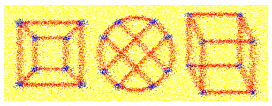

Journal of Graph Algorithms and Applications
http://jgaa.info/ vol. 21, no. 6, pp. 1027–1038 (2017)
DOI: 10.7155/jgaa.00448

Root demotion: efficient post-processing of layered graphs to reduce dummy vertices for hierarchical graph drawing

Daniel Sumner Magruder[1,3] *Stefan Bonn*[1,2]

[1]Institute of Medical Systems Biology
Center for Molecular Neurobiology
University Clinic
Hamburg-Eppendorf, Germany
[2]German Center for Neurodegenerative Diseases
Gttingen, Germany
[3]Genevention GmbH
Gttingen, Germany

Abstract

One of the most popular hierarchical graph drawing frameworks - the Sugiyama, Tagawa, and Toda (STT) framework - often introduces dummy vertices to the given directed acyclic graph as part of its methodology to produce the final drawing. The inclusion of dummy vertices increases the size of the input graph and consequently inflates the run time of the subsequent algorithms. Given that most graph drawing problems are NP-hard, good layering algorithms or efficient post-processing - which produces a minimal number of dummy vertices - restrain the increased run time. Many layering or layering post-processing algorithms have quadratic or cubic run time. Here we introduce Root Demotion, a linear time post-processing algorithm for the layering produced by the Longest-Path algorithm that reduces the number of dummy vertices required compared to current post-processing algorithms, consequently shortening the remaining run time of the STT framework.

Submitted:	Reviewed:	Revised:	Accepted:	Final:
March 2017	June 2017	July 2017	August 2017	September 2017
		Published: October 2017		
	Article type: Concise Paper		Communicated by: G. Liotta	

E-mail addresses: sumner.magruder@zmnh.uni-hamburg.de (Daniel Sumner Magruder) sbonn@uke.de (Stefan Bonn)

1 Introduction

Graphs are often the data type of choice for visualizing complex relationships. The automatic drawing of graphs, however, is a non-trivial matter, and many graph drawing sub-problems are classified as NP-hard [7].

Hierarchical Graph Drawing (HGD), a popular style to present directed graphs which represent hierarchies or processes (e.g. phylogeny or gene interaction networks), places the vertices of the graph in parallel layers with the aim that the arcs of the graph are oriented in the same direction. The methods for HGD were formalized in 1981 by Sugiyama, Tagawa, and Toda in what is now referred to as the STT framework [11, 7, 1, 3]. The STT framework, in general, can be describe by the following steps:

1. convert the input graph to a directed acyclic graph (DAG)
2. produce a layering by partitioning the vertices of the DAG into layers
3. reduce the number of edge crossings by reordering the vertices in the layers of the layering
4. assign vertical and horizontal coordinates to the vertices

As the layering produced in step two of the STT framework is used in the subsequent steps, it affects the run time for the remainder of the STT framework. In addition, the layering has an influence on the aesthetics of the resultant drawing (e.g. by influencing the height and width of the graph).

The layering's effect on run time can be easily identified in the step of edge crossing minimization, one of the many NP-hard sub-problems. Many of the edge crossing minimization approaches aim to reduce edge crossings locally (e.g. between two adjacent layers at a time) to approximate the optimal solution [11, 7, 2, 12]. Prior to local edge crossing minimization approaches, the STT framework introduces placeholders, known as dummy vertices, to account for the placement of long edges - edges which cross through the local layers undergoing reordering but may not start or end there.

Since dummy vertices increase the size of the graph, naturally the run time for drawing these graphs also increases; the increased run time is a deterrent for using HGD on large graphs. Thus, although necessary, it is ideal to find a layering which requires the minimum number of dummy vertices (see, e.g. Figure 1).

2 Related work

Given the effects of dummy vertices on HGD, several layering algorithms [5, 6] have been developed which produce the minimum number of dummy vertices, such as the Network-Simplex Layering Algorithm by Gansner et al. that utilizes integer linear programming (ILP) which runs, putatively, in polynomial time [7, 5, 6]. Despite these minimal-dummy vertex producing algorithms' existence, the

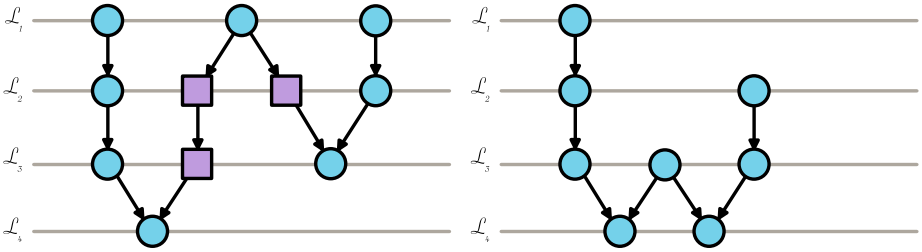


Figure 1: Two alternative layerings of a DAG rendered hierarchically. Long edges have been segmented by introducing dummy vertices, thereby increasing the size of the DAG. By changing the layering of the vertices the number of dummy vertices needed can alter. The layering in (a) requires three dummy vertices, while the layering in (b) requires none. Dummy vertices are represented by purple squares.

Longest-Path algorithm for layering (Longest-Path layering) remains a widely popular layering technique given: 1.) its simplicity and 2.) its linear time complexity [9, 7, 10]. Therefore it may be competitive to post-process the layering produced by the Longest-Path algorithm with the goal of achieving the minimum (or an approximation thereof) number of dummy vertices.

The post-processing approach of handling dummy vertices manifested in 2006 when Nikolov and Tarassov unveiled Promote Layering [10]. Promote Layering aimed to reduce the number of dummy vertices by traversing against the flow of arcs, moving the current vertex up a layer - promotion - and comparing whether or not such a move reduced the number of dummy vertices needed; however, this algorithm could *increase* the number of layers in the layering [10]. While performing relatively well for small (≤ 100 vertices) graphs, Nikolov and Tarassov found that Promote Layering was noncompetitive on simulated graphs of larger size leading to a restriction on their repeat-until statement [10]. Naturally a linear time approximation of a layout with the minimum number of dummy vertices would be ideal. Here we present Root Demotion as an easy to implement, linear post-processing algorithm, which improves the reduction of the number of dummy vertices required compared to other post-processing reduction algorithms.

3 Root Demotion

3.1 Notation

Let $G(V, A)$ be the DAG with vertex set $V = \{v_1, v_2, \ldots, v_{|V|}\}$ and arc set $A = \{a_1, a_2, \ldots, a_{|A|}\}$, where an arc $a = (u, v) : \ u, v \in V$ is defined as an ordered set representing the directed edge from u to v. Given arc (u, v) we say that the source of the arc, u, is an immediate predecessor of v and the target

of the arc, v, is an immediate successor of u. Let $N^+(v) = \{s : (v, s) \in A\}$ be the set of all immediate successors of v. Similarly let the set of all immediate predecessors of v be defined as $N^-(v) = \{p : (p, v) \in A\}$. In addition, let $\mathscr{L} = \mathscr{L}_1 \prec \mathscr{L}_2 \prec \cdots \prec \mathscr{L}_h \quad : h \geq 1$ be a layering of G, where $\mathscr{L}_i$ is "higher" than $\mathscr{L}_j$ if $i < j$ (likewise $\mathscr{L}_j$ is "lower" than $\mathscr{L}_i$). For convenience, let $\mathscr{L}(v) = i$, where i is the index of the layer, $\mathscr{L}_i$, in which v resides. Then the layered DAG, $G(V, A, \mathscr{L})$, allows the span of an arc $a = (u, v)$ to be defined as $\text{span}(a) = \mathscr{L}(v) - \mathscr{L}(u)$; a layered DAG is proper should $\text{span}(a) = 1 \quad \forall a \in A$ (hereafter referred to as proper-layered graph).

Most graph drawing algorithms following the `STT` framework, with the notable exception of Eiglsperger, Siebenhaller and Kaufmann [4], require the transformation of the arc set of an improper-layered DAG into a proper-layered DAG; such a transformation of an improper arc, $\mathscr{T}$, often involves the injection of dummy vertices into arcs with span greater than 1. For example, if $a = (u, v)$ and $\text{span}(a) > 1$ then

$$\mathscr{T}(a) = (u, d_1), (d_1, d_2), \cdots, (d_n, v) \quad n = \text{span}(a) - 1 \tag{1}$$

such that $\mathscr{L}(d_i) = \mathscr{L}(u) + i$. Consequently, the set of all dummy nodes, D, must be introduced, where

$$|D| = \sum_{a \in A} \text{span}(a) - |A| \tag{2}$$

3.2 Approach

The Longest-Path algorithm produces a layering with minimum height in linear time, but it forgoes optimal performance in regards to minimizing the number of dummy vertices, edge density and drawing area [7, 10]. Thus, Root Demotion aims to be a linear time algorithm which reduces the number of dummy vertices required by the Longest-Path algorithm by changing the layers to which the vertices are assigned if said layer is non-optimal. A vertex, v, is in a non-optimal layer if for every arc incident to v the span of the arc is greater than one i.e. $\text{span}(a) > 1 \quad \forall a$ s.t. $v \in a \wedge a \in A$.

The Longest-Path algorithm layers the vertices such that a successor, s, of a given vertex, v, will be a lower layer than the source vertex i.e. $\mathscr{L}(s) > \mathscr{L}(v)$. Thus if the layering is traversed layer by layer, for example from the lowest layer ($\mathscr{L}_h$) to highest layer ($\mathscr{L}_1$), then the only arcs which need to be considered are those against the flow of traversal (in this case the arcs $(v, s) \in A \forall s \in V$), as the predecessors of v will be handled when their layer is visited. Using such a one-sided approach makes it easy to improve the layer of a vertex in a non-optimal layer: simply move the vertex down (demote) to the layer above its nearest successor's layer ($\mathscr{L}(v) = \mathscr{L}(s) - 1$). Traversing layer by layer will likely result in vertices of higher layers being "dragged" down the layering (which can include roots of the DAG if there is more than one).

When moving vertices to other layers, it is important to ensure these changes do not introduce intra-layer edges (layering-preservation), as noted by Nikolov

and Tarassov (2006), as many HGD algorithms assume these edges will not be present in the layering [10]. An intra-layer edge could be introduced if a vertex is moved into the same layer as one of its predecessors or successors. Since Root Demotion ensures that when a vertex is moved the vertex is moved exclusively to be one layer away from its nearest successor, Root Demotion is layering-preserving.

Algorithm 1 Root Demotion

Require: A DAG $G = (V, A)$ with layering $\mathscr{L}$

```
function ROOT_DEMOTION(G, 𝓛)                    ▷ reduce dummy vertices needed
    vertices ← ⋃_{i}^{h} 𝓛_i                    ▷ vertices is a stack formed by the ordered union of the layers in the layering 𝓛, where h ← |𝓛|
    while vertices ≠ ∅ do
        v ← POP(vertices)                       ▷ visit next vertex in lowest layer

        if N⁺(v) ≠ ∅ then
            highest_layer ←MIN({𝓛(s) : s ∈ N⁺(v)})
                                                ▷ if v has at least one successor, then find closest layer with a successor
            if highest_layer − 𝓛(v) > 1 then
                𝓛(v) ← highest_layer − 1        ▷ if highest_layer is more than one layer away from v, then demote v to one layer above the layer of its nearest successor
            end if
        end if
    end while
end function
```

3.3 Algorithm

Root Demotion visits each vertex exactly once moving layer to layer from $\mathscr{L}_h$ to $\mathscr{L}_1$; when a vertex, v, is found such that all of its successors are more than one layer away, $\text{span}(v, s) > 1 \; \forall \; v, s \in A$, then v is "demoted" to the layer above its nearest successor, i.e. $\mathscr{L}(v) = \mathscr{L}(v) + \min(\{\text{span}(v, s) \; \forall (v, s) \in A\}) - 1$. Root Demotion is remarkably simple to implement as seen in algorithm 1.

To calculate the time complexity of Root Demotion, consider Root Demotion's constituent parts:

1. visit each vertex exactly once

2. check the layer for each successor of the current vertex while keeping the minimum

3. if required reassign the current vertex to another layer as described above

It is clear that 1) runs in $\mathcal{O}(|V|)$. As the layer of every successor for every vertex will be recalled, this enumerates the number of arcs in the graph. Keeping track of the minimum layer of the successor as one recalls the current vertex's successors' layers, 2) runs in $\mathcal{O}(|A|)$. The reassignment of layers for a vertex, should it occur, runs in $\mathcal{O}(1)$. If one uses a list of lists for the layering, $\mathscr{L}$, the removal of v from one sub-list and appending v to another can occur in $\mathcal{O}(2)$, by leveraging the traversal scheme. Thus the time complexity of Root Demotion is $\mathcal{O}(|V| + |A|)$. As for space complexity, Root Demotion requires two variables, a stack "vertices" of $|V|$ and an integer "highest_layer" (see alg 1).

3.4 Comparison

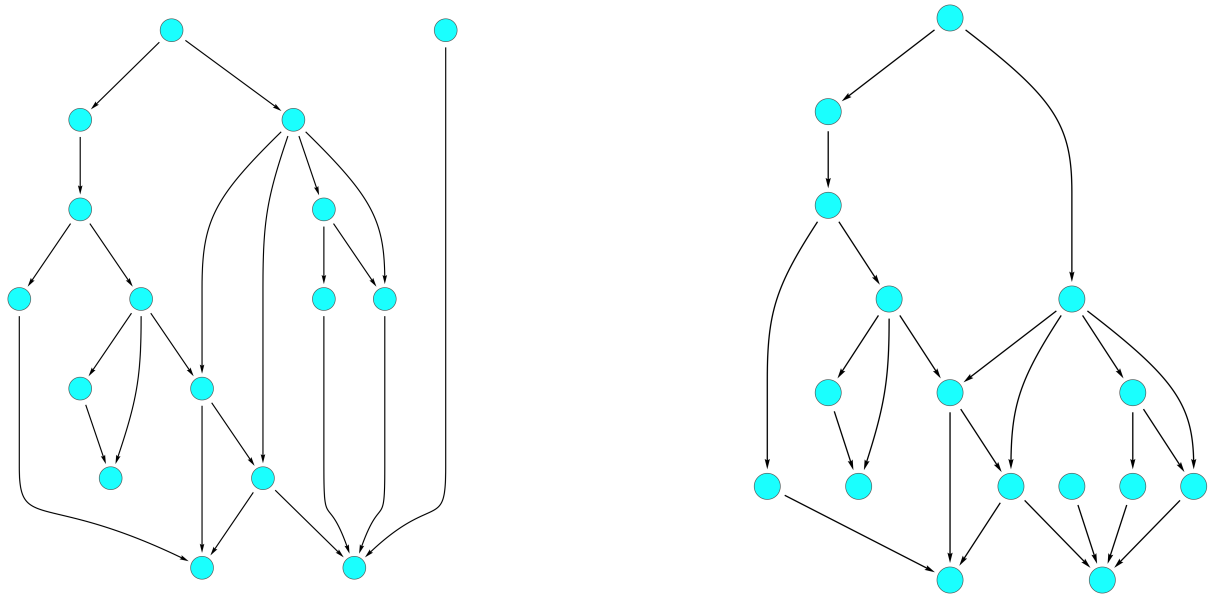
The core distinctions between the Promote Layering technique and Root Demotion are as follows:

complexity Root Demotion has a linear complexity, $O(|V| + |A|)$, whereas, at worst, Promote Layering has a cubic complexity, $O(|D| \times |V| \times (|A| + |V|))$, where the best known upper bound on the number of dummy nodes in a layered DAG is $O(\min(|V|^3, |A|^2))$ [8].

layer reassignment method Root Demotion can move a vertex down multiple layers at a time if the minimum number of layers between that vertex and any of its successors is more than one, whereas Promote Layering moves each called node up one layer and reverts if the heuristic is not met [10].

heuristic Root Demotion requires no heuristic regarding whether or not to change a vertex's layer, whereas Promote Layering uses dummyDiff as an indicator for such a decision.

These differences result in an algorithm that - in linear time - does not create new layers, can move a vertex multiple layers at a time, and requires no memory of the previous layering. Root Demotion does have a trade off, however. Although Root Demotion will move a vertex if there is a valid re-positioning, it may not find the optimal way to reduce dummy nodes. This stands in contrast to Promote Layering, which for some graphs can further reduce the number of dummy nodes needed by adding new layers. A comparison between the two post-processing techniques is shown in Figure 2.



(a) A DAG layered by Longest-Path algorithm and post-processed with Promote Layering.

(b) A DAG layered by Longest-Path algorithm and post-processed with Root Demotion

Figure 2: Example of post-processing the layering of a DAG to reduce the number of dummy vertices. Root Demotion (b) requires 8 dummy vertices, Promote Layering requires 19 and had no improvement Longest-Path algorithm without post-processing.

4 Experimental Results

For convenience, let - for the remainder of this section - the last step in layering process refer inclusively of the entire layering process, e.g. Root Demotion refers to the layering produced by the Longest-Path algorithm post-processed with Root Demotion.

To benchmark Root Demotion, we tallied the number of dummy vertices in layered graphs following the Longest-Path algorithm with no post-processing, Promote Layering, and Root Demotion. In addition we recorded the run time. We used the 1277 DAGs from the AT&T data-set as well as $11,534$ graphs in the ROME data-set; both data-sets have graphs with vertex-sets having cardinality between $[10, 110]$ vertices. Graphs were binned into representative sizes $[10, 20, \ldots, 100]$; consequently 5 graphs with more than 100 vertices were excluded to prevent a poorly represented 110 vertex bin.

Following Nikolov and Tarassov (2006) we also use the binned medians of the normalized dummy vertices [10]. The binned medians of the normalized dummy vertices show that Root Demotion consistently reduces the number of dummy vertices produced by the Longest-Path algorithm compared to Promote Layering (see, e.g. Figure 3). Root Demotion results in a binned median difference of -13 and -18 dummy vertices compared to Promote Layering and Longest-Path algorithm respectively (see, e.g. Figure 4). In the most extreme case, Root Demotion used -184 dummy vertices less than Promote Layering. In addition, Root Demotion consistently requires less run time to reduce the number of dummy vertices compared to Promote Layering (see, e.g. Figure 5).

Graphs rendered with Root Demotion require less run time for the remaining STT framework (e.g. crossing minimization, coordinate assignment) compared to both Longest-Path algorithm and Promote Layering (figure 6). For all but small graphs, the reduced run time of the STT framework makes up for Root Demotion's own run time. Peculiarly, Root Demotion used $-.46307$ layers less than Promote Layering. In the most extreme case, Promote Layering induced a $+12$ layer discrepancy (data not shown). Collectively, Root Demotion requires *less* time to *further* reduce the number of dummy vertices compared to Promote Layering resulting in a shorter run time for the STT framework.

In conclusion Root Demotion out performs Promote Layering regarding the number of dummy vertices removed and run time as a post-processing step for the Longest-Path algorithm. We find Root Demotion to be an overall simple solution with robust performance.

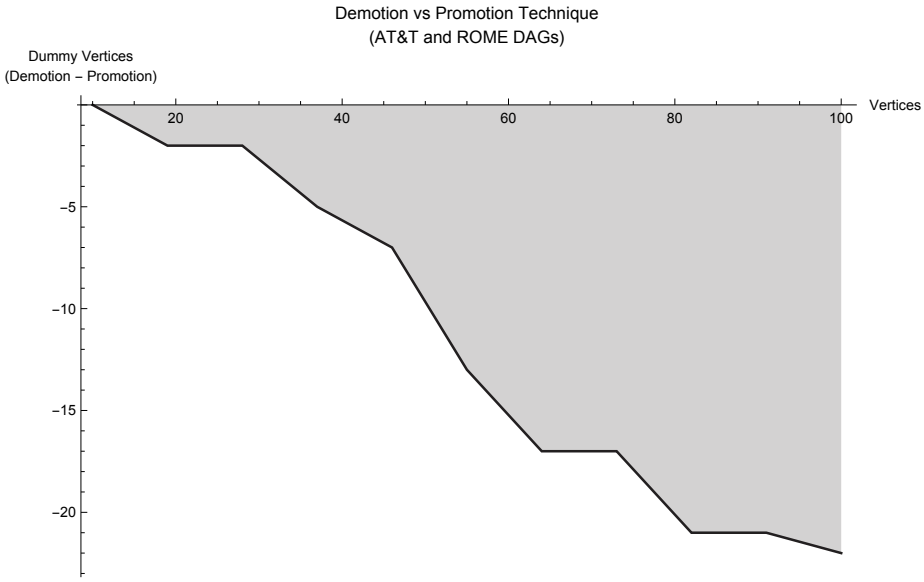


Figure 3: Difference of binned medians for dummy vertices required using Root Demotion verse Promote Layering (inclusive i.e. Longest-Path algorithm + post-processing). Post-processing the Longest-Path algorithm with Root Demotion results in layerings which consistently require fewer dummy vertices than post-processing with Promote Layering.

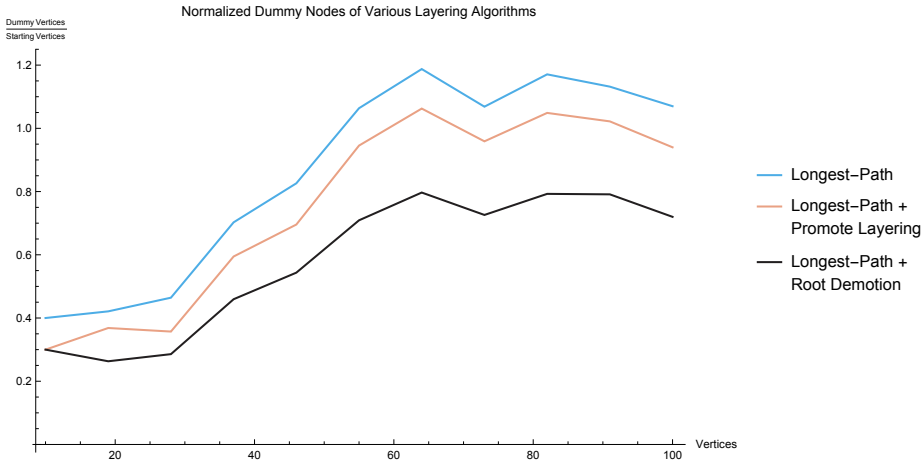


Figure 4: Median of binned normalized number of dummy vertices required for Root Demotion, Promote Layering, and Longest-Path algorithm (inclusive i.e. Longest-Path algorithm + post-processing). Root Demotion consistently results in layerings that require fewer dummy vertices than either Promote Layering or the Longest-Path algorithm without post-processing.

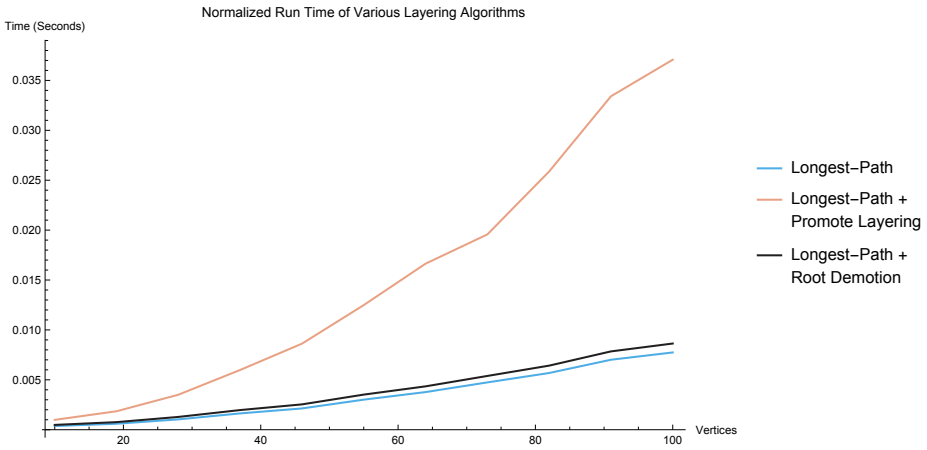


Figure 5: Median of binned run time of Root Demotion, Promote Layering, and Longest-Path algorithm (inclusive i.e. Longest-Path algorithm + post-processing). Root Demotion is consistently faster than Promote Layering as a post-processing step.

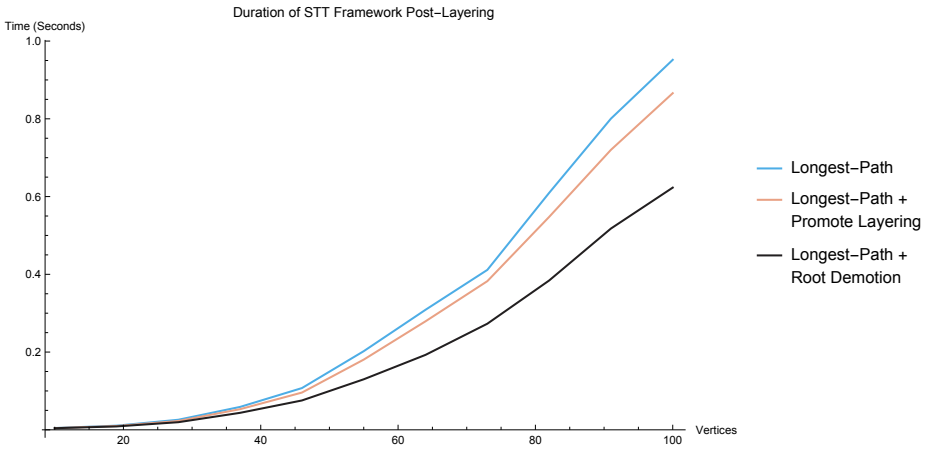


Figure 6: Median of Binned run duration of the STT framework following either Longest-Path algorithm, Promote Layering or Root Demotion.

References

[1] U. Brandes and B. Köpf. Fast and simple horizontal coordinate assignment. In P. Mutzel, M. Jünger, and S. Leipert, editors, *Graph Drawing: 9th International Symposium, GD 2001*, pages 31–44. Springer, 2002. doi:10.1007/3-540-45848-4_3.

[2] C. Buchheim, D. Ebner, M. Jünger, G. W. Klau, P. Mutzel, and R. Weiskircher. Exact crossing minimization. In P. Healy and N. S. Nikolov, editors, *Graph Drawing: 13th International Symposium, GD 2005*, pages 37–48. Springer, 2006. doi:10.1007/11618058_4.

[3] P. Eades and N. C. Wormald. Edge crossings in drawings of bipartite graphs. *Algorithmica*, 11(4):379–403, 1994. doi:10.1007/BF01187020.

[4] M. Eiglsperger, M. Siebenhaller, and M. Kaufmann. An efficient implementation of Sugiyama's algorithm for layered graph drawing. In J. Pach, editor, *Graph Drawing: 12th International Symposium, GD 2004*, pages 155–166. Springer, 2005. doi:10.1007/978-3-540-31843-9_17.

[5] E. R. Gansner, E. Koutsofios, S. C. North, and K. P. Vo. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19(3):214–230, 1993. doi:10.1109/32.221135.

[6] P. Healy and N. S. Nikolov. A branch-and-cut approach to the directed acyclic graph layering problem. In M. T. Goodrich and S. G. Kobourov, editors, *Graph Drawing: 10th International Symposium, GD 2002*, pages 98–109. Springer, 2002. doi:10.1007/3-540-36151-0_10.

[7] P. Healy and N. S. Nikolov. Hierarchical Drawing Algorithms. In Roberto Tamassia, editor, *Handbook of Graph Drawing and Visualization*, chapter 13. CRC Press, 2013.

[8] I. Lemke. Entwicklung und Implementierung eines Visualisierungswerkzeuges für Anwendungen im Übersetzerbau (Diplomarbeit). *Universität des Saarlandes*, 19994.

[9] K. Mehlhorn. *Data Structures and Algorithms, Volume 2: Graph Algorithms and NP-Completeness.* Springer, 1984. doi:10.1007/978-3-642-69897-2.

[10] N. S. Nikolov and A. Tarassov. Graph layering by promotion of nodes. *Discrete Applied Mathematics*, 154(5):848 – 860, 2006. doi:10.1016/j.dam.2005.05.023.

[11] K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man, and Cybernetics*, 11(2):109–125, 1981. doi:10.1109/TSMC.1981.4308636.

[12] L. Zheng and C. Buchheim. A new exact algorithm for the two-sided crossing minimization problem. In A. Dress, Y. Xu, and B. Zhu, editors, *Combinatorial Optimization and Applications: First International Conference, COCOA 2007*, pages 301–310. Springer, 2007. doi:10.1007/978-3-540-73556-4_32.