An Investigation of Tree Layout Algorithms

David Cohen, Josh Goodsell

Introduction to Visualization
Rensselaer Polytechnic Institute

1. Introduction

Automated theorem proving is an enormous field containing many different methods and systems. One method used in this field is to produce what is called a semantic tableaux or a truth tree from a set of logical sentences. Finding a way to present these trees as a result can be rather difficult due to their size and complexity. In this paper we investigate a few algorithms for laying out the nodes of a tree in order to produce understandable results.

2. Algorithms

Our work centered on the design of three different algorithms and interfacing with a third party graph layout library. The main goal in these algorithms was to improve the readability and usability of the original simple algorithm. To allow for easy switching between algorithms each algorithm was implemented in a class deriving from one super class which defined an abstract Layout() method.

2.1 Simple

The initial algorithm investigated was the original layout method which we named the "simple" layout. This algorithm is recursive where each node will call Layout() on its children and then place the children relative to itself. The children are placed sequentially from left to right such that each child's bounding box does not overlap the child to the left. The parent would then place itself in the middle of the horizontal axis of its bounding box.

```
Layout(tree) :
  left = 0
  foreach child in tree.Children :
    Layout(child)
    child.Left = left
    child.Top = tree.Bottom
    left += child.Width
```

Pseudocode for the simple algorithm

2.2 Constructive Solid Geometry (CSG)

The second algorithm is an evolution of the simple algorithm. Bounding polygons were used instead of bounding boxes which would allow for much greater accuracy. The algorithm would keep track of the polygons for the children that had been placed already and instead of placing each child sequentially they would be placed at the leftmost edge and moved right until their bounding polygons no longer intersected any of the placed polygons. A free, open source CSG library named ClipperLib [1] was used

to implement the intersection and union operations. The parent would be placed in middle of its children with the expectation that this would produce cleaner lines stemming from parent to child.

```
Layout(tree) :
  placed = empty polygon
  foreach child in tree.Children :
    Layout(child)
    child.Left = 0
    child.Top = tree.Bottom
    while child.Polygons intersects placed :
      child.Left += width of largest overlapping area
    placed = child.Polygons union placed
```

Pseudocode for the CSG algorithm

2.3 Overlapping Pair

The third algorithm works in a similar manner to the first two where it attempts to resolve overlapping nodes. Initially every node is placed in the same location. Nodes are then taken in pairs and tested for intersection. If it is found that the two are overlapping one of them is shifted according to some simple rules. If the two nodes are at different depths in the tree, that is how many nodes are between it and the root, then the deeper node will be shifted downwards by the amount they were overlapping in the vertical axis plus a little extra to ensure a gap between the two. If the depths are the same then the node that is farthest to the right will be shifted further to the right by the amount they were overlapping in the horizontal axis plus some extra. In an effort to keep the parents centered above the children when a node is shifted horizontally it will also shift its parent horizontally by some fraction of the original shift amount.

```
Layout(tree) :
  nodes = list of all the nodes in tree
  foreach node in nodes :
    node.Left = 0
    node.Top = 0
  while overlappingPairExists() :
    nodea, nodeb, overlap = getOverlappingPair()
    if nodea.Depth > nodeb.Depth :
      shiftTree(nodea, 0, overlap.Height)
    else if nodea.Depth < nodeb.Depth :
      shiftTree(nodeb, 0, overlap.Height)
    else :
      if nodea.Left < nodeb.Left :
        shiftTree(nodeb, overlap.Width, 0)
      else :
        shiftTree(nodea, overlap.Width, 0)
```

Pseudocode for the overlapping pair algorithm

2.4 Spring

The final method implemented was also the most radically different. Instead of algorithmically placing all of the nodes the tree was repurposed as a mass and spring system. Each node in the tree was equivalent to a mass and springs were placed between parent and children nodes. A repulsive force was added between masses that was dependant on their sizes as a node as well as gravity to force the tree

to unfold downwards. The system was then run until reaching some level of equilibrium and the mass's locations were transferred to the appropriate nodes.

2.5 Graph#

The first thing we did when starting this project was to search online to see if there were any good methods readily available. When searching for ones that were specifically made for C# we kept seeing the name Graph# [2] along with much praise for this library. We decided that it would be interesting to see how our methods would compare to those present in such a library. One problem that was never mentioned alongside the praise was that there was in fact no documentation at all for the library. This resulted in a lot of trial and error as well as digging through the object browser trying to find a class that did what we wanted. Eventually we got one of the many layout algorithms in the library working, the SimpleTreeLayoutAlgorithm. After doing some tests to see if this was the algorithm we were looking for we noticed that there would be the occasional overlapping node. Luckily while exploring the library we had seen a package named OverlapRemoval which contained a couple of algorithms to remove such overlaps. The way this method ended up working was to run the SimpleTreeLayoutAlgorithm and then copy the results to the FSAAlgorithm, which was one of the two overlap removal algorithms, and run that. The results from the overlap removal were then copied back into the tree structure.

3. Program

After receiving some of our feedback we decided to create an interactive program that allowed users to create trees and switch between different layout algorithms. The program give the options of add, remove, and resize, which all work on the node you click on. Resize requires a width and height and the parameters are entered just below its radio button. A random button allows the user to make the program generate a tree for them. A save button allows the user to save the picture of the current graph. Lastly a dropdown menu allows the user to change which algorithm is being used to display the tree.

3.1 Feedback

The feedback is broken down into two sections due to the fact that what was presented to those in class and those out of class was different.

3.1.1 In Class

During class, we showed some static images comparing the two initial layout algorithms, simple and CSG, which received mixed reviews. The positive feedback was that the CSG algorithm at that point was a good starting point. The negative feedback received was that there should be more colors available for displaying the tree and that sometimes it may be better to not center children below the parent if there is only one child. For the short term it was suggested that we add some sort of evaluation function so that we could quantitatively show that one algorithm was better than the rest. For the long term we were told to add interactivity.

3.1.2 Out of Class

Outside of class the people were allowed to play with the program itself and were able to spend much more time than those in class. Overall they felt the UI was very user-friendly, with the options being very understandable without being overwhelming. The color was liked as well. The CSG was generally seen as the best and Graph# being the worst. Connecting lines were seen at times to overlap and this was seen

to be confusing for some. We were asked to add more interactivity, add more colors, add limitations to the amount of branches, and fix Graph#.

4. Results

Due to the size of many of the figures, all figures except Figure 1 and 2 have been placed at the end of the paper. In order to decide which algorithm truly was best two scoring functions were used. One for white space and one for line slope, with more weight in the white space score. Considering look is perspective, these were two good tests for look because they were intuitive subjects that people suggested changing. One of the other request we got was to not allow connection lines passed through nodes. The white space score was best way to compare how dense a given tree graph is. The score is between 1 and 0, 1 being perfect and only resulting when the root node is the only node present. In figure 1, A white space score of above 1 can be seen. This is caused by overlapping nodes. The slope score is measured 0 to infinity, 0 being a score resulting from only vertical lines and infinity resulting from horizontal lines. In figure 2, all connections are vertical, resulting in a slope score of 0. The slope score is a sum of all lines and lower scores are preferred but 4 and 5 can be considered similar enough to be equivalent depending on the amount of connections.

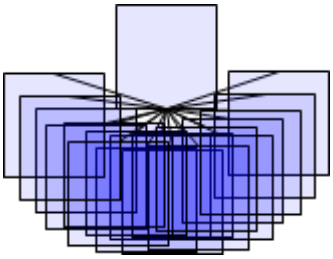

Figure 1. Better than perfect whitespace score

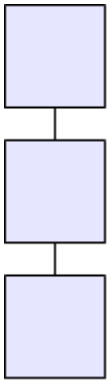

Figure 2. Perfect line score

The algorithms produced similar results with minor differences most of the time. Many times Spring would fail or produce overlap (Figures 1 and 12). The results of three test trees can be seen in Table 1 below.

Tree 1			
Algorithm	White Space Score	Slope Score	Figure
Simple	0.17223	5.05882	3
CSG	0.28872	2.82353	4
Graph#	0.28765	3.11765	5
Overlapping Pair	0.24167	3.44118	6
Spring	N/A	N/A	N/A
Tree 2			
Algorithm	White Space Score	Slope Score	Figure
Simple	0.23446	6.29630	7
CSG	0.45290	3.55556	8
Graph#	0.25395	4.92593	9
Overlapping Pair	0.30955	5.33333	10
Spring	1.12123	1.11111	11
Tree 3			
Algorithm	White Space Score	Slope Score	Figure
Simple	0.30278	5.95652	12
CSG	0.44035	5.37681	13
Graph#	0.21180	3.52174	14
Overlapping Pair	0.27139	6.04348	15
Spring	N/A	N/A	N/A

Table 1. Results

With the white space score closest to 1 in all cases and a reasonable slope score in all cases the algorithm CSG came out to be the best of our algorithms. Tree 3 is an example tree of a realistic result of what the algorithms were being made for.

4.1 Issues

Although we had hoped to avoid this we found that all of the algorithms except the simple one occasionally produced layouts that had lines crossing other lines or crossing nodes. This was likely due to the fact that none of the algorithms were designed with this in mind and generally ignore the fact that there are lines connecting the nodes. Another large issue we had was that the overlapping pair algorithm could enter into an infinite loop if the tree were constructed with a certain structure due to the way overlapping pairs were found. Along similar lines the spring system was incredibly unstable as can be seen by the fact that it did not produce results for two of our three test cases. In many cases the simulation did not reach equilibrium which may have been solved by changing some of the parameters in the system. In other cases the simulation would end immediately after a single step due to the amount of energy overflowing and looping around to less than the maximum required for equilibrium. We could not find any pattern in the trees that produced this issue. This method would likely be much better if the parameters were adjusted for each individual tree. The final major issue we had was that the Graph# method didn't always place nodes in an understandable location and the addition of overlap removal only made this more noticeable. It may be the case that we were simply missing something that the designers of the library expected us to do or we could be doing something wrong.

5. Future Work

In the future many features could be added. Some possible features are right click, hotkey, color, and zoom functionality for the program. For right click functionality, what could be done is combine the Add and Remove radio buttons into one and use left click for add and right click for remove. For hotkey functionality, what could be done is bind 1, 2, 3, and on to the layout algorithms and A, D, R to the functions add, remove/delete, and resize. For color functionality, not only could the color be changed to another color, but also individual nodes could be changed to another color. Beyond that color presets could exist. Also, connection lines crossing over nodes could be fixed. For zoom functionality, zooming in would be good for small trees and zooming out to see the whole tree would be good for large trees. Another feature that was mentioned to us was to have a mass resize that would resize all nodes in the tree.

6. References

[1] ClipperLib, Angus Johnson, 29 October 2011, < http://www.angusj.com/delphi/clipper.php >
[2] Graph#, 1 June 2009, < http://graphsharp.codeplex.com >

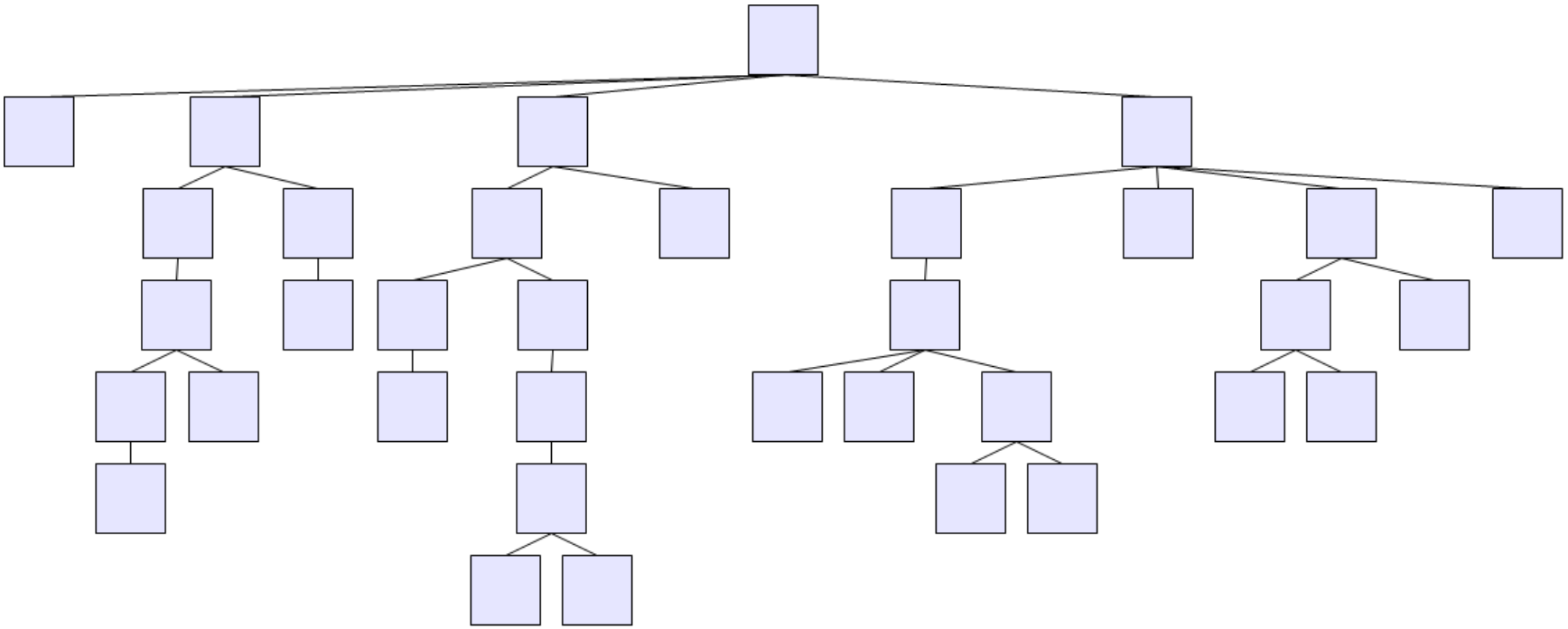

Figure 3. Tree 1 with the simple algorithm

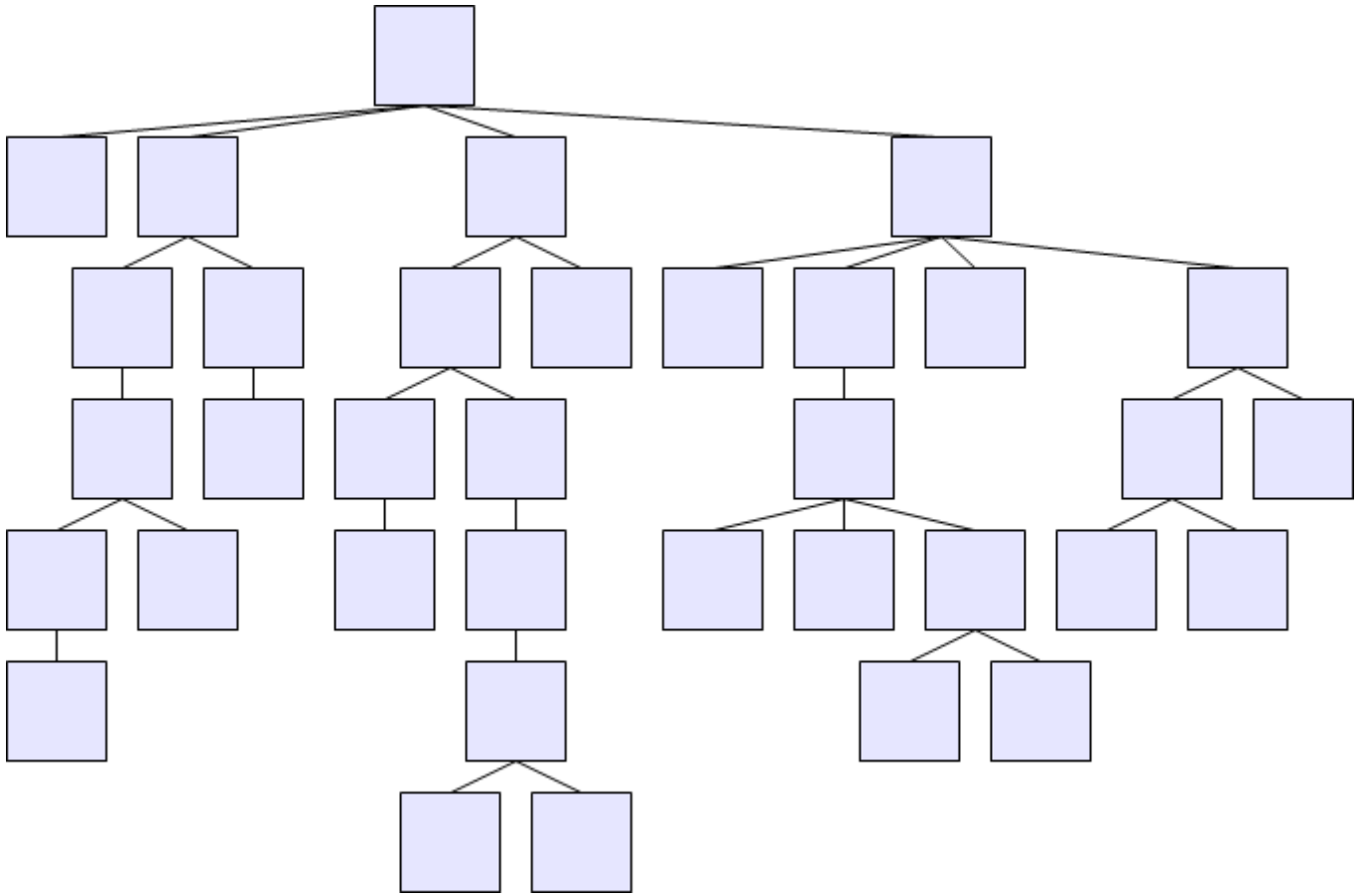

Figure 4. Tree 1 with the CSG algorithm

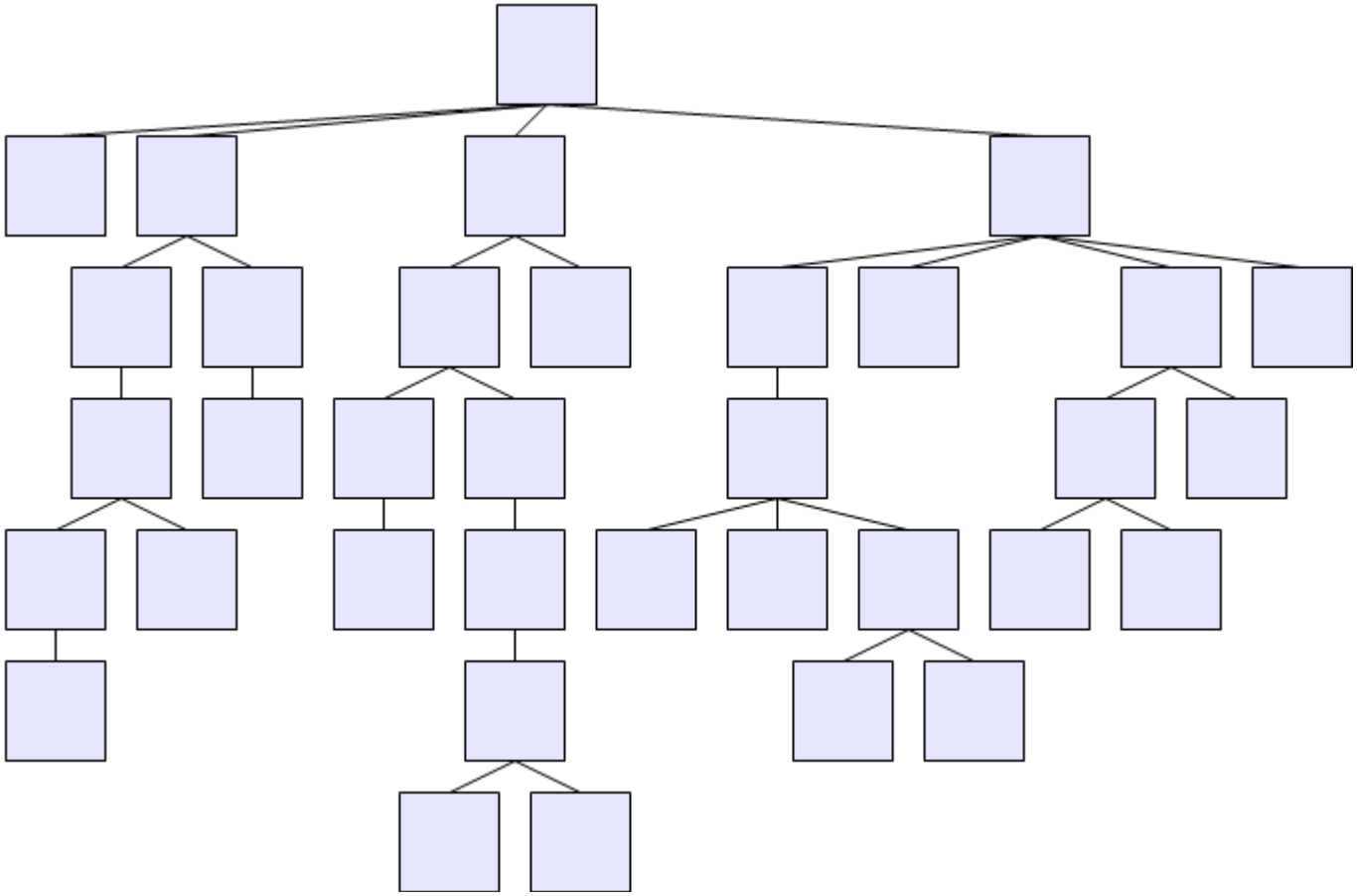

Figure 5. Tree 1 with the Graph# algorithm

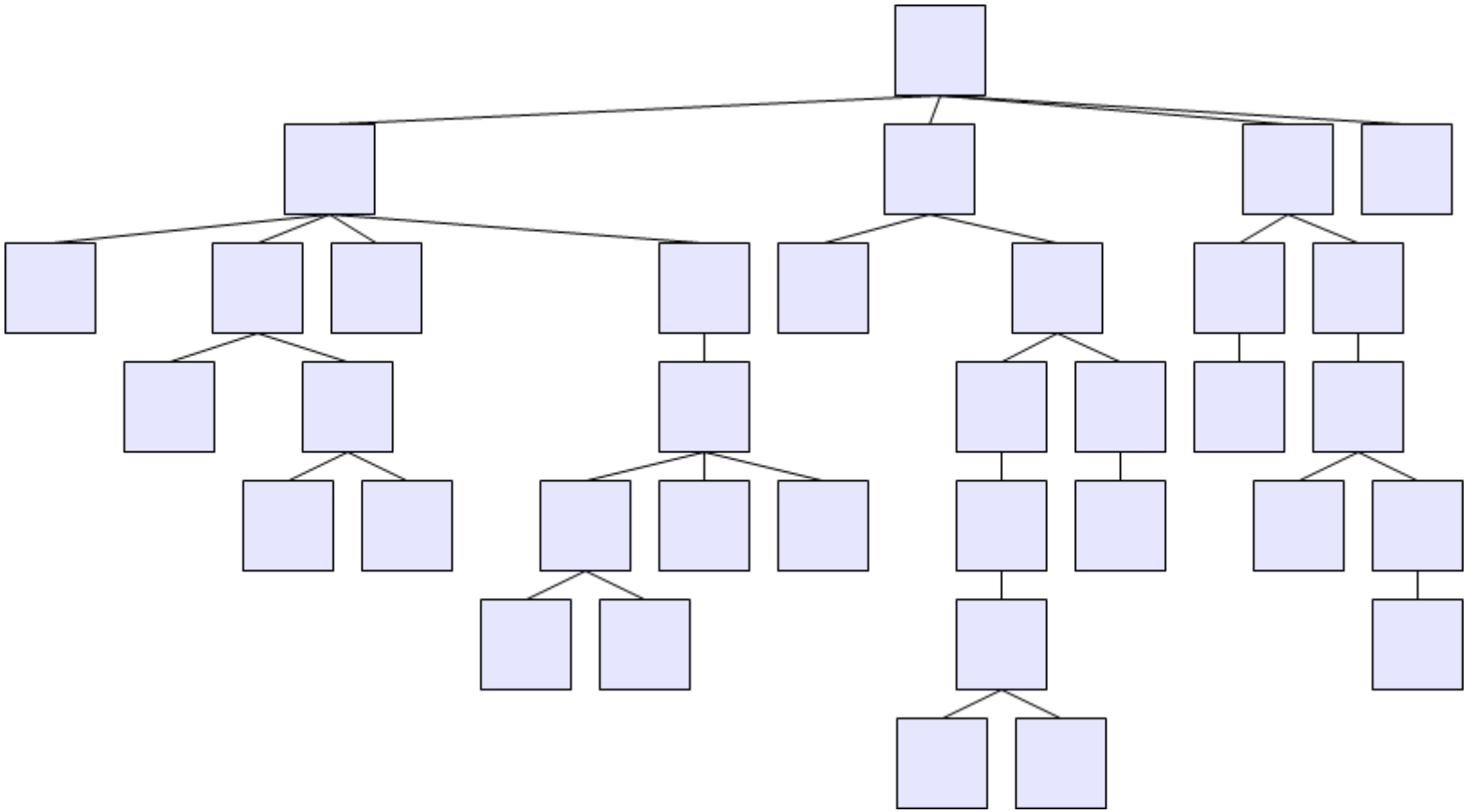

Figure 6. Tree 1 with the overlapping pair algorithm

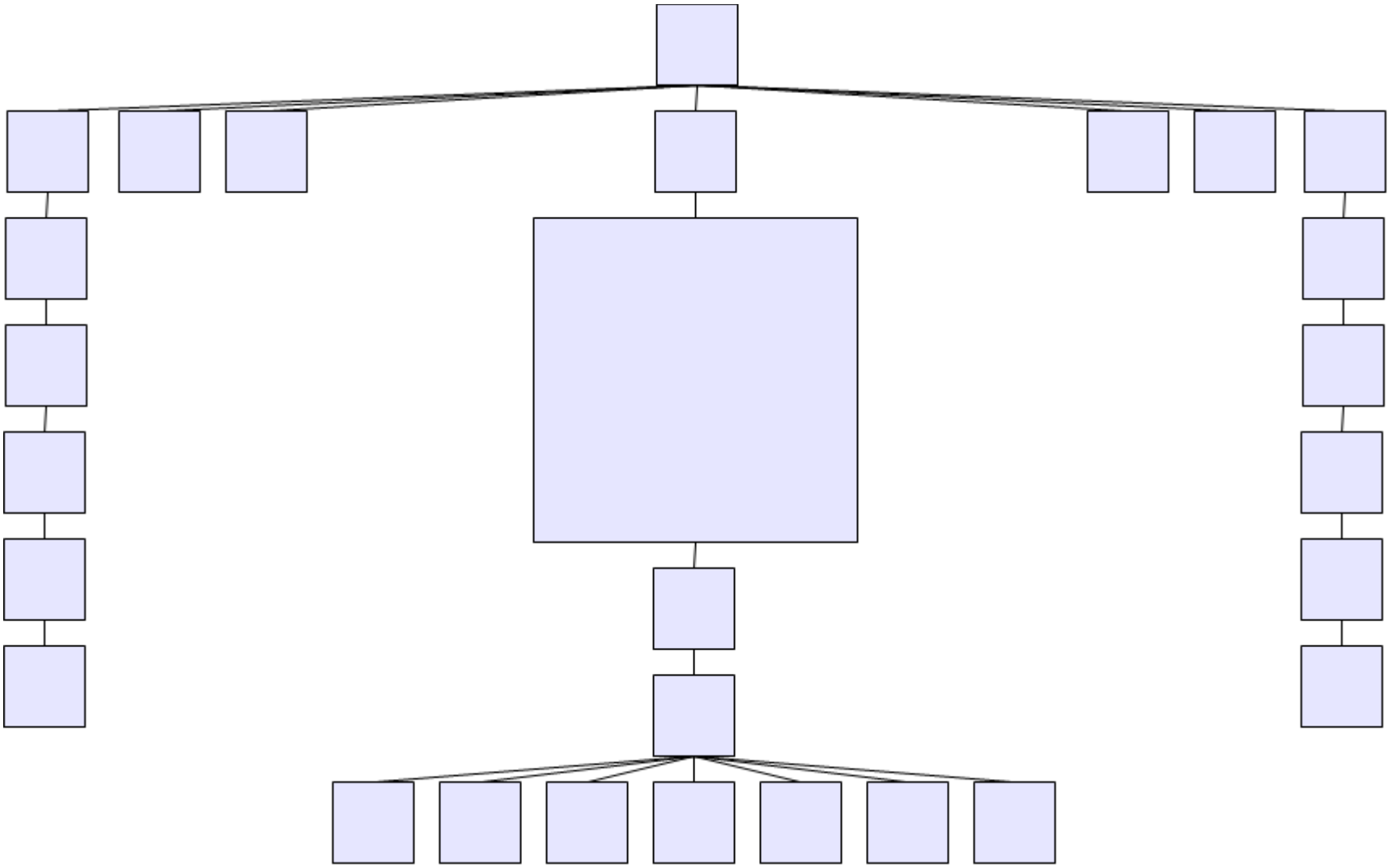

Figure 7. Tree 2 with the simple algorithm

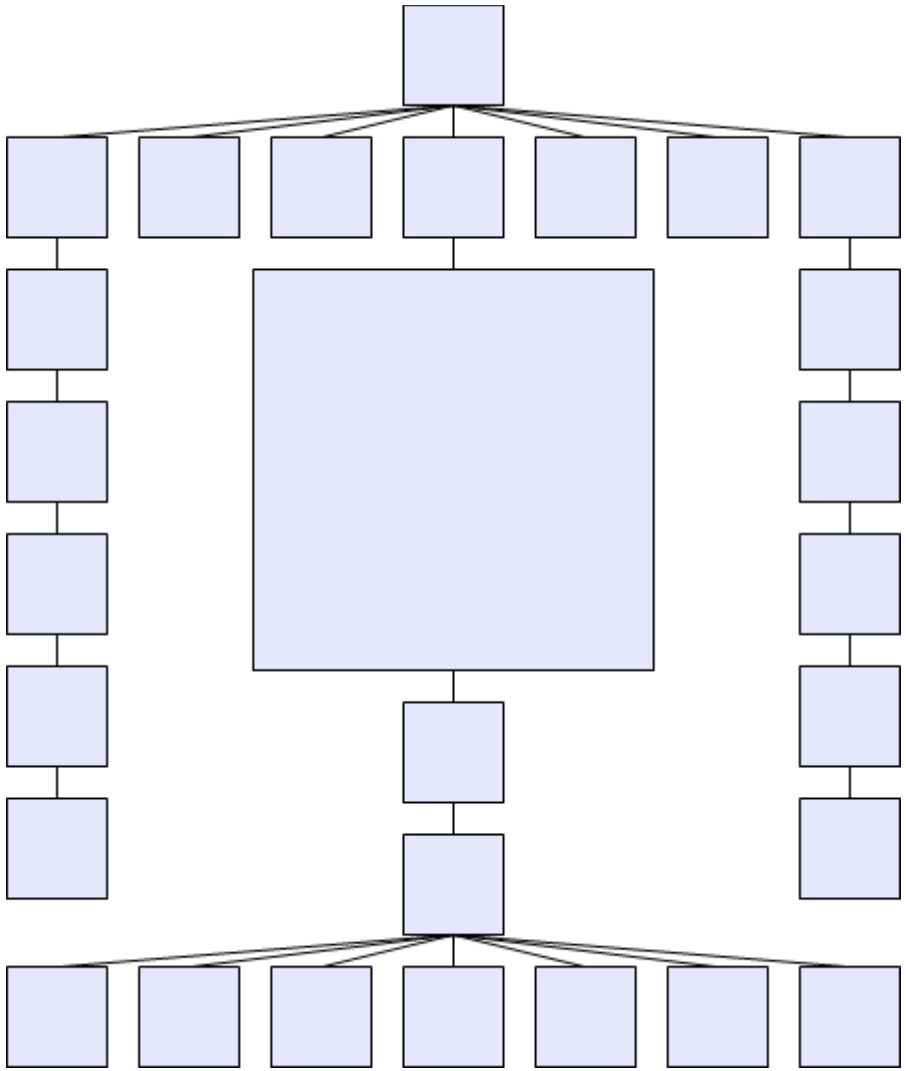

Figure 8. Tree 2 with the CSG algorithm

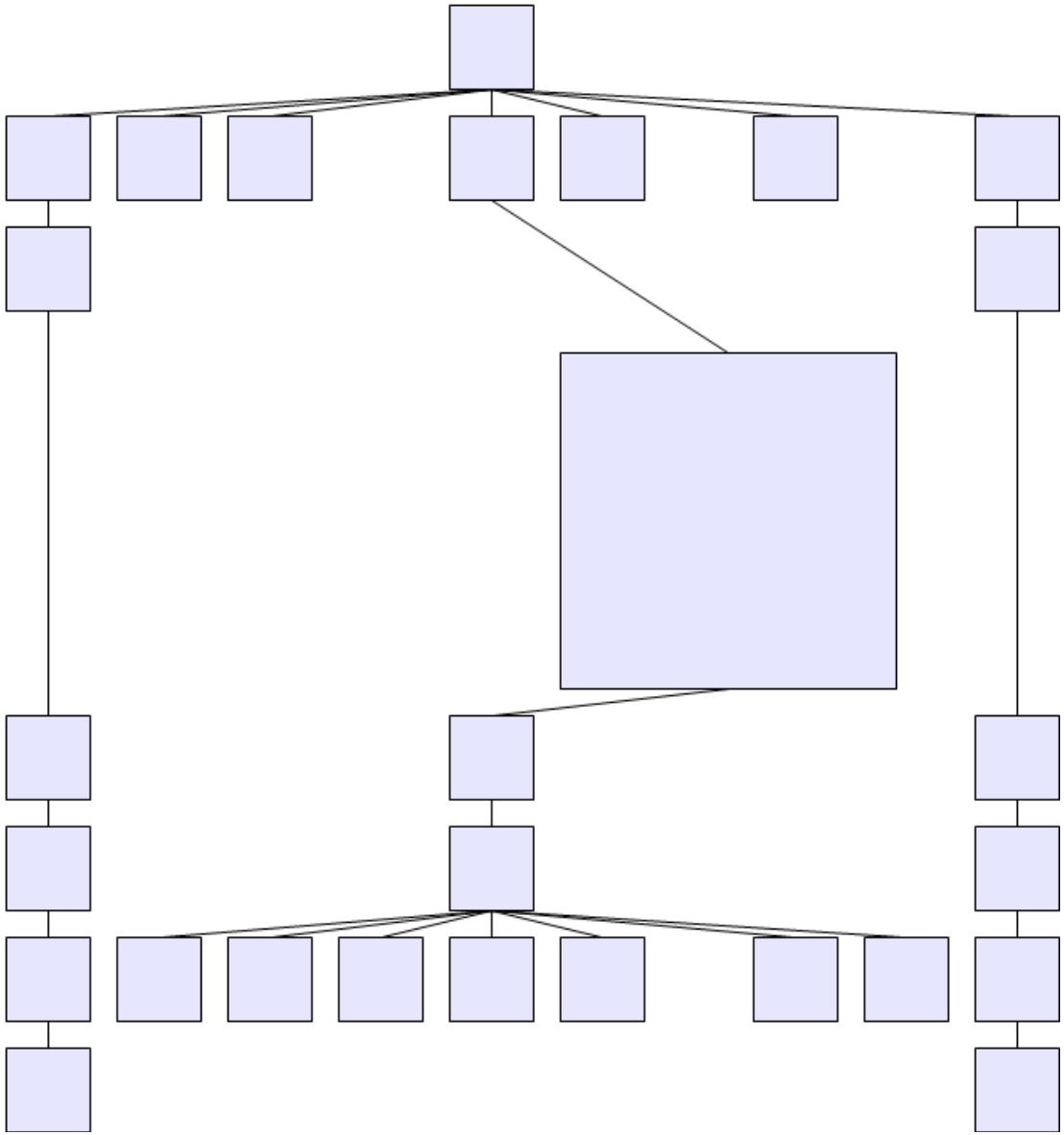

Figure 9. Tree 2 with the Graph# algorithm

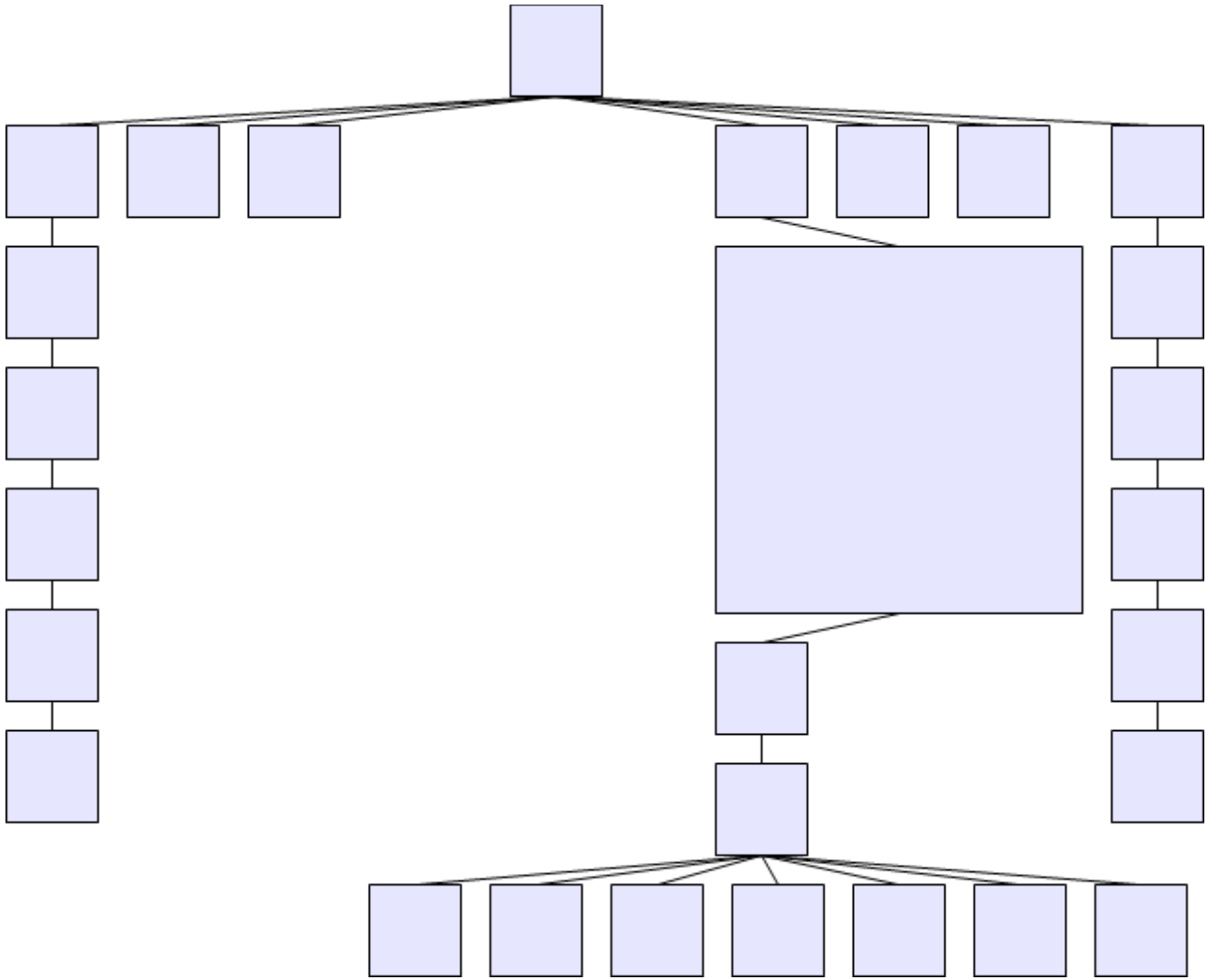

Figure 10. Tree 2 with the overlapping pair algorithm

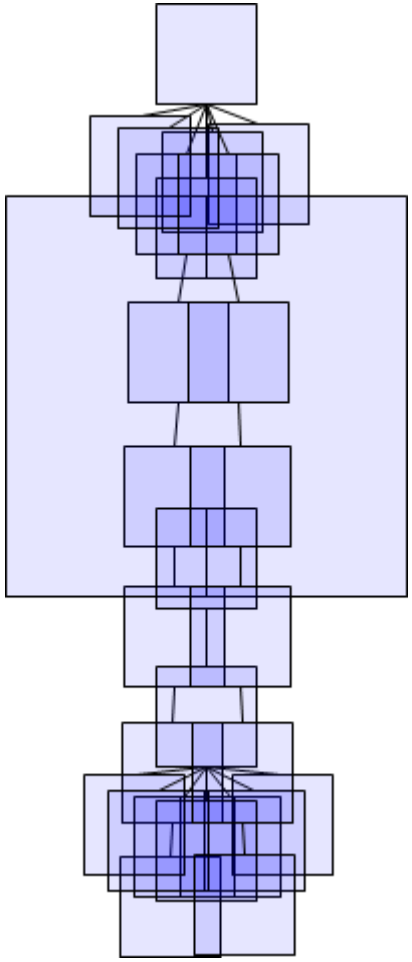

Figure 11. Tree 2 with the spring algorithm

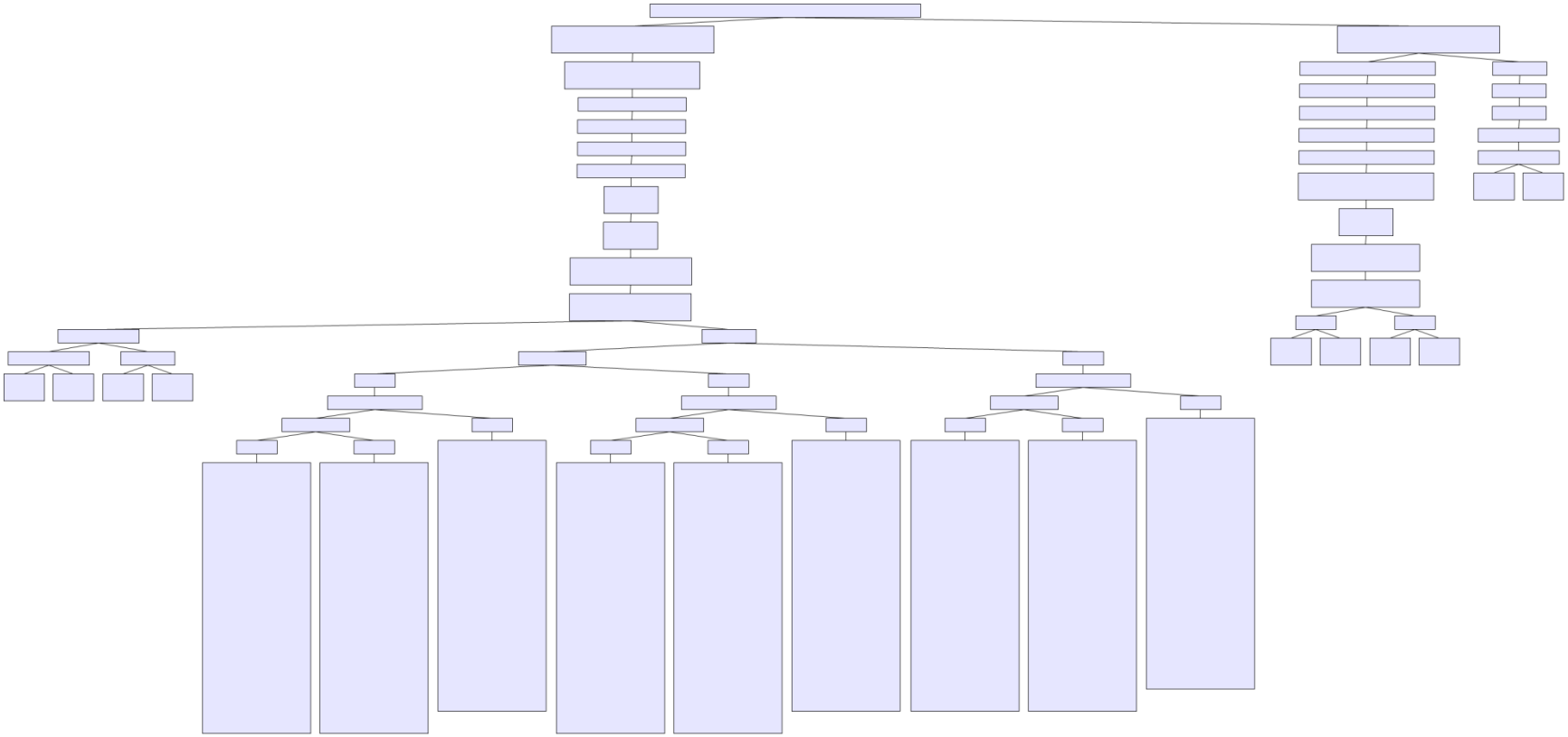

Figure 12. Tree 3 with the simple algorithm

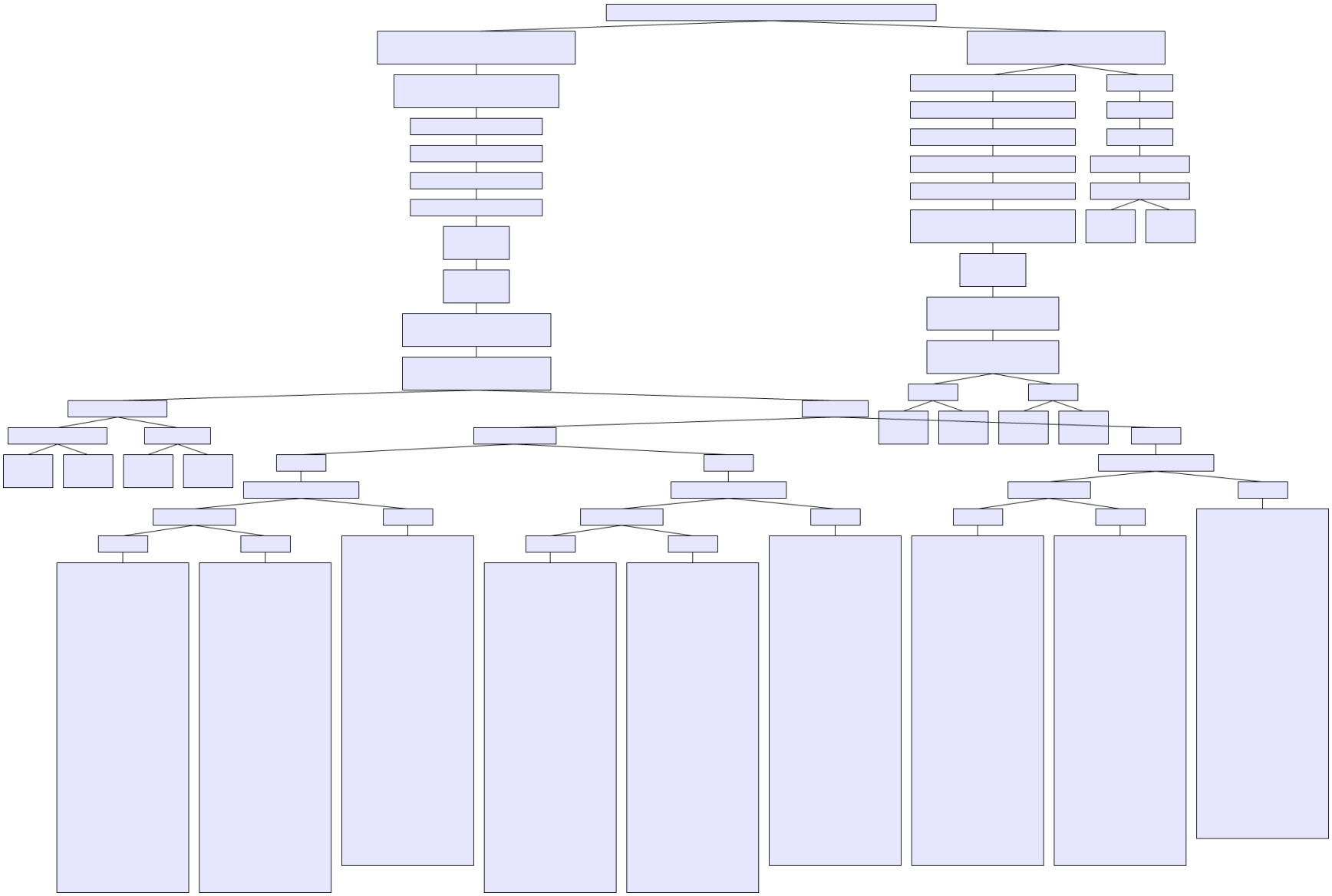

Figure 13. Tree 3 with the CSG algorithm

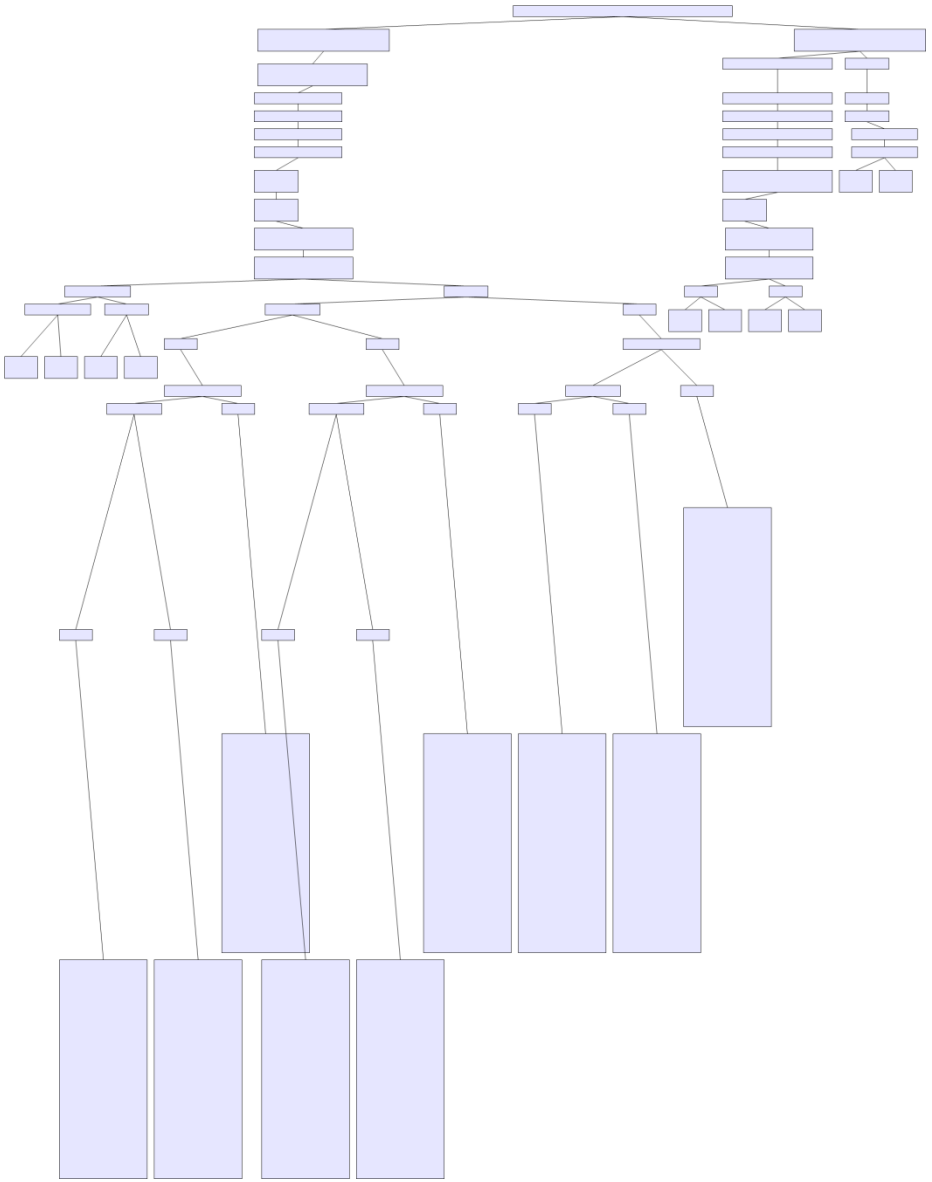

Figure 14. Tree 3 with the Graph# algorithm

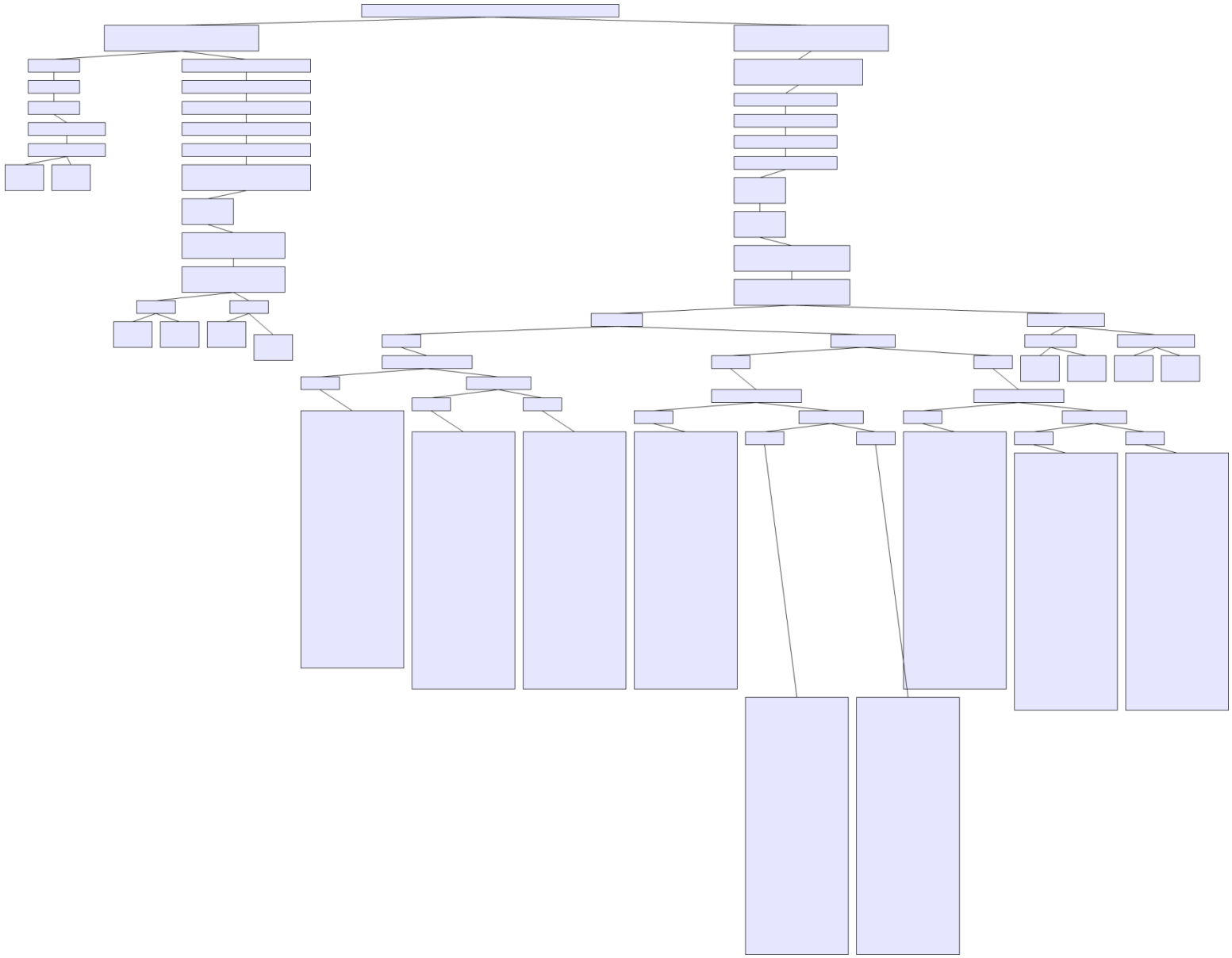

Figure 15. Tree 3 with the overlapping pair algorithm