Crossing Minimization in k-layered Hierarchical Graphs: A Hybrid Approach

Loridge Anne Gacho[1], Zandrew Peter Garais[1]*, Nestine Hope Hernandez[1], and Jhoirene Clemente[1]

Algorithms and Complexity Laboratory, University of the Philippines Diliman, Quezon City, Philippines
*zcgarais@alumni.up.edu.ph

Abstract. Minimizing edge crossings in graph drawings is crucial for improving readability. One variant, the *k-layered hierarchical crossing minimization* problem, seeks optimal vertex orderings across all layers of a k-layered graph to reduce crossings. We explore a hybrid layer-by-layer approach that combines two among the one-sided bipartite crossing minimization (OSCM) heuristics, barycenter, permutation, and sifting. These heuristics are applied above and below a specified *cut-off index*, dividing the graph into upper and lower regions. We implemented six hybrid cut-off algorithms and tested them on sparse k-layered graphs with ten (10) layers with layer widths $b = 7$ and $b = 8$, and edge counts defined by $|E| = 2 \cdot (k-1) \cdot b$. The results demonstrate trade-offs: heuristics with higher solution quality generally incur higher computational cost. For time-sensitive applications, faster hybrids like `bary_sift` and `sift_bary` are preferable, whereas `permu_sift` and `sift_permu` are more suitable when solution quality is prioritized. Selecting the appropriate hybrid and cut-off value is key to balancing performance and efficiency.

Keywords: computational geometry, graph theory, crossing minimization

1 Introduction

Graph drawing has become increasingly important across various fields, such as science, technology, and engineering, with growing demand for better visualizations. Unfortunately, edge crossings become a major obstacle in areas like designing electrical circuit diagrams and travel routes–such as metro line or road maps–where more crossings not only impair readability but also increase production costs. Therefore, crossing minimization is crucial for enhancing visualization and meeting application-specific needs.

Among the many graph classes tackled in crossing minimization, two-layer graph layouts have been widely studied due to their applications in hierarchical graph drawing and scheduling diagrams. A key problem in this domain is the one-sided crossing minimization (OSCM) problem, where one layer is fixed while

J. Caro et al. (eds.), *Proceedings of the Workshop on Computation: Theory and Practice (WCTP 2025)*, Atlantis Highlights in Computer Sciences 24,
https://doi.org/10.2991/978-94-6239-638-8_21

the other can be reordered to minimize crossings. Since OSCM is known to be NP-hard [2, 8], heuristic methods have been developed to find approximate solutions efficiently.

However, many real-world applications involve k-layered hierarchical graphs, where vertices are organized across multiple layers and edges run between adjacent layers. In such graphs, minimizing crossings becomes more difficult, as it is NP-hard [2], and decisions in one layer can significantly affect others. Standard layer-by-layer approaches, such as those used in the Sugiyama framework [10, 4], often extend two-layer heuristics iteratively but tend to get stuck in local minima as the number of layers increases.

Given the limited research for k-layered hierarchical graphs, this paper outlines the development of hybrid heuristics that use existing OSCM algorithms on directed acyclic multi-layered graph drawings. These graph drawings have no long edges, edges that connect nodes of non-consecutive layers.

To achieve this, we aimed to (1) develop and evaluate multiple hybrid heuristics for crossing minimization in multi-layered hierarchical graphs; (2) test the hybrid approaches on multi-layered hierarchical graphs to evaluate their scalability, adaptability, and the impact of different cut-off strategies; and (3) evaluate the trade-offs between runtime and crossing quality across different variants of our hybrid algorithm.

The variables that are relevant for evaluation are the graph size, edge crossing counts, edge numbers, computational time, and algorithm parameters such as a hybrid algorithm's cut-off index. Given that the study employs heuristic methods, achieving the optimal solution is not guaranteed.

Section 2 of the paper will discuss the relevant concepts that will be necessary in understanding the rest of the paper. Meanwhile, Section 3 will discuss the datasets, the aspects of the hybrid algorithm, and the experiments that were conducted to test the variants of the hybrid algorithm. Then, Section 4 outlines the results of the experiments and their analyses. Finally, Section 5 ties the results to the aims of the study and provides future directions.

2 Preliminaries

Before proceeding, we equip the readers, assumed to have a basic grasp of graph theory [1, 2, 5], with the necessary terminologies and concepts that can guide them in understanding the rest of the study.

A *hierarchical graph* $G = (V, E, \phi)$ is a directed acyclic graph (DAG) where V represents the set of the nodes, E the set of the directed edges, and $\phi : V \rightarrow$

$\{1, 2, \ldots, k\}$ is a function that assigns each node to one of k distinct levels. The nodes are divided into k non-overlapping groups $V_1, V_2, \ldots, V_k$, such that every node belongs to exactly one group, and V is the union of these groups. Each group V_j corresponds to level j, determined by $\phi^{-1}(j)$. While edges only connect nodes from lower to higher levels, they may connect vertices across consecutive or non-consecutive layers. However, for this study, we will only consider k-layered hierarchical graphs with directed edges that connect vertices across consecutive layers [4].

2.1 One-Sided Bipartite Crossing Minimization

In the one-sided bipartite crossing minimization problem, a two-layer bipartite drawing has the layer L_0 fixed while L_1 is free. Here, the goal is to determine a permutation π of L_1 that belongs to the set of all L_1 permutations $\prod(L_1)$ such that π causes the least number of edge crossings in the two-layer bipartite graph G. As it is computationally expensive to explore all permutations, several *heuristics* have been developed to approximate the optimal solution:

2.1.1 Barycenter Heuristic The barycenter heuristic calculates the average x-coordinate of neighboring vertices in the fixed layer (L_0) for each vertex in the free layer (L_1). This requires $O(n)$ time to compute averages for n vertices in L_0 and sorting these averages takes $O(m \log m)$ time for m vertices in L_1. Consequently, the overall time complexity of the heuristic is $O(n + m \log m)$ [6]. However, it can perform $\Omega(\sqrt{n})$ worse than the optimal number of crossings, where n is the number of vertices [5, 9].

2.1.2 Median Heuristic The median heuristic is similar to the barycenter heuristic, but it uses the median of the x-coordinates of the neighboring vertices to minimize crossings [5]. This approach has been shown by [2] to have an upper bound of three times the optimum number of crossings. It has the same running time bounds as the barycenter heuristic [4].

2.1.3 Permutation Algorithm The permutation algorithm is exhaustive because it considers every permutation of the free layer, producing an optimal result. However, this is computationally expensive and is only used for small graphs [9]. The time and space complexity of this heuristic is $O(n!)$, where n represents the number of vertices in the free layer.

2.1.4 Sifting for One Sided Crossing Minimization Reference [7] introduced the *sifting heuristic*, a local optimization technique that iteratively moves a single vertex within a layer to minimize crossings. The process involves:

1. Selecting a vertex and temporarily removing it from the ordering. In selecting a vertex to be reordered, it is popped from the head of a priority queue, where priority is determined by the indegree value of a vertex.

2. Reinserting it at various positions while calculating the number of crossings.
3. Placing it in the position that minimizes crossings.

Thus, sifting runs in $O(n^2)$ time, making it more computationally expensive than the barycenter method but still significantly faster than permutation. Compared with other OSCM heuristics, the key findings from their work are:

- **For sparse graphs** ($|E| = |V_1| + |V_2|$): Sifting consistently outperformed the barycenter and median heuristics, producing crossing numbers closer to the optimal solution.
- **For dense graphs**: While sifting remained effective at lower densities, its advantage diminished as edge density increased, with barycenter and split heuristics performing comparably.

2.2 k-Layered Hierarchical Graph Crossing Minimization

Extending beyond two-layered graphs, k-layered crossing minimization tackles graphs with multiple layers, where the goal is to minimize crossings across several connected layers to improve visual clarity. Since the one-sided bipartite problem is a simplified case of the k-layered problem, the heuristics used in OSCM can be applied to the more complex k-layered hierarchical graphs. The common way to apply these OSCM heuristics is to use the layer-by-layer sweep approach, elaborated in Section 3.3.

Current research in k-layered hierarchical graph crossing minimization has focused on the application of common heuristics and their hybrids, along with the layer-by-layer sweep, and the formulation of novel heuristics. Reference [9] considered the performance of a hybrid algorithm and traditional methods with a randomness approach against their unmodified versions. Their hybrid heuristic behaves such that if the number of vertices in a layer is greater than six (6), barycenter is applied. Otherwise, permutation is used. For their modified permutation and barycenter, they randomly permuted the layers before applying a heuristic because they observed that subsequent sweeps would not improve a section of the graph—instead, the swapped vertices would oscillate. However, the randomized and unmodified heuristics do not have a significant difference in their runtime results. Moreover, the permutation heuristic is impractical for moderate and large graphs. Meanwhile, the hybrid approach performed similarly to both barycenter versions, showing no significant improvement or decline in performance. Inspired by these findings, we have decided to implement a hybrid approach in our research to explore its potential effectiveness further.

In [7], a novel algorithm was introduced for k-layer straight-line crossing minimization known as sifting, originally used in the optimization of binary decision diagrams. The authors implemented the sifting algorithm for k-layered graphs in two ways: layer-by-layer sifting and global sifting. The layer-by-layer sifting applies the sifting algorithm as an OSCM problem per layer of a k-layered

graph, while the global sifting considers all of the layered graph's vertices at sifting. First tested on small, sparse 2-layered graphs with $|E| = |V_1|+|V_2|$, sifting performed better than the barycenter method, but for larger sparse 2-layered graphs, barycenter performed better than sifting. Experimenting on graphs with a constant edge density of 20 percent and a varying number of vertices, both layer-by-layer sifting and global sifting performed better than the barycenter. However, sifting yielded higher computation time. For $k * b$ graphs, global sifting performed better than traditional heuristics and layer-by-layer sifting. In this type of graph, layer-by-layer sifting performed slightly less than the barycenter and median heuristic. Overall, this research shows the potential of sifting for further study in other graph types.

3 Methodology

3.1 Dataset

3.1.1 Two-Layered Graphs Two (2) types of graphs are relevant in the bipartite graph dataset: *sparse bipartite graphs* $G_0 = (V, E)$ and *bipartite graphs with varying densities* $G_1 = (V, E)$. In both cases, the vertex set is defined as $V = L_0 \cup L_1$, where $|L_0| = |L_1|$. For G_0, the number of edges is set to $|E| = |L_0|+|L_1|$, ensuring that each vertex has an average of two (2) connections. Meanwhile, for G_1, depending on the size of a graph, the number of edges is set to the number required to achieve the range of bipartite densities [3] 0.1 to 0.9. The graphs were randomly generated using the Python `NetworkX` library, with the constraint that no vertex is disconnected.

3.1.2 k-Layered Graphs For benchmarking, we generated two (2) types of graphs: *uniform-width sparse k-layered graphs* [7] $G_0 = (V, E)$ where $V = V_0 \cup V_1 \cup ... \cup V_k$, $|V_i| = b$ (layer width), and $|E| = 2 \cdot (k-1) \cdot b$; and *alternating-layer sparse k-layered graphs* $G_0 = (V, E)$ where the vertex set is partitioned into multiple layers as $V = V_0 \cup V_1 \cup ... \cup V_k$. The layers with even indices ($i = 0, 2, 4, ...$) have sizes of $|V_i| = 8$, while the layers with odd indices ($j = 1, 3, 5, ...$) have size $|V_j|$ strictly less than 8. For both of the graph types, for each pair of two consecutive layers $|V_p|$ and $|V_{p+1}|$, the number of edges between them is $|E| = 2 * \min(|V_p|, |V_{p+1}|)$. All the edges in the k-layered dataset only connect nodes of consecutive layers.

3.2 Visualization tools

To aid in the development and debugging process, we implemented graph visualization tools using Python libraries `NetworkX` and `Matplotlib`. By visual inspection, before and after applying crossing minimization algorithms, we were able to verify algorithm behavior and graph generation. While *not used for quantitative evaluation*, it allowed us to observe:

- How different algorithms rearrange vertex positions.

– The relative effectiveness of each method in reducing crossings.
– Cases where certain heuristics failed to significantly improve the layout.

3.2.1 Two-Layered Graphs Figures 1 and 2 are visualizations of a generated bipartite graph before and after the application of the barycenter heuristic.

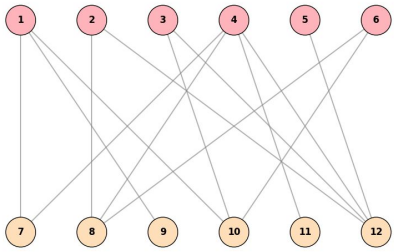

Fig. 1. Bipartite graph before applying the Barycenter heuristic.

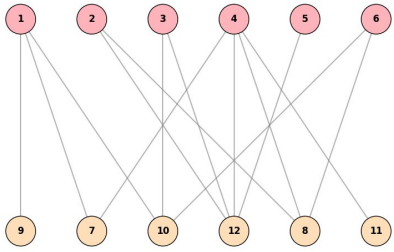

Fig. 2. Bipartite graph after applying the Barycenter heuristic.

3.2.2 k-Layered Graphs Figure 3 is a visualization of a generated k-layered graph with 10 layers and a constant layer width of six (6) nodes.

3.3 The Layer-by-Layer Sweep

The Layer-by-Layer Sweep methodology is an effective strategy for minimizing edge crossings in *hierarchical graphs*. It involves processing the graph in a layer-by-layer manner to achieve an optimized visual layout with minimal edge crossings.

The layer-by-layer sweep technique for a k-layered graph involves the following process: Initially, the vertex positions in the bottommost layer (L_1) are fixed. Then, we utilize known heuristics from the One-Sided Bipartite Crossing Minimization (OSCM) method to minimize crossings between edges in the 2^{nd} layer, treating it as a free layer. This process continues layer by layer, keeping

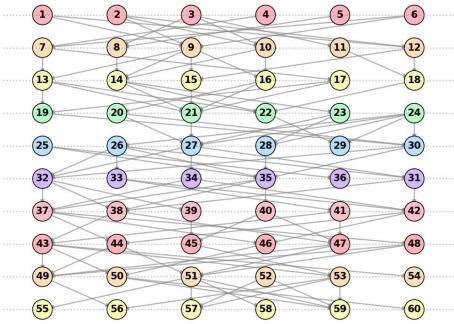

Fig. 3. A k-layered hierarchical graph with 10 layers having a width of six (6) nodes per layer.

the new order of vertices fixed in each subsequent layer and applying the OBCM heuristics. Once all layers have been processed, we then move upwards, maintaining the fixed vertex order in layer L_i, and re-arrange the vertices in layer L_{i-1} for $i = k, k-1, ..., 2$. This iterative sweep is performed alternately from top to bottom and bottom to top, repeating the process until no further improvements can be made. We conclude that the problem involves solving a series of two-layer crossing minimization problems. This involves minimizing the crossings between adjacent layers L_i and L_{i+1} with L_i held fixed, while reordering the vertices in L_{i+1}.

The heuristics will be applied in a layer-by-layer sweep, treating the crossing minimization of every two layers as a one-sided bipartite crossing minimization problem. However, [9] pointed out that there are no clear criteria for determining when to halt the sweeping process. Hence, they devised a 'forgiveness number' with a value of 20. The 'forgiveness number' is the number of times the algorithm makes sweeps when it no longer improves the condition of the graph. The pseudocode of the layer-by-layer sweep is in Appendix B.

3.4 Proposed k-layered Hybrid Heuristics

To address the mentioned limitations of barycenter, sifting, and permutation, multiple hybrid approaches combining the strengths of these heuristics are proposed to be evaluated.

The proposed hybrid crossing minimization method explores the potential trade-off between running time and solution quality by dividing the layered graph into two (2) regions, separated by a *cut-off index*, and uses two (2) OSCM heuristics. The graph is processed in layer-by-layer sweeps, but only a subset of layers is handled by each heuristic: Algorithm A is applied to layer pairs before the cut-off index, while Algorithm B is applied to layer pairs after it. A *cut-off*

index is the index of the layer where algorithms will not go beyond. Hence, when the *cut-off index* is 0, Algorithm B will be the solver for the whole graph, and when its value is the index of the last layer, Algorithm A is. The pseudocode and visualization are in Algorithm 1 and Fig. 4. Table 1 shows the different hybrid methods that were investigated.

The motivation behind introducing the cutoff index is to provide flexibility in balancing between solution quality and computational efficiency. Applying different heuristics to the regions above and below the cut-off allows the method to control how much computational effort is allocated toward achieving near-optimal results. For instance, a more exhaustive heuristic can be applied to a limited portion of the graph to improve optimality, while a faster heuristic handles the remaining layers to conserve resources. This design enables users to adjust the trade-off depending on the desired balance between accuracy and runtime.

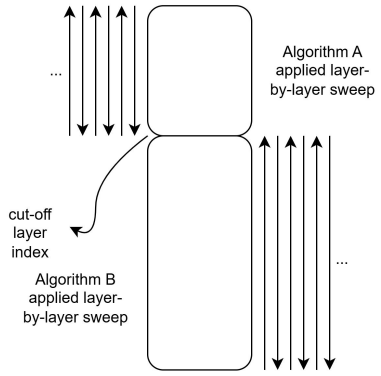

Fig. 4. Diagram of Hybrid Method

Table 1. Hybrid methods and their component heuristics

Name	**Algorithm A**	**Algorithm B**
bary_sift	Barycenter	Sifting
sift_bary	Sifting	Barycenter
permu_sift	Permutation	Sifting
permu_bary	Permutation	Barycenter
bary_permu	Barycenter	Permutation
sifting_permu	Sifting	Permutation

Algorithm 1 HybridMethod

Require: Graph G, Algorithm A, Algorithm B, cutoff value k
Ensure: Updated graph configuration S
$total\_layers \leftarrow$ number of layers in G
if $k > 0$ **then**
    / Layer-by-layer sweep using Algorithm A from layers 0 to k */*
    Downward sweep from layer 1 to k using A
    Checkpoint()
    Upward sweep from layer $k - 1$ to 0 using A
    Checkpoint()
end if
if $k < total\_layers - 1$ **then**
    / Layer-by-layer sweep using Algorithm B from layers k + 1 to total_layers-1*/*
    Downward sweep from layer k to $total\_layers - 1$ using B
    Checkpoint()
    Upward sweep from layer $total\_layers - 2$ to k using B
    Checkpoint()
end if
return G as S

The crossing counter, a function that counts the crossing number of a bipartite graph, is an important subroutine that is used in every aspect of algorithms in the crossing minimization problem. Since this function is one of the most called functions within heuristics and the layer-by-layer algorithm, it has to be efficient. In light of this, the crossing counter developed by [5] was used in our implementation.

3.5 Experiments

3.5.1 Two-Layered Graphs

Experiment 1: Sparse Bipartite Graphs with Increasing Number of Vertices We consider randomly generated sparse bipartite graphs defined as $|E| = |V_1| + |V_2|$ where $|V_1| = |V_2|$. For each graph type with layer sizes $|V| \in \{6, 8, 10\}$, we processed 20 samples. All the OSCM heuristics were used, including the permutation algorithm as a benchmark for the optimal solution. The final results for each graph type are obtained by *averaging the results across the 20 samples.*

Experiment 2: Graphs with Increasing Density We consider a class of bipartite graphs with layers V_1 and V_2, where $|V_1| = |V_2|$. The graphs are generated with increasing edge densities, ranging from 0.1 to 0.9. For each density level, *20 graph samples* are generated and evaluated using *OSCM heuristics*, including the permutation algorithm as the optimal solution. The final results for each density are obtained by *averaging the results across the 20 samples.* Due to hardware limitations and algorithm complexities, we only considered graph classes with $|V_1| = |V_2| \in \{5, 8, 10\}$.

3.5.2 k-layered Graphs

Experiment 3 (N-M Hybrid 1): Hybrid 1 vs Full Barycenter on Increasing Number of Vertices per Layer We evaluate the performance of the hybrid algorithm of [9], which we will refer to as 'Hybrid 1,' against both Full Permutation [9] and Full Barycenter (multisweep) methods. The experiment uses randomly generated sparse k-layered graphs with a fixed number of layers $k = 10$. Instead of uniform layer sizes, we alternate the number of nodes per layer between n and m. In this experiment, we fix $n = 8$ and vary $m \in \{4, 5, 6, 7, 8\}$.

For each configuration of m, we generate 20 random graphs and average the results. All three heuristics are applied: Hybrid 1 (permutation + barycenter), Full Barycenter, and Full Permutation. The **Full Permutation method serves as the baseline**, as it is theoretically the closest to optimal in terms of minimizing edge crossings.

Experiment 3 (N-M Hybrid 1): Hybrid 1 vs Full Barycenter on Increasing Number of Vertices per Layer We evaluate the performance of the hybrid algorithm of [9], which we will refer to as 'Hybrid 1,' against both Full Permutation [9] and Full Barycenter (multisweep) methods. The experiment uses randomly generated sparse k-layered graphs with a fixed number of layers $k = 10$. Instead of uniform layer sizes, we alternate the number of nodes per layer between n and m. In this experiment, we fix $n = 8$ and vary $m \in \{4, 5, 6, 7, 8\}$.

For each configuration of m, we generate 20 random graphs and average the results. All three heuristics are applied: Hybrid 1 (permutation + barycenter), Full Barycenter, and Full Permutation. The **Full Permutation method serves as the baseline**, as it is theoretically the closest to optimal in terms of minimizing edge crossings.

Experiment 4: Performance of Hybrid Algorithms Across Different Cut-off Indices (M-M Uniform Width Graphs) The performance of hybrid algorithms under different cut-off indices against k-layered graphs was investigated. We considered classes of *uniform-width sparse k-layered graphs* (i.e. the width of the layers follows the pattern $n-n-n-n\ldots$) having uniform layer widths of 7 and 8, and 10 layers. For each graph class, twenty (20) samples are generated. We benchmarked every proposed hybrid algorithm stated in Table 1 in every possible cut-off index, on each generated sample. All possible cut-off indices are 0 to 9 (zero-indexed), given that there are 10 layers on all generated samples. In effect, 60 algorithms were tested against one sample, on all twenty samples. Collected data include original graph crossings, new graph crossings on all methods on all their cut-off value variations, and the time spent by a heuristic doing a layer-by-layer sweep within a hybrid algorithm. The **Full Permutation method, serves as the baseline**. In implementation, the results of `permu_sift` with a cut-off value of 9 was treated as the full permutation because at this configu-

ration, permutation is already applied on all graph layers. The final results for each graph type are obtained by *averaging the results across the 20 samples.*

3.6 Performance Metrics for Benchmarking

The primary metrics for comparison and analysis are the average computational time (total and component-wise) and average empirical approximation ratio (EAR). The computational time was measured in seconds.

The Empirical Approximation Ratio (EAR) for method m at cut-off value c is defined as:

$$EAR(m, c) = \frac{\overline{C}(m, c)}{\overline{C}(baseline)}, \tag{1}$$

where

- $\overline{C}(m, c)$ is the average number of crossings produced by method m with cut-off c, averaged over all graph samples,
- $\overline{C}(baseline)$ is the average number of crossings produced by the baseline method, `permu_sift` with cut-off value equal to the last layer index, $k - 1$, averaged over all graph samples. k is the total number of layers in a graph class.

An EAR value closer to 1.0 indicates better approximation to the best-known solution, while larger values imply worse solution quality. The total average computational time is the average of the total running times of a hybrid algorithm across all samples. Meanwhile, the average component-wise computational time refers to the average time spent by each sub-algorithm within a hybrid algorithm, also computed across all samples.

3.7 Hardware and Software Specifications

Table 2. Experiment Specifications

Specification	**Experiment 1**	**Experiment 2 / k-Layered**
System	Personal Computer	High-Performance Computing Server
Processor	Intel Core i5-9300H	AMD EPYC 7513
RAM	16 GB	16 GB DRAM
Operating System	Windows 11 Home Edition	Linux Ubuntu 22.04.5 LTS
Software	Visual Studio Code	—
Programming Language	Python	Python

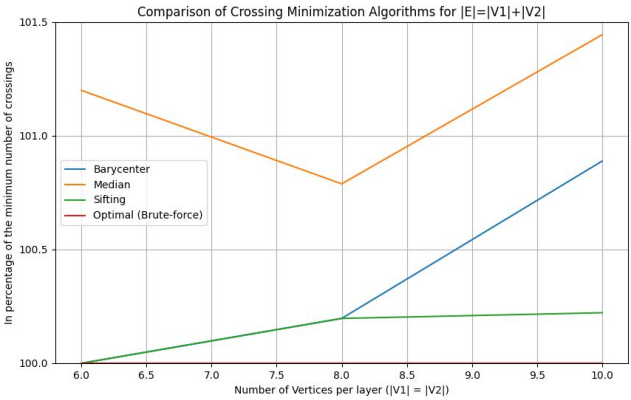

Fig. 5. Performance of heuristics for sparse graphs with layer widths 6, 8, and 10

4 Results and Analysis

4.1 Experiment 1

4.1.1 Observations

- The **median heuristic** performed the worst out of the three heuristics with the number of crossings produced higher for all types of graphs.
- The **sifting heuristic** overall performed the best out of the three heuristics for all types of graphs, with it showing the lowest number of crossings for $|V_1| = |V_2| = 10$.
- The **barycenter heuristic** performed on par with the sifting heuristic for $|V_1| = |V_2| = 6$ *and* 8 but worse at higher vertex count 10.

The results in Fig. 5 indicate that the median heuristic performed the worst for all values of layer sizes for all sparse graphs. For smaller graphs, the barycenter and sifting have equal performance but only diverge at higher values of layer sizes.

4.2 Experiment 2

4.2.1 Observations

- The **sifting heuristic** consistently performs well and remains the closest to the optimal solution across all density levels.
- The **barycenter heuristic** starts off weaker at low densities but improves as density increases, performing more competitively at higher densities.
- The **median heuristic** exhibits inconsistent performance, sometimes exceeding the crossings of both barycenter and sifting heuristics.

– As edge density increases, the performance of the barycenter and sifting heuristics becomes closer to each other, with barycenter improving in higher-density graphs.

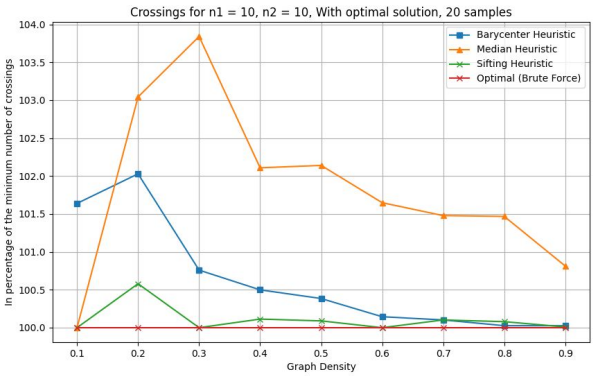

Fig. 6. Performance of heuristics vs. optimal solution across different edge densities.

The results in Fig. 6 indicate that as edge density increases, the performance of the barycenter and sifting heuristics improves and gets closer to the optimal solution. The median heuristic, however, remains inconsistent across different densities. At higher densities, barycenter performs as well as sifting while having better runtime efficiency. Overall, sifting remains the most reliable choice in terms of crossing minimization, but barycenter becomes a competitive alternative when computational efficiency is a priority.

4.3 Experiment 3

4.3.1 Observations

– Hybrid 1 consistently performs better than Full Barycenter in all values of m.
– As m increases, the performance gap between Hybrid 1 and Full Barycenter generally decreases.
– Full Barycenter consistently yields higher crossing percentages than Hybrid 1, except at $m = 7, 8$ where their values converge.

As shown in Fig. 7, both heuristics yield more crossings than Full Permutation, as expected, but Hybrid 1 consistently results in fewer crossings than Full Barycenter. The percentage values are calculated relative to the minimum number of crossings produced by Full Permutation. Notably, Hybrid 1 maintains a performance advantage in most cases, especially when m is smaller. As m increases, the performance of Hybrid 1 and Full Barycenter converges. This is

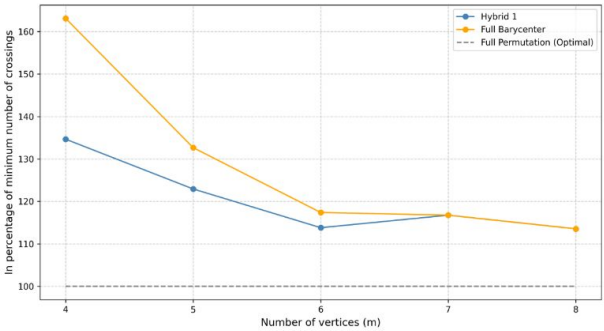

Fig. 7. Performance comparison of Hybrid 1 and Full Barycenter against Full Permutation (baseline)

expected since, with more vertices, Hybrid 1 tends to default to using barycenter alone when no better permutation is found, reducing the distinction between the two heuristics in denser instances. Additionally, the results suggest that Hybrid 1 provides a favorable balance between computational cost and solution quality, particularly in sparser graphs. Unlike Full Permutation, which is computationally infeasible for large inputs, Hybrid 1 achieves lower crossings than Full Barycenter while remaining significantly more scalable than exhaustive permutation. Overall, this experiment highlights the robustness and adaptability of Hybrid 1, showing that it captures the advantages of permutation-based refinement when feasible, while gracefully degrading to the reliability of barycenter when the search space becomes too large.

4.4 Experiment 4

4.4.1 Observations for Graphs with Layer Width of 7 Nodes It is important to note that for a certain graph class, the densities of all generated graphs are identical. Figure 8 shows the performance of the different hybrid algorithms in terms of the EAR across increasing cut-off indices. For `permu_bary` and `sift_bary`, there is a general downward EAR trend as the value of the cut-off index increases. This suggests that increasing the cut-off index allows the first algorithms of these hybrid methods to fully explore the search space, yielding better solutions as they increasingly operate on more regions of the graph. On the contrary, `bary_sift` and `bary_permu` perform worse as the cut-off value increases. This can be attributed to the barycenter increasingly taking on the role of the solver for most regions of the graph. Meanwhile, `sifting_permu` generated the least EAR for most of the cut-off values. Algorithm `permu_sift` almost follows the mentioned trend, except `bary_permu` outperforms it at cut-off values 1 and 3. An explanation could be that since sifting and permutation explore a much larger search space, they were able to produce better solutions than the other algorithms in most of the cut-off values. At the cut-off value 9, all algorithms except `bary_sift` and `bary_permu` achieved a solution very close

to the baseline solution. This can be attributed to sifting and permutation being the sole solver in all of the layers.

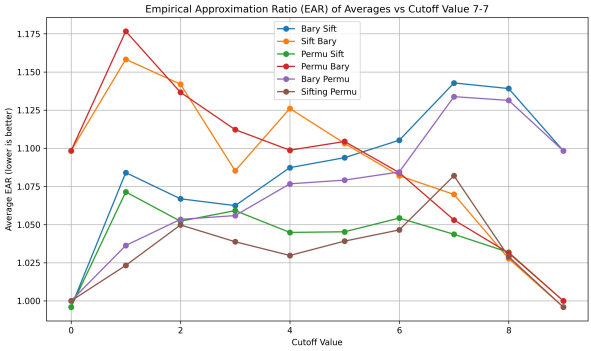

Fig. 8. Empirical Approximation Ratio (EAR) across different cutoff values for layered graphs of width 7.

It can be observed in Fig. 9 that the methods `sift_bary` and `bary_sift` exhibited the generally least amount of total running time in seconds, close to zero, in all cut-off values compared to the algorithms. However, this comes at the cost of a higher EAR than the other algorithms. This may be less desirable when layout quality is critical. For `permu_bary` and `bary_permu`, there is a visible downward trend of EAR, showing improving solutions, while the average total running time increases, as cut-off values increase and decrease, respectively. It indicates that permutation is increasing its prevalence, explaining the increase in runtime. Meanwhile, `permu_sift` and `sift_permu` have EAR that is generally closer to the baseline solution for all cut-off values, albeit with a rising trend in running time, as cut-off values increase and decrease, respectively.

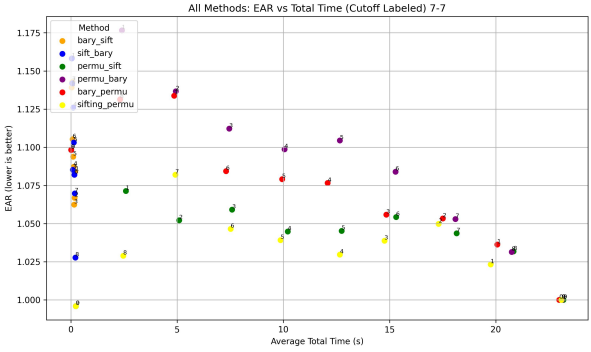

Fig. 9. Empirical Approximation Ratio (EAR) versus average total runtime for various hybrid methods on graphs with layer width 7.

Figures A1 to A6 in Appendix A show stacked bar charts that break down the average execution times of the individual component algorithms in each hybrid method. The graphs track how these times change as the cut-off value increases. In Figs. A1 and A2, the component runtime of the barycenter algorithm is almost negligible, as permutation dominates. In Figs. A6 and A5, the runtime of the sifting algorithm is more distinguishable, but permutation is still dominant. Also, algorithms without permutation tend to have running times of less than 0.5 seconds. These provide empirical evidence for the theoretical running time of these algorithms, where permutation has a factorial running time, sifting is quadratic, and barycenter is linear.

There is a decreasing trend of total runtime as the cut-off value increases in Figs. A1, A3, and A5, while the opposite occurs in Figs. A2, A4, and A6. The decreasing trend indicates that for hybrid algorithms, as the cut-off value increases, the more computationally expensive algorithms (e.g., permutation in `bary_permu` and sifting in `bary_sift`) are applied to a smaller number of layers. Meanwhile, for algorithms with increasing trends, the opposite occurs.

4.4.2 Observations for Graphs with Layer Width of 8 Nodes It can be observed that the same general trends for EAR remain the same for increasing cut-off values and average total running times (see Fig. 10 and Fig. 11). While graphs with a layer width of 8 have similar runtime trends with the smaller graphs, there is a notable increase in overall execution time as seen in the stacked timing plots. This great increase in runtime is highly evident in hybrid algorithms that use permutation, showing its poor scalability with respect to input size.

Similarly, the layered graphs follow the same EAR and runtime trends. However, they demonstrate a more pronounced increase in total execution time, further emphasizing the growing computational cost associated with larger graph sizes. This reinforces the observation that scalability is a major limitation for some heuristics, especially for permutation, as graph sizes grow.

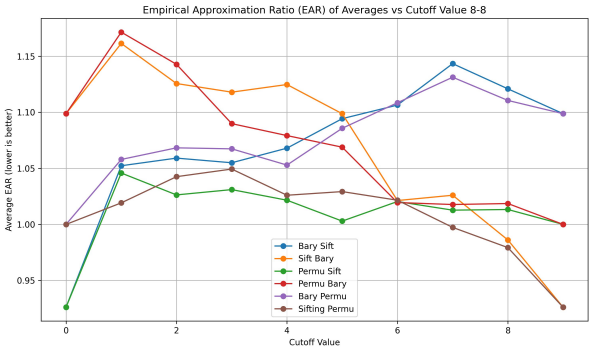

Fig. 10. Empirical Approximation Ratio (EAR) across different cutoff values for layered graphs of width 8.

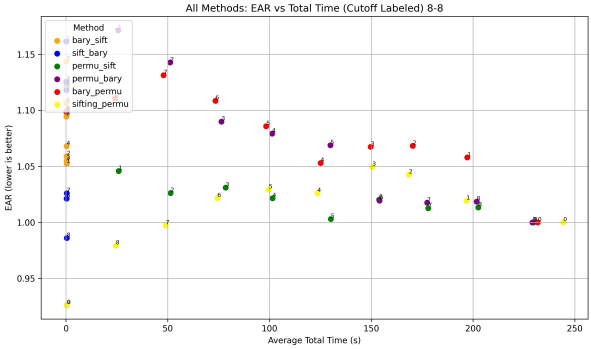

Fig. 11. Empirical Approximation Ratio (EAR) versus average total runtime for various hybrid methods on graphs with layer width 8.

5 Conclusion and Future Work

In this paper, we evaluated the performance of hybrid algorithms, developed with existing heuristics (i.e., barycenter, permutation, sifting) in terms of computational time and solution quality. The proposed hybrid methods featured a cut-off component to divide the input graph into regions solved by differing heuristics and were tested across various cut-off conditions and graph configurations, particularly graphs with widths of 7 and 8 vertices.

Most of the hybrid methods exhibit a consistent trend between average empirical approximation ratios (EARs) and average running times across differing cut-off values. Variations of hybrid algorithms that can provide better solutions often suffer from high computational time, while the opposite is also true. This trade-off between running time and solution quality is apparent because hybrid algorithms with cut-off values that allow more computationally intensive

algorithms to cover more subgraph regions exhibit higher running time. In addition, there is a scalability issue, especially with computationally intensive graphs dealing with larger graphs. This highlights the need to choose the appropriate algorithm based on the graph conditions and in consideration of the trade-offs between solution quality and running time. The appropriate algorithm to be used depends on the priority of the user. When execution time is a constraint in minimizing graphs, hybrid heuristics `bary_sift` or `sift_bary` could be utilized, albeit with lower solution quality. Since `permu_sift` and `sift_permu` provide more stable solution quality across all cut-off values, these could be used when solution quality is preferred. However, these should be used with an appropriately chosen cut-off to balance the computational time.

Future research endeavors may focus on the following areas of exploration:

- Hybrid algorithms involving other one-sided bipartite crossing minimization algorithms, such as the branch-and-cut exact algorithm.
- Comparison with global k-layered crossing minimization algorithms.
- Implementation of a parallelized k-layered algorithm, to improve execution times.
- Conducting theoretical analysis on the convergence and optimality of the proposed hybrid methods, and extending evaluations to a broader range of graph configurations beyond sparse graphs with fixed layer widths.

References

1. Deo, N.: Graph Theory with Applications to Engineering and Computer Science. Prentice-Hall, Englewood Cliffs (1974). https://dl.acm.org/doi/10.5555/1096898
2. Eades, P., Wormald, N.C.: Edge crossings in drawings of bipartite graphs. Algorithmica **11**(4), 379–403 (1994). https://doi.org/10.1007/BF01187020
3. Hagberg, A.A., Schult, D.A., Swart, P.J.: Exploring network structure, dynamics, and function using NetworkX.
In: Varoquaux, G., Vaught, T., Millman, J. (eds.) *Proceedings of the 7th Python in Science Conference (SciPy 2008)*, pp. 11–15 (2008)
4. Healy, P., Nikolov, N.: Hierarchical drawing algorithms. In: Tamassia, R. (ed.) Handbook of Graph Drawing and Visualization. CRC Press. https://cs.brown.edu/people/rtamassi/gdhandbook/chapters/hierarchical.pdf
5. Hol, S.: On crossing minimization problems: complexity and a heuristic approach. Bachelor's thesis, Universiteit Utrecht (2024). https://studenttheses.uu.nl/bitstream/handle/20.500.12932/46720/final_Bachelor_Thesis_Mathematics.pdf
6. Mäkinen, E., Siirtola, H.: The barycenter heuristic and the reorderable matrix. Informatica (Slovenia) **29**, 357–364 (2005)
7. Matuszewski, C., Schönfeld, R., Molitor, P.: Using sifting for k-layer straightline crossing minimization. In: Jünger, M., Mutzel, P. (eds.) Graph Drawing 1999. LNCS, vol. 1731, pp. 217–228. Springer, Berlin, Heidelberg (1999). https://link.springer.com/chapter/10.1007/3-540-46648-7_22
8. Muñoz, X., Unger, W., Vrt'o, I.: One sided crossing minimization is NP-hard for sparse graphs. In: Mutzel, P., Jünger, M., Leipert, S. (eds.) Graph Drawing 2002. LNCS, vol. 2265, pp. 115–123. Springer, Berlin, Heidelberg (2002). https://doi.org/10.1007/3-540-45848-4_10

9. Patarasuk, P.: Crossing reduction for layered hierarchical graph drawing. Master's thesis, Florida State University (2004). https://repository.lib.fsu.edu/islandora/object/fsu:180382/datastream/PDF/view
10. Sugiyama, K., Tagawa, S., Toda, M.: Methods for visual understanding of hierarchical system structures. IEEE Trans. Syst. Man Cybern. **11**(2), 109–125 (1981). https://doi.org/10.1109/TSMC.1981.4308636

Appendix A Additional Runtime Figures

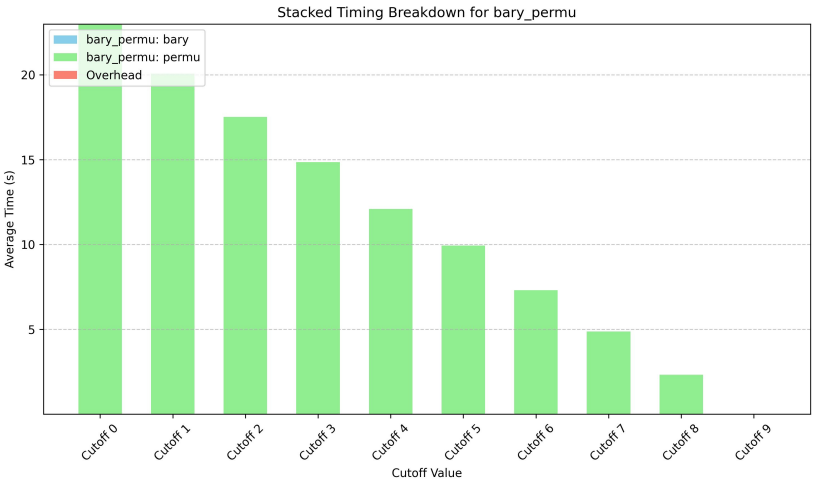

Fig. A1. Stacked runtime of Barycenter and Permutation across cutoff values in `bary_permu`, for layer width 7.

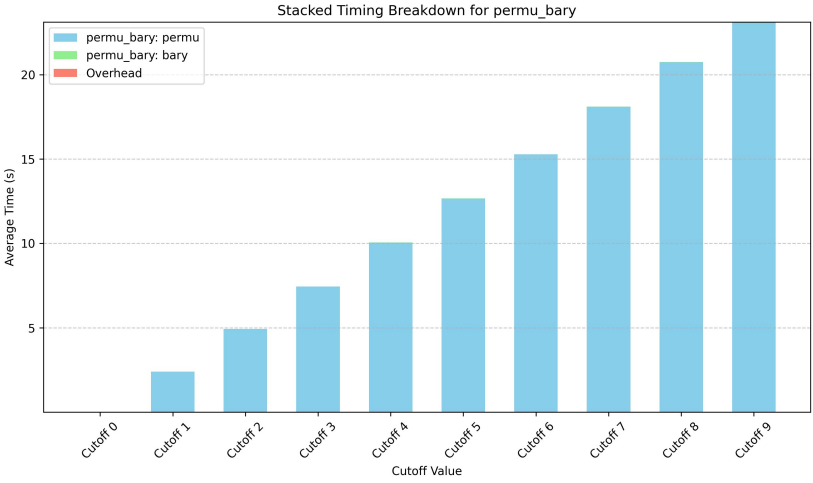

Fig. A2. Stacked runtime of Permutation and Barycenter across cutoff values in `permu_bary`, for layer width 7.

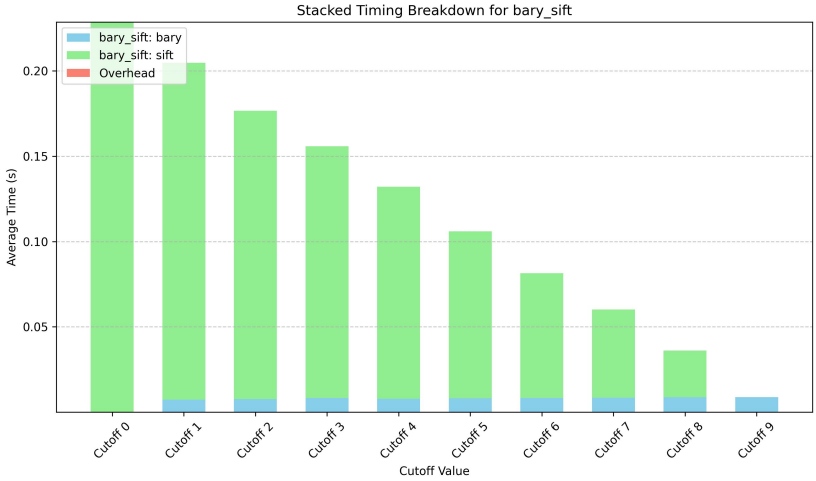

Fig. A3. Stacked runtime of Barycenter and Sifting across cutoff values in `bary_sift`, for layer width 7.

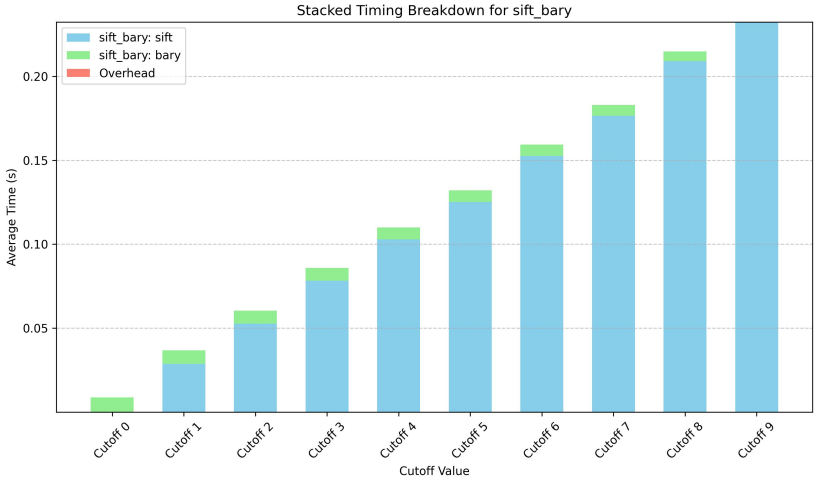

Fig. A4. Stacked runtime of Sifting and Barycenter across cutoff values in `sift_bary`, for layer width 7.

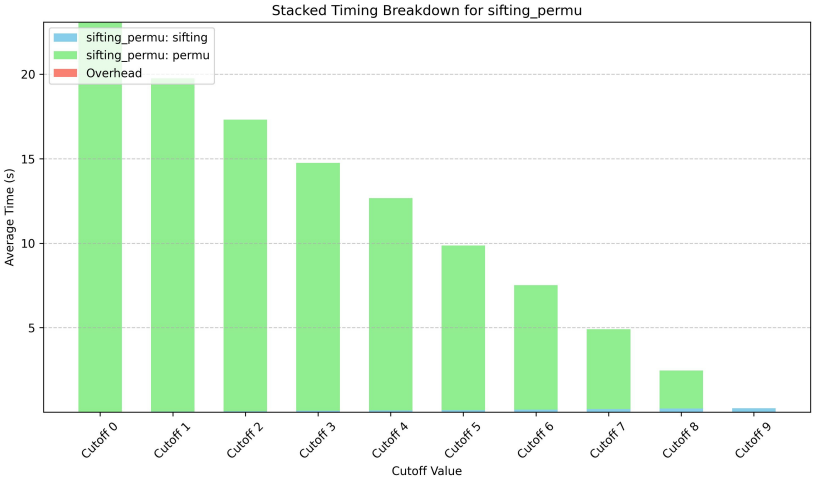

Fig. A5. Stacked runtime of Sifting and Permutation across cutoff values in `sifting_permu`, for layer width 7.

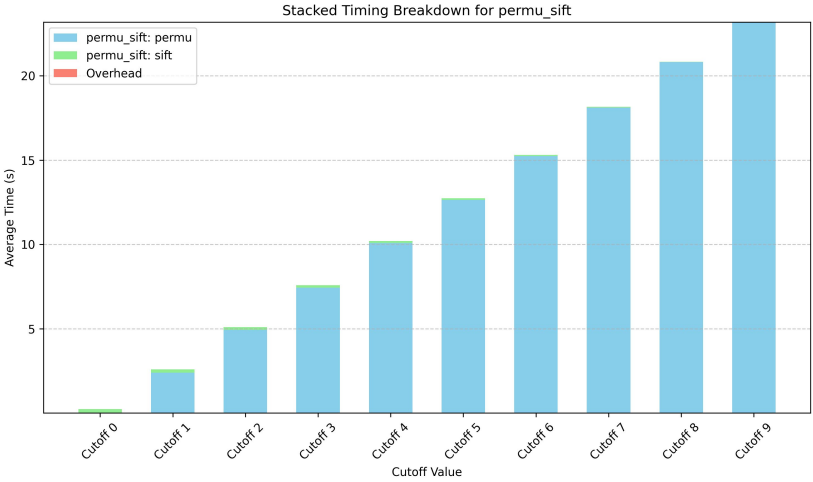

Fig. A6. Stacked runtime of Permutation and Sifting across cutoff values in `permu_sift`, for layer width 7.

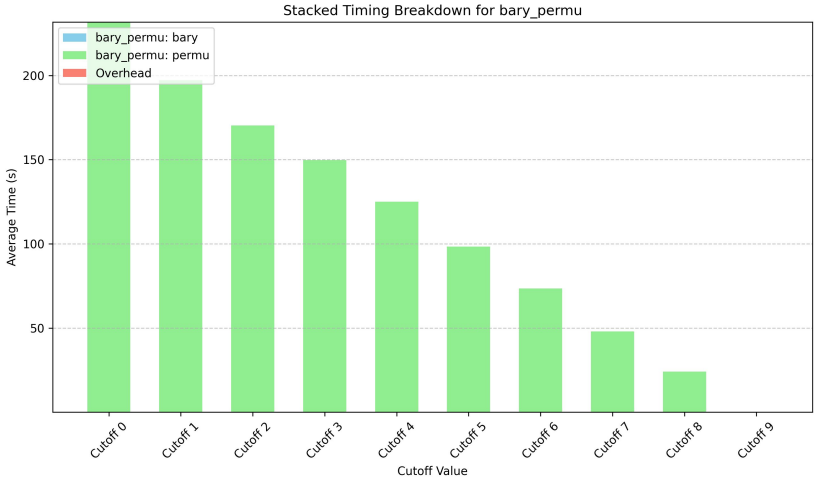

Fig. A7. Stacked runtime of Barycenter and Permutation across cutoff values in `bary_permu`, for layer width 8.

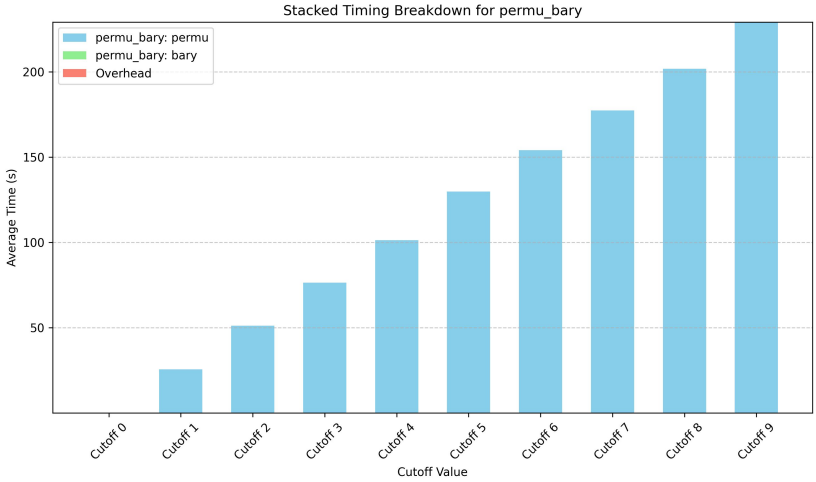

Fig. A8. Stacked runtime of Permutation and Barycenter across cutoff values in `permu_bary`, for layer width 8.

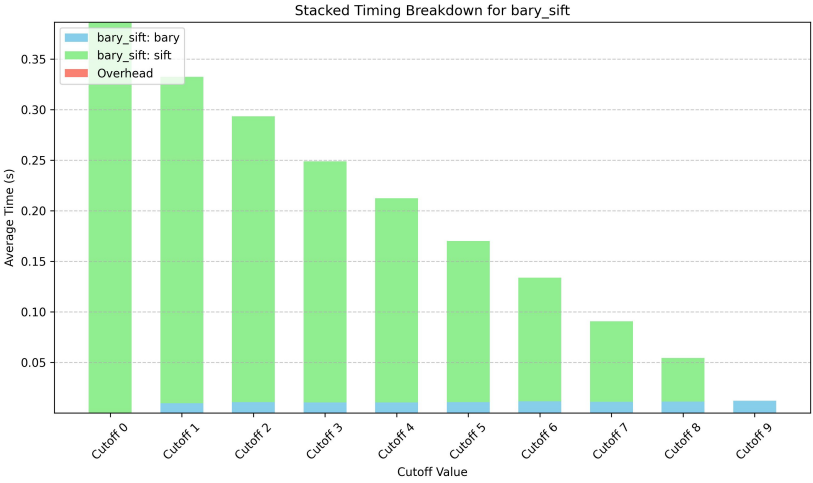

Fig. A9. Stacked runtime of Barycenter and Sifting across cutoff values in `bary_sift`, for layer width 8.

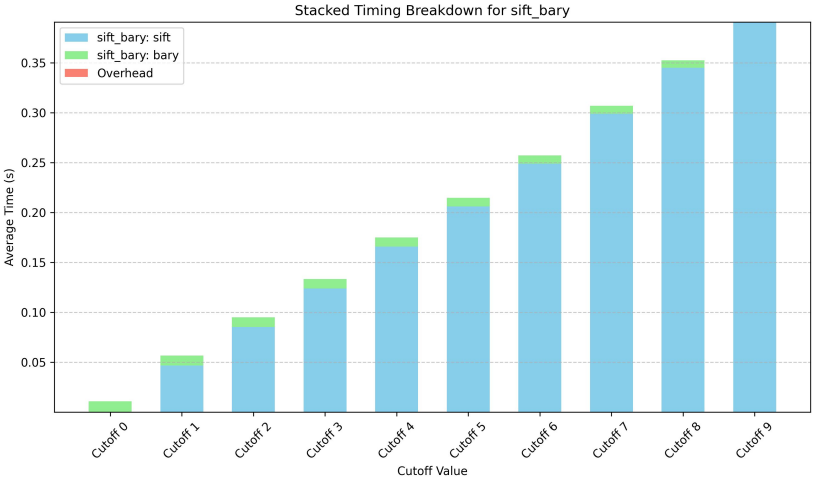

Fig. A10. Stacked runtime of Sifting and Barycenter across cutoff values in `sift_bary`, for layer width 8.

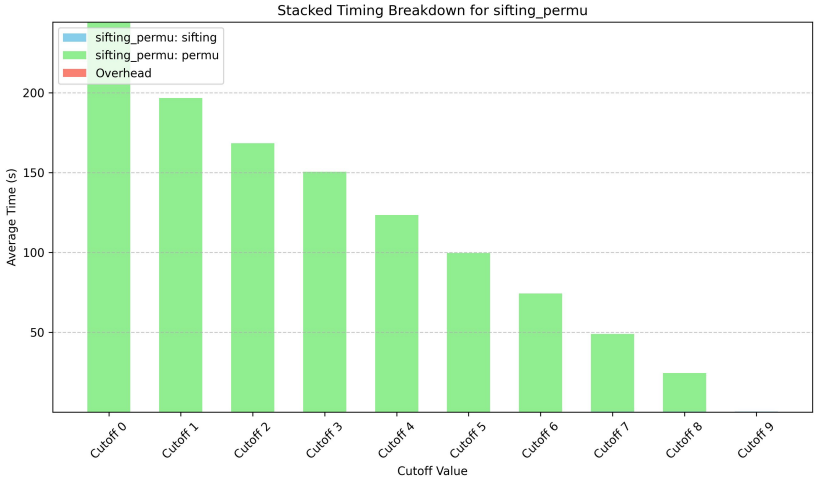

Fig. A11. Stacked runtime of Sifting and Permutation across cutoff values in `sifting_permu`, for layer width 8.

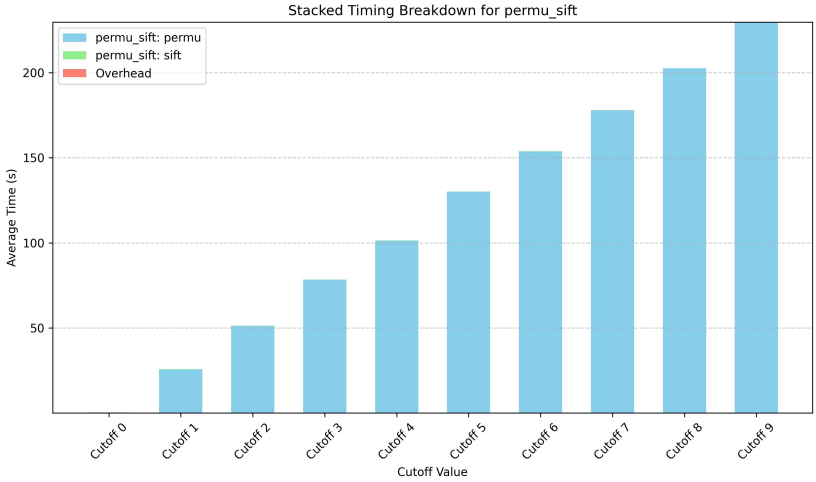

Fig. A12. Stacked runtime of Permutation and Sifting across cutoff values in `permu_sift`, for layer width 8.

Appendix B Layer-by-Layer Sweep Algorithm

Algorithm B1 LayerByLayerSweep(G)

Require: Graph G
Ensure: Optimized graph vertex ordering S

$min\_crossings \leftarrow \infty$ ▷ minimum number of crossings found so far
$current\_crossings \leftarrow 0$
$best\_layer\_struct \leftarrow$ initial layout of G
$current\_layer\_struct \leftarrow$ initial layout of G
$forgiveness\_value \leftarrow 20$
while true **do**
    // Downward sweep: top-down layer-by-layer
    for $i \leftarrow 1$ **to** boundary layer **do**
        Fix vertices in layer $i - 1$
        Permute layer i using a heuristic
        Update $current\_layer\_struct$
        $current\_crossings \leftarrow$ compute crossings of $current\_layer\_struct$
    end for
    // Checkpoint after downward sweep
    if $current\_crossings < min\_crossings$ **then**
        $min\_crossings \leftarrow current\_crossings$
        $best\_layer\_struct \leftarrow current\_layer\_struct$
    else
        $forgiveness\_value \leftarrow forgiveness\_value - 1$
    end if
    if $forgiveness\_value = 0$ **then**
        break
    end if
    // Upward sweep: bottom-up layer-by-layer
    for $j \leftarrow$ boundary layer -1 **to** 0 **do**
        Fix vertices in layer $j + 1$
        Permute layer j using a heuristic
        Update $current\_layer\_struct$
        $current\_crossings \leftarrow$ compute crossings of $current\_layer\_struct$
    end for
    // Checkpoint after upward sweep
    if $current\_crossings < min\_crossings$ **then**
        $min\_crossings \leftarrow current\_crossings$
        $best\_layer\_struct \leftarrow current\_layer\_struct$
    else
        $forgiveness\_value \leftarrow forgiveness\_value - 1$
    end if
    if $forgiveness\_value = 0$ **then**
        break
    end if
end while
return $best\_layer\_struct$