their X coordinates, the total number of visits across the entire tree would increase significantly, leading to higher algorithmic complexity.

To avoid this issue, a property called MOD has been introduced. The MOD property for an arbitrary node A contains a value that indicates how much to move all children of node A along the X coordinate. It's important to note that while the MOD value of node A allows its descendants to be moved along the X axis, the X coordinate of node A itself remains unaffected by these changes. Instead, the X coordinate of node A is influenced by the MOD values of all its ancestors. Therefore,

when the algorithm states that an entire subtree is being moved by a certain value, it implies that the MOD value of the subtree's root will be updated accordingly.

3.1.3. Reviewing any conflicts with previous trees

In previous calculations of the X coordinates for tree nodes, the main focus was on preventing overlaps with left-sibling nodes and centering parent nodes over their children. However, the process of determining the X coordinate is not yet fully complete. In each level of the tree, there could be multiple nodes that do not have to be siblings, meaning they may not share the same parent. So far, conflicts between nodes at the same level have not been addressed.

When processing a node that is neither a leaf nor a leftmost child of its parent, it is necessary to check for conflicts between the subtrees. To determine if there is a conflict, a minimum distance between trees must be set. For node D, the conflict is assessed by comparing the subtree rooted at D with each subtree rooted in all of node D's left siblings. In Figure 1 those would be nodes B and C. The comparison starts with the leftmost sibling and continues until reaching node D itself.

Conflicts are examined at each level of the subtree as contours are created. The left contour of node D and the right contour of each of its left siblings are established. To create a contour, a Dictionary structure can be used, where for each level y, the X coordinate of the node that serves as the rightmost at that level in the analyzed subtree is recorded for the right contour, and the leftmost is recorded for the left contour. In Figure 1 the blue nodes represent the right contour of the subtree rooted in the node B, and the pink nodes represent the left contour of the subtree rooted in the node D.

After establishing the left contour for the subtree rooted at D and the right contour for the left siblings of node D, it is essential to examine all contour levels and calculate the shortest distance between tree contours. If the shortest distance between tree contours is less than the minimum distance between trees, node D and all its children should be shifted accordingly. Specifically, the X coordinate and MOD property of node D should be increased by the difference between the minimum distance between trees and the shortest distance between tree contours.

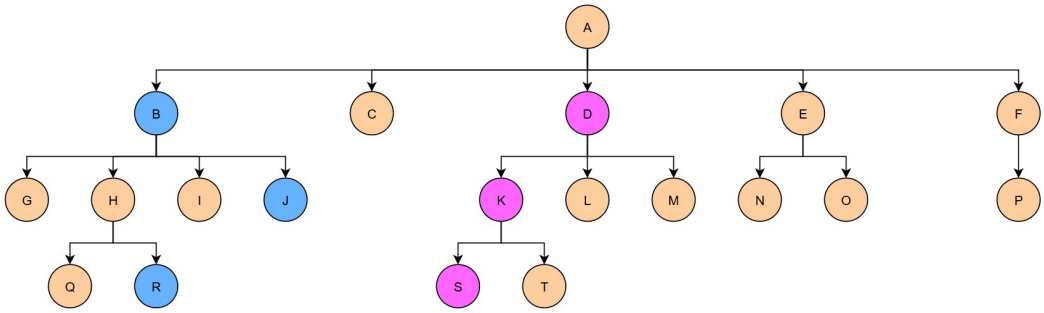

Figure 1: The right contour of node B and the left contour of node D.

3.2. Ensuring that all nodes are displayed on the screen

After performing all the specified operations and making several changes to the X, the final X coordinate can be negative. To address this issue, it is unnecessary to traverse the entire tree. Instead, we can simply determine the left contour of the tree.

Similar to the previous case, once the contour is identified, we can examine all levels of the tree to find the smallest X coordinate among the nodes in the left contour. If this smallest coordinate is less than 0, we will need to adjust the root's X coordinate.

3.3. Place the leaves in the center

After completing all the previous steps, the nodes are now correctly positioned concerning their left siblings, centered in relation to their children, and there is no overlapping of nodes or subtrees.

All nodes are displayed appropriately on the screen. However, one potential issue remains: the centering of leaves that are positioned between their parents, meaning they are neither the leftmost nor the rightmost children.

While each internal node is centered with respect to its children and placed correctly concerning its left sibling, leaves are only positioned relative to their left siblings. Each parent node is positioned in the center between the leftmost and rightmost children. Although nodes can be shifted further due to the MOD properties of their descendants, leaves, which do not have children, are not centered concerning them. As a result, leaves that are situated in the middle of their parents may not be centered relative to their siblings.

The traversal begins at the root and is performed recursively in a preorder manner. If the current node has more than one child, the traversal of the children proceeds from the rightmost child to the leftmost child. If the node being examined is a leaf and is neither the leftmost nor the rightmost child of its parent, its X coordinate will be calculated as the mean of the X coordinates of its left and right siblings. Therefore, the leaf node is centered between its left and right siblings.

3.4. Determination of the final X coordinate

After considering all the cases and changes to the X coordinates and MOD properties of the nodes, the final X coordinate should be determined. The final X coordinate will be the sum of the current X coordinate of the node and the MOD values of all its descendants.

To calculate this, a recursive function that takes one parameter *modSum* is used. During the execution of this function, the X coordinate of node A is incremented by *modSum*, and then *modSum* is updated by adding the MOD property of node A. The function is called for each of node A's children using the updated *modSum* parameter.

By the end of this function, each node will have its final X and Y coordinates determined, enabling the nodes to be drawn in the shape of a tree.

4. Workflow visualization

The distinction between the RT and WF algorithms is evident in the data structures they visualize. The TR algorithm is specifically designed for visualizing trees. The WF algorithm is intended to represent the flow of creating a product. Consequently, the visualization must depict the sequence of operations required and the order in which they must be performed. Unlike trees, workflows do not have a single root, instead, they can have multiple starting nodes. Each node, except for the starting nodes, may have one or more parent nodes. Internal nodes can also have one or more child nodes. Notably, there is only one leaf node, which is the terminal node that has no successors. This terminal node signifies the final product.

The visualization of this workflow resembles that of a tree. It is essential to clearly distinguish between levels with nodes and to differentiate predecessor nodes from their successors. Each predecessor must be positioned at a level above its respective successor. Additionally, it is important to incorporate aesthetics 1 and 3 when visualizing the tree, along with other recommendations, such as avoiding edge crossings. In tree visualizations, the only allowed intersection of edges occurs at nodes. However, since a node in a workflow can have multiple parents, it is permissible to merge edges that lead from several parents into one child node. In this case, the joining point of these branches should be above the child node, specifically at the same X coordinate as the child but at a higher Y coordinate.

For trees, it is accepted that no node should be positioned closer to the root than its predecessors. Given that workflows do not have a root, we will consider that no node should be closer to the initial nodes than its predecessors. As with tree structures, it is crucial to minimize the usable space, striving for a dense arrangement of nodes. To visualize the workflow with these characteristics, modifications to the RT algorithm were necessary. For the WF algorithm, the implementation of the RT algorithm described in the previous section was used. In this section, the changes that the RT algorithm leads to the WF algorithm will be described.

For each node, its X and Y coordinates are recorded, where the Y coordinate indicates the depth at which the node is located. Each node has a MOD property, as explained in the previous section, which optimizes the algorithm by determining how much the descendants of the current node should be shifted in the final calculation of the X coordinate. Each node also keeps track of its previous and next siblings, remembers its first child, and has a boolean property that indicates if it has more children. If the node has additional children, the list of all its children is also maintained.

Since each node can have multiple parents, similar to its children, each node records its main parent. There is a boolean property that indicates whether the node has more than one parent, and if so, all parents are stored in an appropriate list. Additionally, for each node, it is noted whether it is a leaf node. Along with each node, the operation in the workflow that it represents is also recorded.

4.1. Creating a workflow structure

Workflow represents the process of creating a product. It illustrates the flow and interdependence of operations needed to produce the final item. The workflow is developed based on a single product, using the list of operations that constitute the product and outlining their mutual dependencies. Instances of the Node class were created for each operation. The previous operations that led to the current one are the node's parents and predecessors. Each operation is assigned a unique ID, which allows us to determine whether the corresponding node has already been established. For each newly created node, its predecessor is checked. If no such node exists, it is added to the list of root nodes. Conversely, if a predecessor does exist, a Node class object for that predecessor is created as well, connecting the two in a parent-child relationship. The child node is added to the parent node as the first child if the parent was previously a leaf node. Otherwise, it is added to the list of other children. The child node is assigned as the main parent if it has not had a parent before, or the identified parent is added to the other parents' list. After processing all mutual dependencies, nodes are created containing data about the operation they represent, along with lists of their parents and children.

4.2. Virtual root

Unlike a tree structure that has a single root, a workflow can consist of multiple initial nodes. Since all functions for drawing a tree start from a single root node and then proceed recursively through the children of that node, a *VirtualRoot* is created. The *VirtualRoot* is an instance of the Node class equipped with the same properties as the other nodes in the workflow. It serves as the initial node for calling functions for visualization, though its own rendering is skipped.

The *VirtualRoot* is assigned as the main parent for all nodes in the roots list, and all nodes from this list are added to the *VirtualRoot*'s children list. The depth of the *VirtualRoot* node is set to -1, which serves as the initial depth for all nodes. After creating the *VirtualRoot* node, a function is called to calculate the depth of this node and its descendants. The depth of a node in the tree is defined as the parent's depth plus one. However, since a node can have multiple parents in a workflow, the main parent property becomes relevant. The main parent of node N is defined as the parent located at the greatest depth of all nodes in the list of parents. It is essential to visually represent node N after its deepest parent to ensure a coherent visual flow of data. This method allows viewers to see the sequence of operations, illustrating which actions precede and which follow them.

4.3. Determining the X coordinate

As explained in the RT algorithm section, the function for determining the initial X coordinate is utilized in the postorder traversal of the tree. This function is first called recursively for all children, and then for the parent node. Within the function, the X coordinate is set based on the X coordinate of the left sibling, the MOD property is established, and the parent is centered over its children. After calculating the initial X coordinate, conflicts between the subtrees are checked.

As mentioned earlier, the processing of nodes varies based on whether a node is a leaf, has one or more children, and whether the left sibling is a child of its parent. While the cases considered and the values assigned for the X coordinate and the MOD value remain unchanged, it is essential to clarify how the workflow identifies whether a node is a leaf, counts its children, determines if the left sibling is a child, and distinguishes between the previous and next siblings.

In this workflow, a node is deemed a leaf if it has no descendants, similar to a leaf in a tree. However, this is not the sole criterion. A node is also considered a leaf if it is not the main parent of any of its children. This indicates that no children will be drawn beneath this node, allowing it to be classified as a leaf during the drawing process.

Node N is identified as the leftmost if it has no parent or is the only child of its main parent. If the main parent P of node N has more children, the first child of node P to which P is the main parent is searched. If this child is node N, then node N is considered to be the leftmost.

To determine if a node is the rightmost, the process is analogous to that of identifying the leftmost. If node N has no parent or is the only child of its main parent, it is classified as the rightmost. If not, the search is conducted for the last child of the main parent P of node N.

A node N in a workflow is considered to have one child if it has only one child for which it is the main parent. If N has multiple children, it is necessary to count all the children for which N is the primary parent. If this count equals one, then the node is considered to have only one child.

The methods for determining the previous and next siblings are commonly used for nodes. Initially, these values are set to null. Siblings are defined as nodes that share the same main parent, under which they will be drawn later. When calling these methods for a node N, the function checks whether its main parent P has more children. If so, it traverses the list of children belonging to node P to find the previous and the following sibling.

The leftmost child of node N is the first child of node N, where N is the main parent. Similarly, the rightmost child is the last child of node N, with N as the main parent.

Centering the leaves is crucial due to the potential asymmetry among sibling leaves. To center the leaves in the WF, the process begins at the *VirtualRoot* node and follows a preorder traversal. For each child C of node N where this function is invoked, the following conditions are checked: C's main parent must be node N, C must be a leaf, and C should not be the leftmost nor rightmost sibling among its siblings. If all these criteria are met, node C will be positioned in the middle of its siblings. If node C is not a leaf, the function is called recursively for node C.

The final X coordinate is determined based on the current X value and the MOD values of all main ancestors. The function is initially called with a *VirtualRoot* and a MOD value of zero as a parameter. This follows a preorder workflow traversal.

4.4. Plotting nodes and edges

After establishing the nodes, their interconnections, and their X and Y coordinates, nodes can be plotted. Before plotting, key parameters such as the height, width, and node distance should be determined. Nodes are depicted as rectangles with dimensions defined by the node height and width, and the position of each rectangle's upper left corner is determined.

The dimensions of the nodes should accommodate the content that will be placed inside them. If the content within the nodes is larger, it will be necessary to increase the dimensions accordingly. Additionally, to allow for different displays, the height and width of the nodes, as well as the distances between them, can be adjusted.

After each node is drawn, edges are created to connect it to each of its children. To draw an edge, it’s necessary to determine both the starting and ending points, along with any curves along the way. Each edge ends with an arrow. If the parent and child nodes share the same X coordinate, the stroke begins at the parent’s X coordinate, adjusted by half the node width to ensure that it is centered. The starting Y coordinate is calculated by the parent’s Y coordinate increased by the node height. A vertical line will extend downward to a point where the X coordinate matches that of the parent, but the Y coordinate is equal to the child’s Y coordinate minus half of the node distance.

From that point, a horizontal line will be drawn towards the child's X coordinate, which is again increased by half the node width. Finally, a vertical line will descend to the child node's final position. Figure 2 shows an example of a plotted workflow.

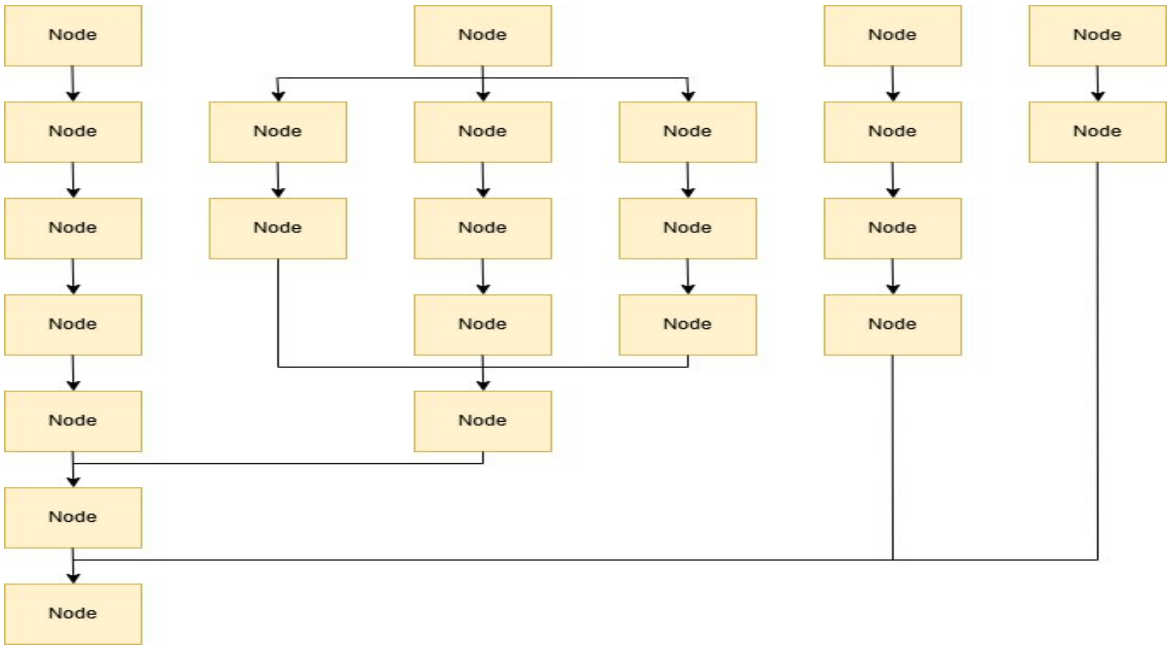

Figure 2: Example of a plotted workflow

5. Discussion

This paper describes the modification of the existing RT algorithm for drawing trees in order to visualize the workflow in production. As mentioned in the Related Work, there are various methods for visualizing trees based on the specific needs of the data. In a hierarchical representation of tree visualization, a parent node is always depicted above its child, with edges connecting them. This arrangement allows for the visual tracking of the order of predecessors and successors among the nodes.

The WF algorithm is intended to represent the flow of creating a product. It is crucial that the visualization clearly depicts the sequence of operations required and the order in which they must be performed. Although workflows are not trees, the importance of hierarchical representations of previous and subsequent operations resembles that of a tree. Therefore, the RT algorithm was used as a foundation for creating the WF algorithm, which aims to simplify the work in product manufacturing. Unlike trees, workflows do not have a single root, instead, they can have multiple starting nodes. Since workflow does not have a single root but can have several starting nodes, a placeholder node called *VirtualRoot* was introduced. Each node, except for the starting nodes, may have one or more parent nodes. Internal nodes can also have one or more child nodes. Notably, there is only one leaf node, which is the terminal node that has no successors. This terminal node signifies the final product.

The focus was primarily on the implementation aspects and the necessary algorithmic modifications to repurpose an existing tree visualization method for a different type of structure. The goal was to demonstrate how a tree-based algorithm can be effectively transformed to support the specific visualization needs of industrial workflows, to improve clarity and usability for workers involved in complex product creation processes. This paper also demonstrates that algorithms designed for one purpose can be adapted to solve different problems.

The proposed visualization provides the state of the production process, where each node represents a specific activity or checkpoint, annotated with textual descriptions and dynamically colored

to reflect its status. Unlike standard notations, this personalized layout emphasizes readability, simplicity, and immediate interpretability, enabling workers to quickly grasp the process flow and current progress. A formal comparison or mapping to existing modeling languages is beyond the scope of this work, but may be addressed in future research.

6. Conclusions

This paper provides an overview of tree visualization algorithms, focusing specifically on the development and enhancement of the RT tree drawing algorithm, one of the most widely used in this field. Section 3 describes a possible implementation of this algorithm based on various existing implementations. This implementation served as inspiration for addressing the workflow visualization challenges. The workflow does not meet the formal definition of a tree, but can still be described using tree terminology. In that terminology, the workflow node may have multiple parents. While there may not be a unique root, several nodes can exist without parents, and there is always one terminal node, a leaf.

To effectively visualize such a workflow, it was necessary to modify the existing tree visualization algorithm, as detailed in Section 4. This modification involved the introduction of a placeholder instead of the unique root, the list of node's parents and children, along with the term main parent and other explained modifications. Each node is assigned to a main parent, which is defined as the parent located at the greatest depth.

In addition to reviewing existing solutions and types of tree visualization, this work demonstrates how established and modified algorithms can be leveraged to tackle problems that were not initially considered in their design. This paper aimed to present a conceptual approach for adapting the well-known Reingold-Tilford tree layout algorithm to the domain of workflow visualization in product manufacturing. This work did not aim to provide a formal analysis of algorithmic complexity or performance metrics such as rendering time, memory usage, or scalability. Instead, the focus was placed on the conceptual adaptation of an existing algorithm to a new purpose, demonstrating the potential for algorithmic reuse in a different context. The significance of formal evaluation and user-centered validation is acknowledged, and future research will address these aspects through comprehensive performance testing and potential real-world deployment.

Acknowledgements

This work was supported by the Ministry of Science, Technological Development and Innovation of the Republic of Serbia [grant number 451-03-65/2024-03/200102].

Declaration on Generative AI

The authors have not employed any Generative AI tools.

References

[1] G.A. Pavlopoulos, T.G. Soldatos, A. Barbosa-Silva et al., A reference guide for tree analysis and visualization, BioData Mining 3 (2010) 1. https://doi.org/10.1186/1756-0381-3-1

[2] A. Shannon, C. Wetherell, Tidy drawings of trees, IEEE Transactions on Software Engineering 5 (1979) 514–520. https://doi.org/10.1109/TSE.1979.234212.

[3] E.M. Reingold, J.S. Tilford, Tidier drawings of trees, IEEE Transactions on Software Engineering SE-7 (1981) 223–228. https://doi.org/10.1109/TSE.1981.234519

[4] J.Q. Walker, A node-positioning algorithm for general trees, Software: Practice and Experience 20 (1990) 685–705. https://doi.org/10.1002/spe.4380200705

[5] C. Buchheim, M. Jünger, S. Leipert, Improving Walker's algorithm to run in linear time, in: Proceedings of the 10th International Symposium on Graph Drawing (GD 2002), volume 2528, 2002. https://doi.org/10.1007/3-540-36151-0_32

[6] M. Bruls, K. Huizing, J.J. van Wijk, Squarified treemaps, in: W.C. de Leeuw, R. van Liere (eds.), Data Visualization 2000, Eurographics, Springer, Vienna, 2000. https://doi.org/10.1007/978-3-7091-6783-0_4

[7] B. Johnson, B. Shneiderman, Treemaps: a space-filling approach to the visualization of hierarchical information structures, Proceeding Visualization '91, San Diego, CA, USA, 1991, 284–291. https://doi.org/10.1109/VISUAL.1991.175815

[8] G. Robertson, J. Mackinlay, S. Card, Cone trees: animated 3D visualizations of hierarchical information, Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, 1991, 189–194. https://doi.org/10.1145/108844.108883

[9] J. Carriere, R. Kazman, Research report. Interacting with huge hierarchies: beyond cone trees, Proceedings of Visualization 1995 Conference, Atlanta, GA, USA, 1995, 74–81. https://doi.org/10.1109/INFVIS.1995.528689

[10] S. Tee Teoh, M. Kwan-Liu, RINGS: a technique for visualizing large hierarchies, in: M.T. Goodrich, S.G. Kobourov (eds.), Graph Drawing, GD 2002, Lecture Notes in Computer Science, vol. 2528, Springer, Berlin, Heidelberg, 2002. https://doi.org/10.1007/3-540-36151-0_25

[11] S. Grivet, D. Auber, J.-P. Domenger, G. Melançon, Bubble tree drawing algorithm, International Conference on Computer Vision and Graphics, 2006, 633–641. https://doi.org/10.1007/1-4020-4179-9_91

[12] Q.V. Nguyen, M.L. Huang, Space-optimized tree: a connection+enclosure approach for the visualization of large hierarchies, Information Visualization 2 (2003) 3–15. https://doi.org/10.1057/palgrave.ivs.9500031

[13] D. Aleksic, D. Janković, L. Stoimenov, A case study on the object-oriented framework for modeling product families with the dominant variation of the topology in the one-of-a-kind production, International Journal of Advanced Manufacturing Technology 59 (2012). https://doi.org/10.1007/s00170-011-3466-4

[14] D.S. Aleksic, D.S. Jankovic, P. Rajkovic, Product configurators in SME one-of-a-kind production with the dominant variation of the topology in a hybrid manufacturing cloud, International Journal of Advanced Manufacturing Technology 92 (2017) 2145–2167. https://doi.org/10.1007/s00170-017-0286-1

[15] R. Lim, Algorithm for Drawing Trees, Rachel's Blog, URL: https://rachel53461.wordpress.com/2014/04/20/algorithm-for-drawing-trees/

[16] D. Condit, Orgchart, Github repository, URL: https://github.com/DavidCondit/orgchart

[17] J. Lamping, R. Rao, The Hyperbolic Browser: A Focus+Context Technique for Visualizing Large Hierarchies, J. Vis. Lang. & Comput. 7.1 (1996) 33–55. doi:10.1006/jvlc.1996.0003.

[18] I. Herman, G. Melançon, M. M. de Ruiter, M. Delest, Latour — A Tree Visualisation System, in: Graph Drawing, Springer Berlin Heidelberg, Berlin, Heidelberg, 1999, pp. 392–399. doi:10.1007/3-540-46648-7_40.

[19] I. G. Tollis, G. D. Battista, P. Eades, R. Tamassia, Graph Drawing: Algorithms for the Visualization of Graphs, Prentice Hall, 1998.

[20] E. H. Chi, J. Pitkow, J. Mackinlay, P. Pirolli, R. Gossweiler, S. K. Card, Visualizing the evolution of Web ecologies, in: the SIGCHI conference, ACM Press, New York, New York, USA, 1998. doi:10.1145/274644.274699.

[21] E. Kleiberg, H. van de Wetering, J. J. van Wijk, Botanical visualization of huge hierarchies, in: IEEE Symposium on Information Visualization, 2001. INFOVIS 2001., IEEE. doi:10.1109/infvis.2001.963285.

[22] G. J. Wills, NicheWorks: Interactive Visualization of Very Large Graphs, J. Comput. Graph. Stat. 8.2 (1999) 190. doi:10.2307/1390633.

[23] F. Ruskey, M. Weston, Venn Diagrams, Electron. J. Comb. 1000 (2005). doi:10.37236/26.

[24] M. Dumas, A. H. M. ter Hofstede, UML Activity Diagrams as a Workflow Specification Language, in: ≪UML≫ 2001 — The Unified Modeling Language. Modeling Languages, Concepts, and Tools, Springer Berlin Heidelberg, Berlin, Heidelberg, 2001, pp. 76–90. doi:10.1007/3-540-45441-1_7.

[25] M. Chinosi, A. Trombetta, BPMN: An introduction to the standard, Comput. Stand. & Interfaces 34.1 (2012) 124–134. doi:10.1016/j.csi.2011.06.002.

[26] B. Leasure, D. J. Kuck, S. Gorlatch, M. Cole, G. R. Watson, A. Darte, D. Padua, U. Banerjee, O. Schenk, K. Gärtner, et al., Petri Nets, in: Encyclopedia of Parallel Computing, Springer US, Boston, MA, 2011, pp. 1525–1530. doi:10.1007/978-0-387-09766-4_134.