Eurographics Conference on Visualization (EuroVis) 2022
R. Borgo, G. E. Marai, and T. Schreck
(Guest Editors)

Volume 41 (2022), Number 3

Six methods for transforming layered hypergraphs to apply layered graph layout algorithms

Sara Di Bartolomeo [2,1], Alexis Pister[1], Paolo Buono[4], Catherine Plaisant[1,3], Cody Dunne[2], Jean-Daniel Fekete[1]

[1]Université Paris-Saclay, CNRS, Inria, Orsay, France
[2]Northeastern University, Boston, USA
[3]University of Maryland, College Park, USA
[4]University of Bari, Italy

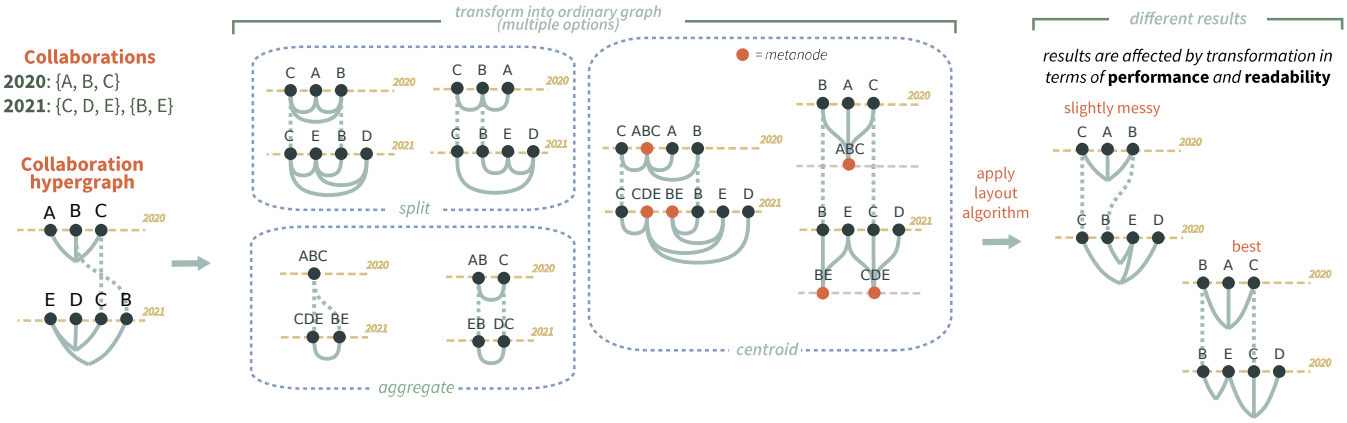


Figure 1: *On the left: Authors A, B, and C collaborate on a paper in 2020. In 2021, C works on another paper with D and E, while B works on a third paper with E. A collaboration network like this can be easily represented as a hypergraph where hyperedges denote co-authoring a paper. We show the network with a layered hypergraph visualization in which every layer corresponds to a year. In order to obtain a readable visualization, though, we need a layout algorithm, and to apply the algorithm to a hypergraph, we transform it into a graph. Yet, there are many methods for transforming a hypergraph into a layered graph, each with different layout readability and computational performance.*

Abstract

*Hypergraphs are a generalization of graphs in which edges (*hyperedges*) can connect more than two vertices—as opposed to ordinary graphs where edges involve only two vertices. Hypergraphs are a fairly common data structure but there is little consensus on how to visualize them. To optimize a hypergraph drawing for readability, we need a layout algorithm. Common graph layout algorithms only consider ordinary graphs and do not take hyperedges into account. We focus on layered hypergraphs, a particular class of hypergraphs that, like layered graphs, assigns every vertex to a layer, and the vertices in a layer are drawn aligned on a linear axis with the axes arranged in parallel. In this paper, we propose a general method to apply layered graph layout algorithms to layered hypergraphs. We introduce six different transformations for layered hypergraphs. The choice of transformation affects the subsequent graph layout algorithm in terms of computational performance and readability of the results. Thus, we perform a comparative evaluation of these transformations in terms of number of crossings, edge length, and impact on performance. We also provide two case studies showing how our transformations can be applied to real-life use cases. A copy of this paper with all appendices and supplemental material is available at osf.io/grvwu.*

CCS Concepts

• ***Human-centered computing*** → ***Graph drawings;*** *Visualization theory, concepts and paradigms;*

1. Introduction

In an ordinary graph, each edge connects exactly two vertices. *Hyper*graphs relax this constraint and allow each *hyper*edge to connect one or more vertices. Given the vast number of real-life applications of hypergraphs in fields as diverse as circuit design [PM07], databases [BFMY83], and machine learning [HZY15], it is particularly important to visualize them effectively. However, the problem of how to visualize a hypergraph still does not have a clear answer, and research on graph layout algorithms for hypergraphs is struggling to keep up with the need to visualize them.

In general, to draw a graph in a node-link style, we take the set of vertices and the set of edges contained in the graph and apply a layout algorithm to them to map every vertex to a coordinate in space. After the coordinates are mapped, we can finally render the graph, using any desired mark for vertices and any desired line, spline, or polyline for edges. With hypergraphs, though, we first need to agree on a way to display a hyperedge. Multiple hyperedge representations have been proposed, such as using polygons or introducing aggregate vertices (a.k.a. metanodes) [FFKS21], and each one has advantages and disadvantages. Indeed, even simple representations such as the one in Figure 1 must first transform the hypergraph into an ordinary graph to draw hyperedges. There, each hyperedge is replaced with a centroid, which is connected by ordinary two-way edges to each of the original incident vertices.

The vast majority of proposed hypergraph layout techniques rely on transforming the hypergraph into a graph [AB17]. More specifically, this paper uses the term *transformation* to refer to a process for creating a graph $G_1 = (V_1, E)$ from a hypergraph $G = (V, H)$. Figure 2 illustrates how this transformation fits into the process for creating a node-link visualization of a hypergraph.

The choice of transformation technique (Figure 2, step 1) can result in wildly different graphs. In particular, the number of vertices, number of edges, and degree distribution of the vertices in the output graph will dramatically impact the time and memory performance of the subsequent layout algorithm 2. If we assume a time complexity of $O(|V_1| * |E|)$—for example, the standard barycentric method used in Sugiyama's layout for layered graphs [STT81] shown in Figure 3—the complexity of the same layout algorithm applied to a hypergraph would be $O(|f(V)| * |g(H)|)$, in which the functions f, g are defined by the chosen transformation. In addition, standard graph layout methods will behave differently in the face of the varied size and connectivity of these transformed graphs and result in different coordinate assignments. After converting back to a hypergraph representation in the final visualization 3, the result may not necessarily be the most readable according to standard readability criteria, such as number of crossings and edge length as outlined in [WPCM02, DRSM15, Pur97].

With this paper, we discuss the challenges involved in the still unexplored field of transformations and graph layout algorithms applied to *layered hypergraphs*. In a *layered graph*, each vertex in the graph is assigned to a layer. Visualizations of them usually align horizontally or vertically the vertices that share a layer and arrange the layers in parallel rows or columns [BETT98, Tam16]. The definition for *layered hypergraph* is identical, substituting hypergraph for graph. Visualizing them, though, is not as straightforward.

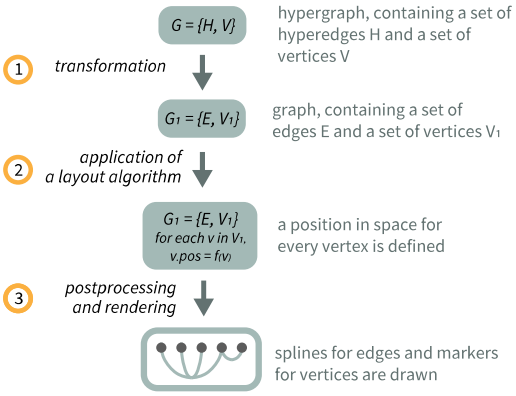


Figure 2: *Steps to be taken from hypergraph defintion to drawing.*

Layered graphs are widely studied and used, for instance, for Sankey diagrams [KS98, MBT*20], neural networks [CMJK20], and SQL visualizations [DBRGD21, LZD*20, GDR22]. Layered hypergraphs are particularly useful for modeling dynamic social networks, as in PAOHvis [VBP*21]. Valdivia et al. did not address how transformations affect the final layout, which led to issues scaling to larger hypergraphs, but they heavily inspired our direction. Our motivating case study is likewise the representation of collaborations between groups of researchers over time, shown in Figure 1. The methods we propose are not meant to replace PAOHvis or other similar hypergraph representations, but rather to be used in combination with them. We take special interest in *dynamic* hypergraphs, in which every hyperedge has an associated time step. If we consider every time step t as a layer in the hypergraph, and assign vertices to layers based on their t, we can consider this style of dynamic graphs as a special case of layered hypergraphs.

Our proposed process for visualizing layered hypergraphs (Figure 2) is to 1 transform the layered hypergraph into a layered graph, then 2 apply a traditional layered graph layout algorithm (e.g., [STT81, GKNV93]), and, ultimately, 3 post-process the layout and visualize the result. The focus of this paper is on the transformation step 1—and what effect it has on the successive step.

We analyze six methods: **split-clique**, **split-path**, **aggregate-collapse**, **aggregate-summarize**, **centroid-within-layer**, **centroid-across-layer**. These methods were selected based on pre-existing techniques [AB17, Kap10, PT11] and variations over them. For each of these methods, we are interested in comparing the impact on performance and quality of the final layouts. In particular, we contribute:

- Six hypergraph-to-graph transformation algorithms and a comparison of their benefits and drawbacks for laying out hypergraph visualizations using the barycentric method;
- A free and open-source implementation of the transformations;
- Two case studies using real data demonstrating the practical utility of these transformations; and
- Benchmark datasets, performance metrics, and computational results comparing the six transformation approaches.

A copy of this paper with all appendices, supplemental material, and the implementation can be found at osf.io/grvwu.

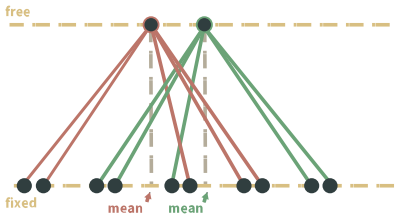


Figure 3: *One iteration of the barycentric method over two layers. Every vertex on the free layer is positioned according to the position of the mean of its neighboring vertices on the fixed layer. The process is repeated over many iterations, alternating the direction in which layers are traversed until the crossings stop improving.*

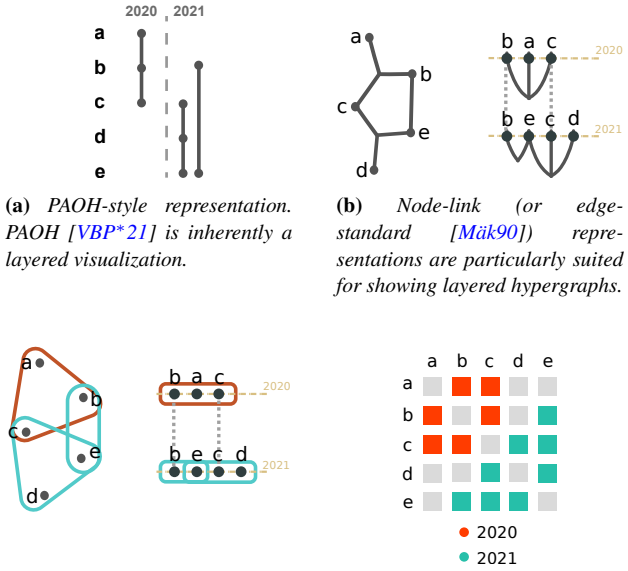
2. Background

Graph layout algorithms: A graph layout algorithm maps vertices and edges in a graph to coordinates in space. Graph layout algorithms have a number of different purposes and use cases, but their general objective is to improve the readability of a node-link visualization by optimizing a number of criteria, either directly or indirectly. A large amount of work has already been done on layout algorithms, giving birth to many popular and widely used algorithms (e.g. [EAD84, STT81, KK89, FR91]).

Perhaps one of the most popular and intuitive of these algorithms is Sugiyama's layered graph layout algorithm [STT81]. It relies on the *barycentric method*, a heuristic that positions every vertex u at the barycenter (or mean position in their layer) of all the vertices v which share an incident edge with u. It is usually applied in an iterative layer-by-layer sweep, where the vertex positions in the preceding layer are kept fixed and used to inform the position of vertices in the current free layer (Figure 3). It is generally regarded as a simple, standard way to minimize crossings and reduce edge length in a layered graph, but using the same approach on hypergraphs requires some adjustments that we explore in this paper.

The design space of hypergraph visualizations: Mäkinen [Mäk90] defines two ways to represent hyperedges: *edge-standard* and *subset-standard*. Both methods represent vertices as points in space, but while edge-standard uses a representation inspired classical graph drawing (e.g. Sander [San04] and Eschbach et al. [EGB04]), subset-standard replaces each hyperedge with an enclosure containing the set of its incident vertices (e.g. Bertault & Eades [BE00]). More recent approaches include subdivision drawings [KvKS09], in which vertices are represented as regions in space and a hyperedge is a set of contiguous subdivisions, matrix-style representations [FAS*21], or PAOHvis [VBP*21], in which vertices are aligned vertically and hyperedges are represented as vertical lines connecting them. More broadly, other creative techniques, such as storyline-style visualizations [PAXP*21, BZSD21], can be included in the design space for hypergraphs as well. Several of these hypergraph representations are compared in Figure 4.

In their 2021 survey, Fischer et al. [FFKS21] use a different classification of hypergraph visualization techniques: node-link, timeline, and matrix. Their survey highlights concerns about the scalability of the different representations for general (not layered) hypergraphs, claiming that the node-link representation is the least



(a) *PAOH-style representation. PAOH [VBP*21] is inherently a layered visualization.*

(b) *Node-link (or edge-standard [Mäk90]) representations are particularly suited for showing layered hypergraphs.*

(c) *Subset-standard-style representations [Mäk90] are not well-suited to be used with layers as they create overlapping.*

(d) *Adjacency matrices hardly allow layers. Additional encodings are needed to show hyperedges.*

Figure 4: *Four representations of the hypergraph in Figure 1.*

scalable. We argue that scalability assessments should take into account the layout method chosen for the representation, and with it, the transformation that needs to be applied to a hypergraph for the layout algorithm to be applicable—which we discuss in this paper.

As our focus is on visualizing layered hypergraphs, the discussions, and examples in this paper use node-link representations. Edge-standard and matrix-based representations are not well-suited to be used with layered graphs (see Figure 4). All the concepts in this paper are relevant to PAOHvis-style representations as well.

Transforming hypergraphs to graphs: Many node-link-based visualization techniques seem to favor transforming a hypergraph into a graph at first [PT11, FVPR22]. After this transformation, an algorithm such as Fruchterman-Reingold's force-directed method [FR91] can be applied to the resulting bipartite graph—examples of this approach are found in [AB17] and [Kap10]. Although intuitive, this method, as well as clique expansion, adds many new edges for the layout computation to address. The bipartite approach adds one edge for each of the N vertices incident to a hyperedge h, while clique expansion adds $N*(N-1)/2$ edges for every h.

Concerned with the poor scalability of clique expansion methods, Ouvrard et al. [OGM17] propose a clustering-based method using Louvain clustering [BGLL08] followed by the ForceAtlas2 layout algorithm [JVHB14]. This particular transformation is similar to what we propose in our *aggregate* class of transformations, but still needs careful analysis when applied: first of all, the heavy lifting of the computation is moved to the clustering algorithm instead of the layout algorithm. Second, it is unclear whether the layout algorithm applied to the aggregate version of the graph will

best represent the underlying topology of the original graph after transforming back to the non-aggregate version.

3. Transformation methods and benchmarks

In the barycentric method, vertices are arranged in their layer according to the mean position of their neighbors [STT81]. The process is repeated over many sweeps back and forth across the layers. This usually results in fewer crossings and reduced edge length. Being designed for ordinary graphs, the barycentric method does not directly apply to hypergraphs and requires caution when handling hyperedges. One can imagine ways to adapt layout algorithms to work with hyperedges: one option that comes to mind is to consider hyperedges as connections between all the pairs of vertices incident to a hyperedge, thus replacing the hyperedge with a clique between the vertices, but this could skew the results giving excessive weight to a single hyperedge. Another option could be to introduce a centroid for every hyperedge and connect all the vertices incident to the hyperedge to the centroid, but this is going to affect the performance of the layout algorithm. As the barycentric method deals with layered graphs, all the methods we present assume a pre-existing layering in the hypergraph.

All these approaches ultimately transform the hypergraph into a graph that a standard layout algorithm can be applied to. However, deciding which transformation to apply requires careful consideration, as they all affect results and performance in different ways. In this section, we analyze six transformations (step ① in Figure 2), divided into three classes, and compare them in terms of impact on computational performance and the readability of the visualization (as defined by metrics discussed in Section 3.4.1).

Each method contains a description, a visual example, and a small discussion including how it affects the performance. The purpose of every method is to transform a hypergraph into a graph. After sorting the graph through a layout algorithm, we might want to revert the representation back to the original representation for display through a post-processing step [YPP21]. Each class of methods contains a brief description of how to post-process the graph, with pseudocode given Appendix A.

Table 2 illustrates an overview of the complexity of the transformations, their effect on the complexity of the barycentric method, and the post-processing cost. Table 1 contains definitions of the notation used in the formulas.

3.1. Split methods

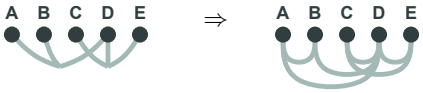
3.1.1 Split-clique: This first method replaces every hyperedge h_i with edges between every pair of vertices incident to h_i. The result of the transformation of a hyperedge effectively forms a clique between all vertices incident to h_i. In the example below, hyperedge *ABC* becomes edges *AB*, *BC*, and *AC*, while hyperedge *CDE* becomes *CD*, *DE*, and *CE*.

$$E = \{e : v_i, v_j \in I(e) \quad \forall v_i, v_j \in I(h_k),\ \forall h_k \in H\} \tag{1}$$

$$V_1 = V \tag{2}$$

Definitions	
$G = (V, H)$	A hypergraph containing a set of vertices $v_i \in V$ and a set of hyperedges $h_i \in H$.
$G_1 = (V_1, E)$	A graph resulting from one of our proposed transformations, containing a set of vertices $v_i \in V$ and a set of edges $e_i \in E$.
$\ell_0, ..., \ell_k \in L$	Layers in G. Each layer contains a set of vertices $v_i \in \ell_k$.
$N(v_i)$	The set of neighbors of vertex v_i.
$I(h_i)$	The set of vertices incident to hyperedge h_i. Can be used for edges as well.
$a(h_i)$	Aggregate vertex for hyperedge h_i.
$c(h_i)$	Centroid of hyperedge h_i.
$x(v_i)$	Horizontal position of vertex v_i.

Table 1: *Definitions and notation used throughout the paper*

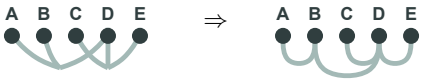


Discussion: Split-clique does not add vertices but adds many edges, and the number of edges scales up very fast depending on the number of vertices incident to a hyperedge. This will naturally worsen computational performance. Moreover, as every hyperedge h becomes $|I(h)| * (|I(h)| - 1)/2$ edges, the transformation might have an unintended effect on the influence of hyperedges over the layout algorithm—if every edge has the same weight, vertices in cliques will end up trying to be as close as possible. The problem can be mitigated by assigning to the newly created edges a weight equal to the weight of the hyperedge divided by the number of newly created edges.

3.1.2 Split-path: Every hyperedge h_i is split into $|I(h_i)| - 1$ edges, in which $|I(h_i)|$ is the number of vertices incident to h_i, forming a single path between all the incident vertices. In the example, hyperedge *ABC* becomes edges *AB* and *BC*, while hyperedge *CDE* becomes *CD* and *DE*.

$$E = \{e : v_i, v_{i+1} \in I(e), \forall v_i \in I(h_k), \forall h_k \in H\} \tag{3}$$

$$V_1 = V \tag{4}$$



Discussion: This is an attempt at reducing the previous method's side effects on performance and hyperedge weights. However, this method does not specify a unique set of edges to create: hyperedge *ABC* could become either *AB* and *BC*, or *AC* and *BC*. Refer to Appendix B for a discussion of how the choice of which edges to create influences the output.

Post-processing for split methods: In the case of split methods, it is sufficient to keep the vertices in the same positions obtained through the sorting algorithm and replace the newly-added edges with a hyperedge.

	Transform complexity	Layout complexity	Post-processing complexity
split-clique	$O(\|H\| * \|I\|^2)$	$O(\|H\| * \|V\| * \|I\|^2)$	$O(\|H\| * \|I\|)$
split-path	$O(\|H\| * \|I\|)$	$O(\|H\| * \|V\| * \|I\|)$	$O(\|H\| * \|I\|)$
aggregate-collapse	$O(\|H\|^2 * \|I\|^2)$	$O(\|H\|^3)$	$O(\|V\|^2 * \|C\|! * \|H\|)$
aggregate-summarize	variable—$O(\|I\|^3)$	$O(\|H\| * \|V\|)$	$O(\|V\|^2 * \|C\|! * \|H\|)$
centroid-within-layer	$O(\|H\| * \|I\|)$	$O(\|H\| * \|I\| * (\|V\| + \|H\|))$	$O(\|H\| * \|I\|)$
centroid-across-layer	$O(\|H\| * \|I\|)$	$O(\|H\| * \|I\| * (\|V\| + \|H\|))$	$O(\|H\| * \|I\|)$

Table 2: *Worst-case complexities of all the transformation methods and their impact on the complexity of the barycentric method. In this table, $|I|$ indicates the maximum number of vertices incident to a single hyperedge. The complexity of the barycentric method is defined as $O(|V| * |E|)$ [STT81], and the impact of the transformation is computed by replacing V and E with the corresponding elements after the transformation. "Variable" in the transform complexity of aggregate-summarize is specified because it depends on the chosen summarization algorithm—$O(|I|^3)$ is the one corresponding to the algorithm used in our example [NRS08]. $|C|$ indicates the number of vertices competing for the same position in the post-processing step of aggregate-collapse and aggregate-summarize—in the worst-possible case, all the vertices in the graph might be competing for the same position, but this occurrence would only be happening if all the hyperedges in the hypergraph are incident to all the vertices, thus extremely rare.*

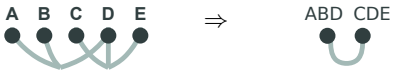
3.2. Aggregate methods

Aggregate methods rely on creating *aggregate vertices* (a.k.a. metanodes) so that the final result does not contain hyperedges. These methods shift the weight onto a preprocessing step needed to compute how to aggregate the graph. The resulting graph is much smaller and the layout faster to compute, but the preprocessing can still be expensive. In addition, aggregate graphs are more complex to transform back into the original hypergraph.

3.2.3 Aggregate-collapse: Every hyperedge h_i is transformed into an aggregate vertex $a(h_i)$, which aggregates all vertices incident to h_i. The original vertices in the hypergraph are removed, replaced by aggregate vertices. We call every vertex stored in an aggregate a *member* of said aggregate. $M(a(h_i)) = I(h_i)$ is thus the set of members of aggregate vertex $a(h_i)$, and corresponds to the vertices incident to h_i. Vertices are allowed to exist as members of more than one aggregate vertex and are stored within the aggregate vertices. We then add edges between aggregate vertices that share member vertices. In other words, all the aggregate vertices $a(h_i)$ and $a(h_j)$ are connected by an edge iff the intersection of vertices incident to their corresponding hyperedges is not empty $I(h_i) \cap I(h_j) \neq \emptyset$. In the example below, aggregate vertices *ABD* and *CDE* are connected by an edge because they share vertex *D*.

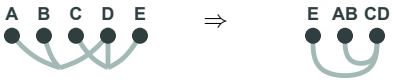
$$V_1 = \{a(h_i) \quad \forall h_i \in H\} \tag{5}$$

$$E = \{e : a(h_i), a(h_j) \in I(e), \; I(h_i) \cap I(h_j) \neq \emptyset, \quad \forall h_i, h_j \in H\} \tag{6}$$



Discussion: Aggregate-collapse replaces all the vertices in the original hypergraph in addition to the hyperedges. Although it successfully removes hyperedges and can reduce the number of elements, the number of new vertices and new edges introduced can in some cases be very large. This depends on the connectedness of the graph: a very connected hypergraph will have a large number of new edges, while a sparsely connected hypergraph will be much smaller than the starting graph.

3.2.4 Aggregate-summarize: Aggregate-summarize uses a graph summarization algorithm to reduce the number of vertices to sort, while also eliminating hyperedges. For our tests, we chose Navlakha et al.'s graph summarization algorithm [NRS08], which aggregates vertices based on the amount of information about the structure of the graph lost when aggregating two vertices. Other summarization techniques can be considered. This aggregation technique does not allow for vertices to be members of more than one aggregate vertex, as opposed to aggregate-collapse, and always produces a smaller graph than the one we started with.



Discussion: The choice of using [NRS08] as an algorithm is just an example of using a summarization algorithm. Different algorithms that have the same purpose can be used as a replacement, and will give different results and different output graph size. The complexity for the transformation we state in Table 2 corresponds to the complexity of [NRS08]. The resulting graph, though, will always be smaller than the original hypergraph, making the worst-case scenario for the barycentric method's complexity remain $O(|V| * |H|)$.

Post-processing for aggregate methods: In the case of aggregate methods, we store in an aggregate vertex a_i every vertex represented by it. When, at the end of the layout computation, these aggregate vertices need to be removed, we can then collect the list of original vertices and assign to each vertex v_i a weight w_i based on the position of the aggregate vertices it belongs to, then sort the vertices in the layer based on the weights. In the case of **aggregate-summarize**, it will be one single vertex, while in the case of **aggregate-collapse**, it will be more than one. In any case, vertices' positions are subject to collisions: more than one vertex might end up having the same w_i, leaving us with doubt on how to order them. This issue can be solved in a post-processing step (whose pseudocode can be found in Appendix A) in which we collect every vertex with the same w_i and test every permutation of the vertices in the set to find the one which produces the least number of crossings.The cost of this process depends on the number of vertices with colliding weights, which can become high (Table 2).

3.3. Centroid methods

This class of methods adds *aggregate vertices* to the graph, which take the place of hypothetical centroids of the vertices incident to a hyperedge. We alternatively refer to this class as *bipartite* methods as they rely on viewing a hypergraph as a bipartite graph. Bipartite graphs are graphs made of two distinct types of vertices where links can only exist between vertices of different types. This type of graph can be seen as an alternative representation of a hypergraph if we consider the hyperedges as vertices of a specific type, which connect the original vertices of the hypergraph.

In centroid methods, we create a new aggregate vertex—that we will refer as centroid vertex—$c(h_i)$ for every hyperedge $h_i \in H$, then replace h_i with edges connecting every vertex incident to h_i to the newly created centroid vertex $c(h_i)$.

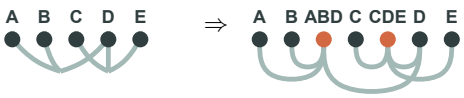
$$E = \{e : c(h_i), n_j \in I(e) \qquad \forall n_j \in I(h_i), \forall h_i \in H\} \quad (7)$$

$$V_1 = V \cup \{c(h_i) \qquad \forall h_i \in H\} \quad (8)$$

Once the replacement is done, as we are dealing with layered graphs we need to consider which layer these newly created centroid vertices should belong to. This decision will affect the layout algorithm. One option is to consider centroid vertices as belonging to the same layer of all the vertices incident to h_i (*centroid-within-layer*), while the other is to consider the centroid vertices as belonging to a separate layer (*centroid-outside-layer*).

3.3.5 Centroid-within-layer:

Here, $c(h_i)$ is inserted into the same layer as $I(h_i)$. This will make the layout algorithm sort the new centroid vertices together with the rest of the vertices.



3.3.6 centroid-across-layer:

Here the centroid vertices are sorted on a different and new pseudo-layer, making the layout algorithm sort vertices and centroid vertices on different layers.

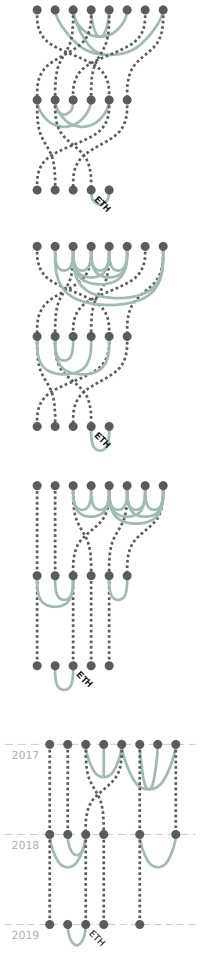
Discussion: Both methods add the same number of new edges and new vertices. Although the transformation is not costly compared to other methods, the addition of new vertices might add a relevant overhead to the computation of the layout algorithm.

Post-processing for centroid methods: As with split methods, we keep the order of the vertices in the same order obtained through the layout algorithm, then we remove the centroids and replace the newly introduced edges with their respective hyperedges. Having computed the layout algorithm with additional edges, the final result might benefit from an additional step that compacts the layout wherever unneeded space was created.

3.4. Multiple layers

All the methods we have seen work with a hyperedge joining vertices within a single layer. How can we extend these ideas to work with hyperedges across multiple layers? We will call hyperedges with all vertices on a single layer *1-layer* hyperedges and hyperedges with vertices on more than one layer *n-layer* hyperedges.

In general, there is no need to change anything in the case of 1-layer hyperedges, as they can go through exactly the same trans-



1 Original
The original hypergraph without a layout algorithm applied. Vertices are in file order.
crossings: 16
edge length: 42
number of hyperedges: 6
number of edges: 10

2 Transform— split-clique
A transformation (here split-clique) is applied to the hypergraph.
number of edges: 23
time to transform: 0 ms

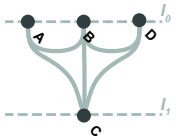
3 After applying layout algorithm
A standard algorithm for computing layouts of layered graphs is applied (here Gansner et al. [GKNV93]), making it much cleaner.
time to sort: 37 ms

4 Final result
The transformation is reversed, going back to the original hypergraph structure. A post-processing step is applied to clean up edge bendiness. The result has fewer crossings and lower edge lengths compared to the starting hypergraph.
crossings: 2
edge length: 14
time to postprocess: 1 ms

Figure 5: *Steps involved to transform and apply a layout algorithm to a layered hypergraph using the* ***split-clique*** *method. This example represents the collaboration network of ETH Zurich: vertices represent research groups in given years, each layer corresponds to a year, and vertices representing the same group in different years are connected with a dashed edge. Gray hyperedges represent collaborations between multiple groups. Collaboration can only happen within a year/layer as publications have only one publication date, so the hyperedges are all horizontal. The same steps would be applied with any other of our proposed methods. A summarized view of the process is shown in Figure 2.*

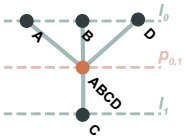
formations seen in the previous section. In collaboration networks such as the ones in our case studies, every collaboration happens in a given year. Therefore, the hypergraph only contains 1-layer hyperedges, plus a number of n-layer ordinary edges which do not require transformation. Figure 5 illustrates the process. More reasoning is needed when dealing with n-layer hyperedges.

In the case of the split class of methods, there is little difference: the new edges generated will just link vertices according to their layers. The example below shows **split-clique** applied to hyperedge h, in which $\{A,B,C,D\} = I(h)$ and $\{A,B,D\} \in \ell_0$, $C \in \ell_1$.



Hyperedges with incident vertices that are more than one layer apart can be further divided through the use of anchors a.k.a. dummy vertices [GKNV93].

In the case of methods with new aggregate vertices being created, instead, we have to reason more on what layers will contain the newly created aggregate vertex. One option is to create a new pseudolayer between the ones containing vertices incident to the hyperedge, as shown in the example below:



Once the layout algorithm has been applied, the pseudolayers can be removed together with the aggregate vertices.

3.4.1. Metrics

In order to characterize one approach as better than another one, we have to define appropriate readability metrics. A number of these metrics have been defined in previous research [WPCM02, DRSM15, Pur97]. Minimizing the number of crossings (also referred to as *planarity* [AB17]) is generally regarded as having the most negative effect on readability [Pur97]. Other metrics take into account symmetry in the graph structure, minimizing edge length and bendiness, optimization of crossing angles, etcetera. In this paper, we evaluated our results based on number of crossings and edge length (which also takes into account edge bendiness), as those are the metrics optimized by the barycentric method.

Number of crossings: We consider two hyperedges h_1 and h_2 to cross if some of the vertices incident to one hyperedge falls between two vertices incident to the other hyperedge, but not all of them together.

$$c(h_1,h_2) = \begin{cases} 0, & \text{if } \forall v_i \in I(h_1), v_j \in I(h_2), \quad x(v_i) \le x(v_j) \\ 0, & \text{if } \forall v_i \in I(h_1), v_j \in I(h_2), \quad x(v_i) \ge x(v_j) \\ 0, & \text{if } \forall\, v_i \in I(h_1),\ \exists\, v_j, v_k \in I(h_2) \\ & \quad \text{s. t. } x(v_j) \ge x(v_i) \ge x(v_k) \\ 0, & \text{if } \forall\, v_i \in I(h_2),\ \exists\, v_j, v_k \in I(h_1) \\ & \quad \text{s. t. } x(v_j) \ge x(v_i) \ge x(v_k) \\ 1, & \text{otherwise.} \end{cases}$$

The case of one edge completely comprised between two vertices of a hyperedge is not considered a crossing, as shown below:

Edge length: We measure the length of a hyperedge h as the maximum horizontal distance between two vertices incident to h. This measure of edge length also takes into account the bendiness of an edge—how much the edge is curved.

$$len(h) = max(abs(x(v_i) - x(v_j))) \quad \forall v_i, v_j \in I(h), \forall h \in H$$

In the examples and benchmarks, one unit of distance is equal to the minimum distance between two vertices.

3.4.2. Benchmark

We ran a benchmark to test the effect on performance and quality of the results of each one of the methods. The hypergraphs used in the benchmark are ego-centered slices from our two case studies: we computed all hypergraphs with one, two, and three degrees of separation for each one of the vertices in the datasets.

After preprocessing each hypergraph with the methods described in Section 3, we applied the barycentric method using our own implementation based on [GKNV93]. (Other layered graph layout algorithms would also be suitable, e.g., the optimal integer linear programming solution in Stratisfimal Layout [DBRGD21].) For each one of the graphs, we measured the time required to complete the computation of the layout and the number of crossings in the final result, using the definition for the number of crossings and edge length described in Section 3.4.1. All the results were computed with node.js on a 2020 Macbook Pro with a quad-core Intel Core i5 and 32 GB of RAM. The results presented in Table 3 show timing, number of crossings, and total number of vertices on hypergraphs containing increasing numbers of hyperedges, split by method.

Implementation: We implemented the six methods in JavaScript to test and compare results, using D3 [BOH11] to produce the visualizations. A live comparison of the transformation methods applied to hypergraphs of different sizes and features can be found at picorana.github.io/hypergraph_layouts, while our implementation, as well as the rest of our supplemental material, can be found at osf.io/grvwu.

4. Case studies

Research institute collaboration network: Inria is a research institute comprised of hundreds of research groups. Throughout the years, these groups collaborate on a number of projects. In our visualization, hyperedges are used to model collaborations between two or more groups. The temporal aspect of the dataset is also important to visualize, as one of the desired tasks is to be able to see changes in the relationships between groups over time. Thus, vertices are aligned in layers, which represent years in the visualization. The color of a vertex represents its scientific domain of research.

Figure 6 (top section) shows the results of the six methods applied to the collaborator of group AVIZ between 2017 and 2020. This section of the dataset contains 69 nodes (average degree: 3.1,

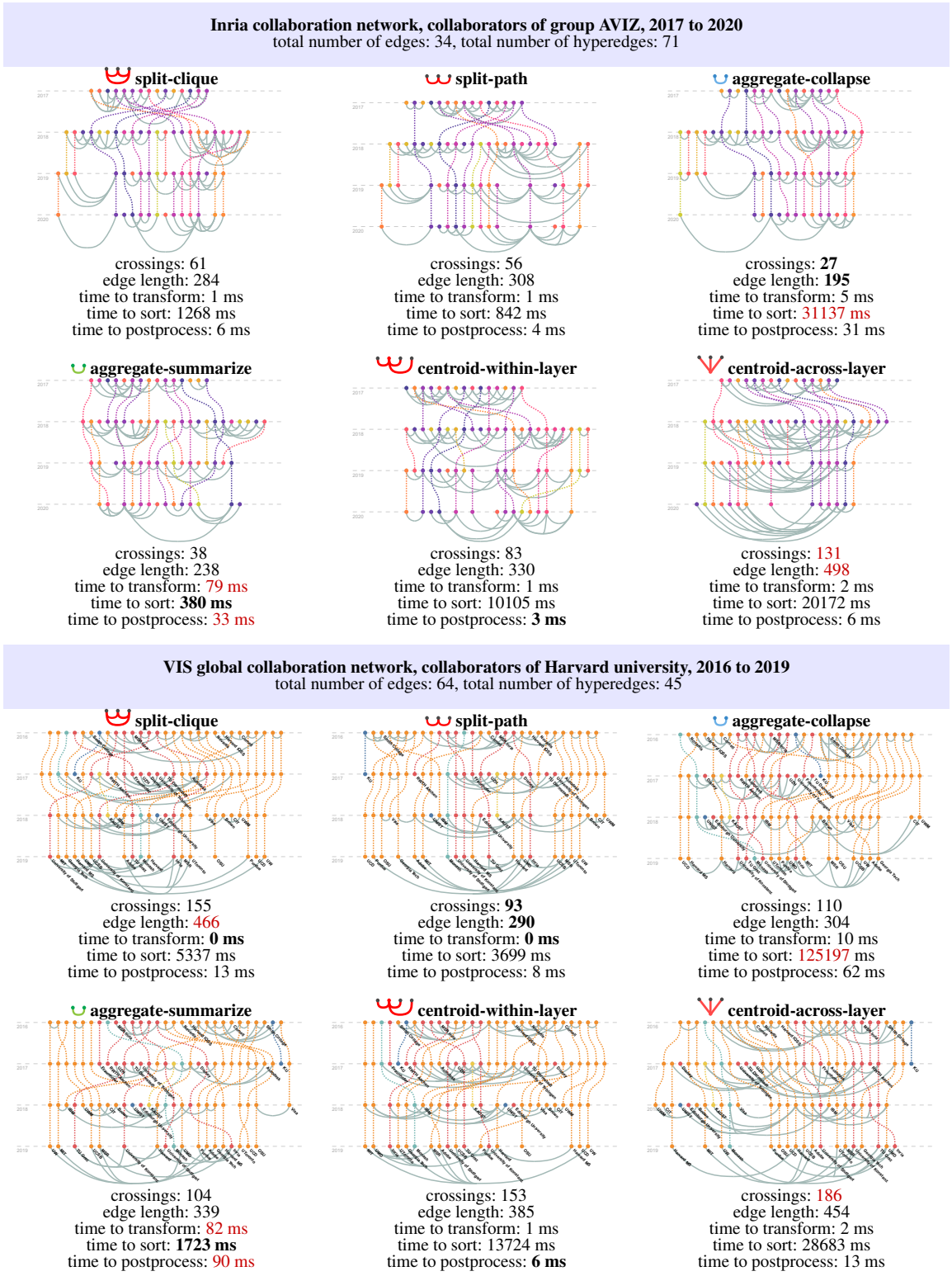


Figure 6: *Comparison of the transformation methods on the Inria and Harvard collaboration networks. Each layer represents a year and each vertex indicates a research group publishing that year. Dashed edges connect the same research group throughout the years. Gray hyperedges represent collaborations between different research groups. All methods show noticeable clustering of the groups, as expected. In the Inria collaboration network, aggregate-collapse produces the best result, while in the VIS collaboration network the best one is obtained with split-path.*

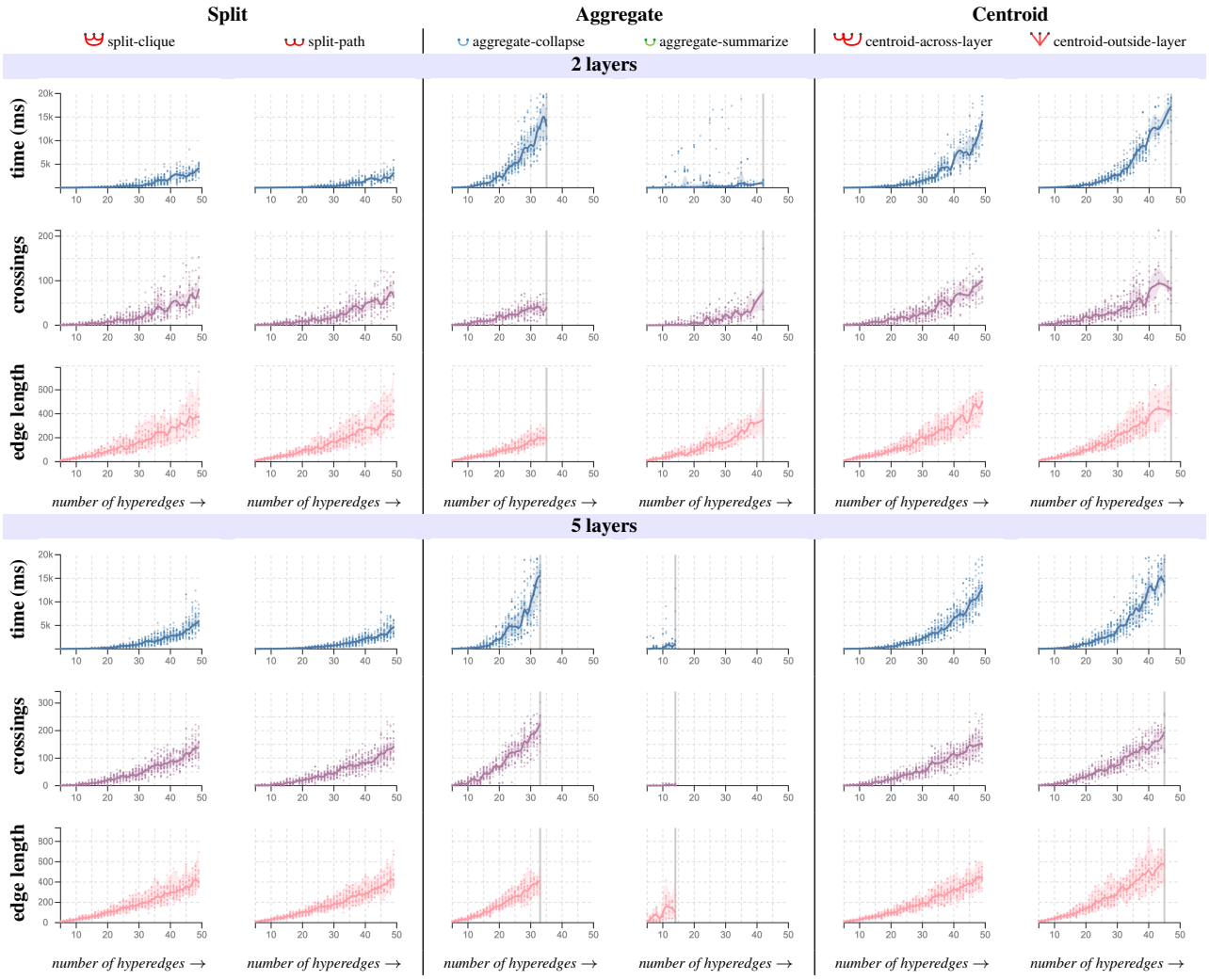


Table 3: *Benchmark comparison. Graphs with an increasing number of hyperedges are tested for processing time when applying the barycentric method, number of crossings, and edge length obtained. For each number of hyperedges, we tested 26 hypergraphs, for a total of 1300 samples. We set a maximum timeout of 20 seconds and stopped the computation whenever we could not complete the layout algorithm within that time. The line represents the median values. Charts are cut wherever we could not compute the layout for at least half the hypergraphs in the dataset with a given number of hyperedges within the timeout. High variance in the aggregate methods (which causes aggregate-summarize to not be able to complete most 5-layer samples) is from instances in which multiple vertices are competing for the same position—resulting in the need for a post-processing step. The fastest and most consistent method in delivering good results is split-path.*

median: 3), 34 edges, and 71 hyperedges (average number of incident nodes: 2.1, median: 2). ∪ aggregate-collapse gives the best results both in term of number of crossings (27) and edge length (195). It is, however, the slowest method, taking 31137 ms to sort. These results match with visually assessing the layout, which appears more readable than with other methods, mainly because there are fewer long edges and vertices which are connected together are often positioned closer. We can therefore better detect patterns in the layout, such as the group of several vertices which collaborate frequently at the right of the figure.

VIS collaboration network: This second case study represents publications at the VIS conference throughout the years, taken from the *Visualization Publication Dataset* [IHK*17]. Similar to our first case study, vertices represent research institutes, and their color corresponds to the country in which they are located. Hyperedges model research collaboration between two or more institutes.

The bottom section of Figure 6 shows the results of the six transformation methods in a section of the full dataset, which includes all nodes up to 2 degrees of separation from Harvard University. The section contains 103 nodes (average degree: 2.4, median: 2), 64 edges, and 45 hyperedges (average number of incident nodes: 2.5, median: 2). This time the ∪ split-path gives the best metrics with a number of crossings of 93 and edge length of 290. It was also the second-fastest method after ∪ aggregate-summarize. We

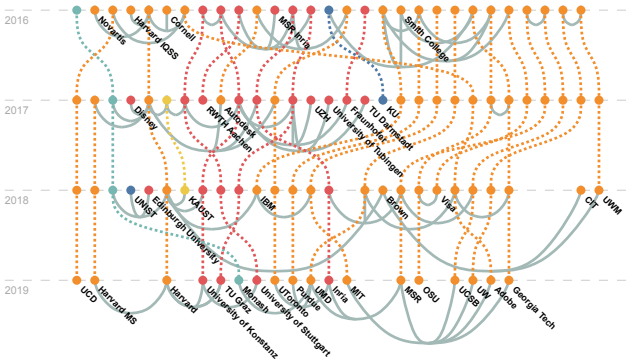


Figure 7: *Collaborations in papers published at VIS between a set of universities—in particular, these are the collaborators of Harvard University and the collaborators of the collaborators (up to 2 degrees of separation). This figure shows the result of the application of ∪ aggregate-collapse.*

can clearly see that there are fewer crossings in the resulting layout, allowing us to better detect patterns. When the hyperedges are packed together, we can see that there are fewer collaborations each year. We can, however, detect a group of vertices in the middle of the layout which seems to be in more collaborations than the rest of the hypergraph, and are often collaborating between them.

5. Discussion

Our benchmarks show some interesting takeaways: although the most intuitive of the two split methods seems to be the first one, ∪ split-clique, the results of the benchmarks show that the difference in resulting crossings between the two is not large, but performance improves considerably with the second method, ∪ split-path. The difference between the two becomes especially evident wherever hyperedges with a large number of incident vertices are transformed, as they create large cliques.

∪ Aggregate-collapse produces good results, but the processing time required increases quickly—as the number of new edges created increases steeply with the connectedness of the hypergraph. It works well for small hypergraphs only. Conversely, ∪ aggregate-summarize transforms the hypergraph into a smaller graph compared to the one we start with, leading to faster mean times, but has a high variance in the results. Indeed, it requires a post-processing step to move vertices with colliding positions, which is costly in cases where several vertices are competing for the same position. This effect becomes very noticeable in our benchmark (Table 3), where a high variance in computing times shows up for aggregate-summarize, and it is even more evident for 5-layer graphs, where collisions prevent the computation from terminating quickly even for relatively small graphs.

Among centroid methods, ∪ centroid-within-layer is the one that performs best, while still being slower than ∪ split-path. ∨ centroid-outside-layer performs worse than ∪ centroid-within-layer, but ends up obtaining fewer crossings.

As a general recommendation, we believe ∪ split-path to be the best option to try first, due to its simplicity, efficiency, and consistency in processing times. In any case, the structure of the hypergraph should be taken into account when choosing the transformation method, giving particular attention to the average hyperedge degree (the number of nodes incident to a hyperedge) and the number of hyperedges. Indeed, cases with a high average edge degree should avoid methods that scale steeply in complexity with the average degree (such as ∪ split-clique and ∪ aggregate-collapse), and cases with a large number of hyperedges should avoid methods that scale quadratically with the number of hyperedges (such as ∪ centroid-within-layer and ∨ centroid-across-layer). The post-processing step is optional, but it should be a concern if methods such as ∪ aggregate-collapse and ∪ aggregate-summarize are used, as it can become expensive quickly based on the number of colliding nodes.

It is important to keep in mind that our considerations and results are based on the barycentric method, which might be limiting for larger graphs. Our choice was motivated by the method's popularity in the graph drawing community, and thus its utility as a general case study. While methods adjacent to the barycentric method will give similar results to our transformation methods, different layout algorithms might require different considerations. Our main intention with this study was to open up a larger conversation on how we apply layout algorithms to hypergraphs, and focusing on a widespread layout algorithm such as the barycentric method gave us a representative example to discuss the outcomes of choosing a transformation method over another one.

6. Conclusion and future work

We claimed that in order to apply a standard layout algorithm to a layered hypergraph, the hypergraph must be transformed into a graph. We proposed six different transformation techniques, then studied their effects on the performance and quality of the results, using the barycentric method as a layout algorithm. We measured the number of crossings, edge length, and impact on the computational performance of the barycentric method used in combination with all the proposed transformation techniques. We found that no method gives a generally better solution, but some methods have a better trade-off between computational performance and readability of the resulting visualization. An exploration of how these transformations interact with layout algorithms other than the barycentric method, and an exploration of the effectiveness of the methods on hypergraphs with no layers, are left for future work.

7. Acknowledgments

This work was supported in part by the U.S. National Science Foundation (NSF) under award number IIS-2145382 and by the DATAIA Institute.

References

[AB17] ARAFAT N. A., BRESSAN S.: Hypergraph drawing by force-directed placement. In *Database and Expert Systems Applications* (Cham, 2017), Benslimane D., Damiani E., Grosky W. I., Hameurlain A., Sheth A., Wagner R. R., (Eds.), Springer International Publishing, pp. 387–394. doi:10.1007/978-3-319-64471-4_31. 2, 3, 7

[BE00] BERTAULT F., EADES P.: Drawing hypergraphs in the subset standard. In *Proceedings of the 8th International Symposium on Graph Drawing* (Berlin, Heidelberg, 2000), GD '00, Springer-Verlag, p. 164–169. doi:10.5555/647552.760489. 3

[BETT98] BATTISTA G. D., EADES P., TAMASSIA R., TOLLIS I. G.: *Graph Drawing: Algorithms for the Visualization of Graphs*, 1st ed. Prentice Hall PTR, USA, 1998. 2

[BFMY83] BEERI C., FAGIN R., MAIER D., YANNAKAKIS M.: On the desirability of acyclic database schemes. *Journal of the ACM 30*, 3 (July 1983), 479–513. doi:10.1145/2402.322389. 2

[BGLL08] BLONDEL V. D., GUILLAUME J.-L., LAMBIOTTE R., LEFEBVRE E.: Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment 2008*, 10 (oct 2008), P10008. doi:10.1088/1742-5468/2008/10/p10008. 3

[BOH11] BOSTOCK M., OGIEVETSKY V., HEER J.: D3 data-driven documents. *IEEE Trans. Vis. Comput. Graph. 17*, 12 (Dec. 2011), 2301–2309. doi:10.1109/TVCG.2011.185. 7

[BZSD21] BARTOLOMEO S. D., ZHANG Y., SHENG F., DUNNE C.: Sequence braiding: Visual overviews of temporal event sequences and attributes. *IEEE Trans. Vis. Comput. Graph. 27*, 2 (2021), 1353–1363. doi:10.1109/TVCG.2020.3030442. 3

[CMJK20] CHATZIMPARMPAS A., MARTINS R. M., JUSUFI I., KERREN A.: A survey of surveys on the use of visualization for interpreting machine learning models. *Information Visualization 19*, 3 (2020), 207–233. doi:10.1177/1473871620904671. 2

[DBRGD21] DI BARTOLOMEO S., RIEDEWALD M., GATTERBAUER W., DUNNE C.: Stratisfimal layout: A modular optimization model for laying out layered node-link network visualizations. *IEEE Trans. Vis. Comput. Graph.* (2021), 1–1. doi:10.1109/TVCG.2021.3114756. 2, 7

[DRSM15] DUNNE C., ROSS S. I., SHNEIDERMAN B., MARTINO M.: Readability metric feedback for aiding node-link visualization designers. *IBM Journal of Research and Development 59*, 2/3 (2015), 14:1–14:16. doi:10.1147/JRD.2015.2411412. 2, 7

[EAD84] EADES P.: A heuristic for graph drawing. *Congressus Numerantium vol.42* (1984), 149–160. URL: https://ci.nii.ac.jp/naid/10000023432/en/. 3

[EGB04] ESCHBACH T., GÜNTHER W., BECKER B.: Orthogonal hypergraph routing for improved visibility. In *Proceedings of the 14th ACM Great Lakes Symposium on VLSI* (2004), GLSVLSI '04, p. 385–388. doi:10.1145/988952.989045. 3

[FAS*21] FISCHER M. T., ARYA D., STREEB D., SEEBACHER D., KEIM D. A., WORRING M.: Visual analytics for temporal hypergraph model exploration. *IEEE Trans. Vis. Comput. Graph. 27*, 2 (2021), 550–560. doi:10.1109/TVCG.2020.3030408. 3

[FFKS21] FISCHER M. T., FRINGS A., KEIM D. A., SEEBACHER D.: Towards a survey on static and dynamic hypergraph visualizations. *2021 IEEE Visualization Conference (VIS)* (2021), 81–85. doi:10.1109/VIS49827.2021.9623305. 2, 3

[FR91] FRUCHTERMAN T. M. J., REINGOLD E. M.: Graph drawing by force-directed placement. *Software: Practice and Experience 21*, 11 (1991), 1129–1164. doi:10.1002/spe.4380211102. 3

[FVPR22] FRIDMAN G., VASILIEV Y., PUHKALO V., RYZHOV V.: A mixed-integer program for drawing orthogonal hyperedges in a hierarchical hypergraph. *Mathematics 10*, 5 (2022). URL: https://www.mdpi.com/2227-7390/10/5/689, doi:10.3390/math10050689. 3

[GDR22] GATTERBAUER W., DUNNE C., RIEDEWALD M.: Relational Diagrams: a pattern-preserving diagrammatic representation of non-disjunctive relational queries, 2022. doi:10.48550/ARXIV.2203.07284. 2

[GKNV93] GANSNER E., KOUTSOFIOS E., NORTH S., VO K.-P.: A technique for drawing directed graphs. *IEEE Transactions on Software Engineering 19*, 3 (1993), 214–230. doi:10.1109/32.221135. 2, 6, 7

[HZY15] HUANG J., ZHANG R., YU J. X.: Scalable hypergraph learning and processing. In *2015 IEEE International Conference on Data Mining* (2015), pp. 775–780. doi:10.1109/ICDM.2015.33. 2

[IHK*17] ISENBERG P., HEIMERL F., KOCH S., ISENBERG T., XU P., STOLPER C., SEDLMAIR M., CHEN J., MÖLLER T., STASKO J.: vispubdata.org: A metadata collection about IEEE visualization (VIS) publications. *IEEE Trans. Vis. Comput. Graph. 23*, 9 (Sept. 2017), 2199–2206. doi:10.1109/TVCG.2016.2615308. 9

[JVHB14] JACOMY M., VENTURINI T., HEYMANN S., BASTIAN M.: ForceAtlas2, a continuous graph layout algorithm for handy network visualization designed for the Gephi software. *PLo S One 9*, 6 (June 2014). doi:10.1371/journal.pone.0098679. 3

[Kap10] KAPEC P.: Visualizing software artifacts using hypergraphs. In *Proceedings of the 26th Spring Conference on Computer Graphics* (2010), SCCG '10, p. 27–32. doi:10.1145/1925059.1925067. 2, 3

[KK89] KAMADA T., KAWAI S.: An algorithm for drawing general undirected graphs. *Information Processing Letters 31*, 1 (1989), 7–15. doi:https://doi.org/10.1016/0020-0190(89)90102-6. 3

[KS98] KENNEDY A. B. W., SANKEY H. R.: The thermal efficiency of steam engines. *Minutes of the Proceedings of the Institution of Civil Engineers 134*, 1898 (1898), 278–312. doi:10.1680/imotp.1898.19100. 2

[KvKS09] KAUFMANN M., VAN KREVELD M., SPECKMANN B.: Subdivision drawings of hypergraphs. In *Graph Drawing* (Berlin, Heidelberg, 2009), Tollis I. G., Patrignani M., (Eds.), Springer Berlin Heidelberg, pp. 396–407. doi:10.1007/978-3-642-00219-9_39. 3

[LZD*20] LEVENTIDIS A., ZHANG J., DUNNE C., GATTERBAUER W., JAGADISH H. V., RIDEWALD M.: QueryVis: Logic-based diagrams help users understand complicated SQL queries faster. In *Proc. ACM SIGMOD International Conference on Management of Data* (2020), SIGMOD, p. 2303–2318. doi:10.1145/3318464.3389767. 2

[Mäk90] MÄKINEN E.: How to draw a hypergraph. *International Journal of Computer Mathematics 34* (1990), 177–185. doi:10.1080/00207169008803875. 3

[MBT*20] MOY C., BELYAKOVA J., TURCOTTE A., BARTOLOMEO S. D., DUNNE C.: Just typeical: Visualizing common function type signatures in r. In *2020 IEEE Visualization Conference (VIS)* (2020), pp. 121–125. doi:10.1109/VIS47514.2020.00031. 2

[NRS08] NAVLAKHA S., RASTOGI R., SHRIVASTAVA N.: Graph summarization with bounded error. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2008), SIGMOD '08, Association for Computing Machinery, p. 419–432. doi:10.1145/1376616.1376661. 5

[OGM17] OUVRARD X., GOFF J. L., MARCHAND-MAILLET S.: Networks of collaborations: Hypergraph modeling and visualisation. *CoRR abs/1707.00115* (2017). arXiv:1707.00115. 3

[PAXP*21] PENA ARAYA V., XUE T., PIETRIGA E., AMSALEG L., BEZERIANOS A.: HyperStorylines: Interactively untangling dynamic hypergraphs. *Information Visualization* (Sept. 2021), 1–21. URL: https://hal.inria.fr/hal-03352276, doi:10.1177/14738716211045007. 3

[PM07] PAPA D. A., MARKOV I. L.: Hypergraph partitioning and clustering. *Handbook of Approximation Algorithms and Metaheuristics 20073547* (2007), 61–1. 2

[PT11] PAQUETTE J., TOKUYASU T.: Hypergraph visualization and enrichment statistics: how the EGAN paradigm facilitates organic discovery from big data. *Proceedings of SPIE - The International Society for Optical Engineering 7865* (02 2011). doi:10.1117/12.890220. 2, 3

[Pur97] PURCHASE H.: Which aesthetic has the greatest effect on human understanding? In *Graph Drawing* (Berlin, Heidelberg, 1997), DiBattista

G., (Ed.), Springer Berlin Heidelberg, pp. 248–261. doi:10.5555/647549.728779. 2, 7

[San04] SANDER G.: Layout of directed hypergraphs with orthogonal hyperedges. In *Graph Drawing* (Berlin, Heidelberg, 2004), Liotta G., (Ed.), Springer Berlin Heidelberg, pp. 381–386. 3

[STT81] SUGIYAMA K., TAGAWA S., TODA M.: Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man, and Cybernetics 11*, 2 (1981), 109–125. doi:10.1109/TSMC.1981.4308636. 2, 3, 4, 5

[Tam16] TAMASSIA R.: *Handbook of Graph Drawing and Visualization*, 1st ed. Chapman & Hall/CRC, 2016. 2

[VBP*21] VALDIVIA P., BUONO P., PLAISANT C., DUFOURNAUD N., FEKETE J.-D.: Analyzing dynamic hypergraphs with parallel aggregated ordered hypergraph visualization. *IEEE Trans. Vis. Comput. Graph. 27*, 1 (2021), 1–13. doi:10.1109/TVCG.2019.2933196. 2, 3

[WPCM02] WARE C., PURCHASE H., COLPOYS L., MCGILL M.: Cognitive measurements of graph aesthetics. *Information Visualization 1*, 2 (2002), 103–110. doi:10.1057/palgrave.ivs.9500013. 2, 7

[YPP21] YOUNG J.-G., PETRI G., PEIXOTO T. P.: Hypergraph reconstruction from network data. *Communications Physics 4*, 1 (2021), 135. URL: https://doi.org/10.1038/s42005-021-00637-w, doi:10.1038/s42005-021-00637-w. 4

Appendix A: Post-processing for aggregate methods

The following pseudocode describes the post-processing step we used to reverse aggregate methods. In aggregate methods, when trying to turn the graph back to its original hypergraph representation, two or more nodes might be competing for the same position. For this reason, we test all the permutations of the colliding nodes to figure out which order returns the smallest amount of crossings. Although this can become costly very quickly, the number of competing nodes is usually very small.

Algorithm 1 Post-processing—Reversing aggregation

```
for v of graph.vertices
    a ← aggregate_vertices.filter(a => a.members.includes(v))
    v.w ← avg(y(a))
graph.vertices.sort((v1, v2) => v1.w > v2.w)
cur_crossings ← count_crossings()

for v of graph.vertices
    collisions ← graph.vertices.filter(vertex => vertex.w == v.w)
    for p of permutations(collisions)
        if count_crossings(p) < cur_crossings
            best = p
            cur_crossings ← count_crossings(p)
    apply(best)
```

Appendix B: Discussion on the set of edges being created with split-path

The order in which the path in split-path is built slightly influences the results. We show different ways of splitting hyperedges in Figure 8. One countermeasure is to attempt every possible path for every hyperedge, but that becomes costly quickly. In our experiments, the best results were obtained by starting the path from the vertex with the highest degree from other edges—so that those would end up closer to the edges of the hyperedge. Overall, though, no variation on the path chosen will cause large disruption in the layout, and the variation in the results will not be large with any choice.

Appendix C: Zoom-in on the visualizations of Figure 6

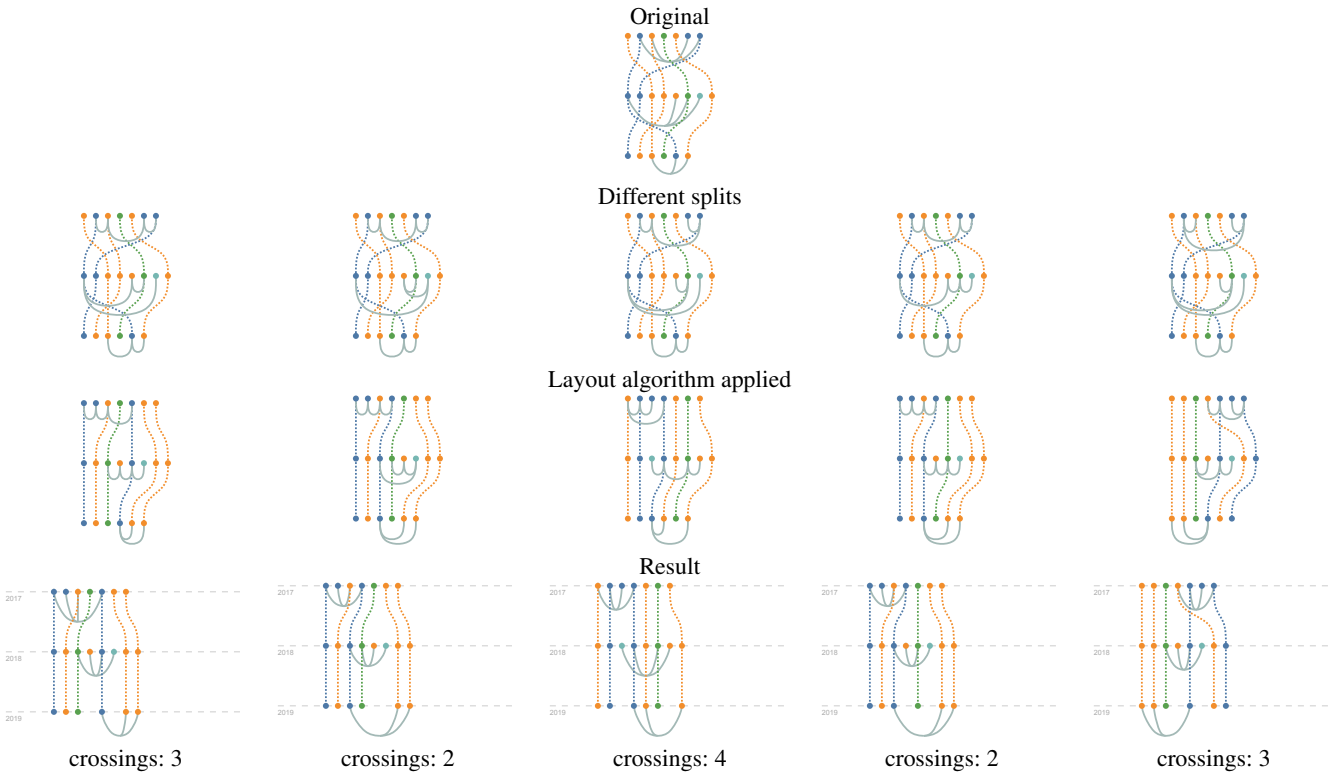
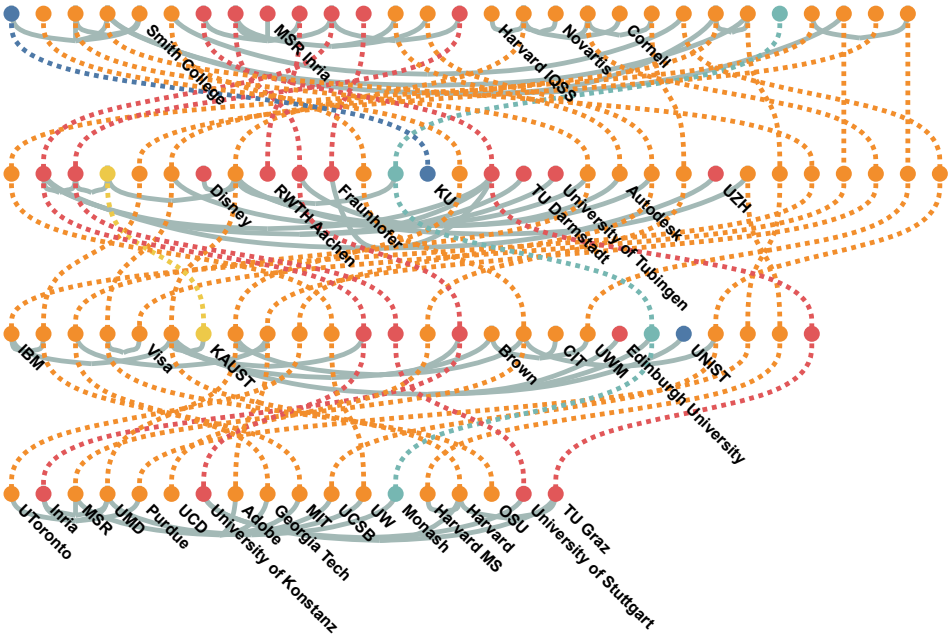
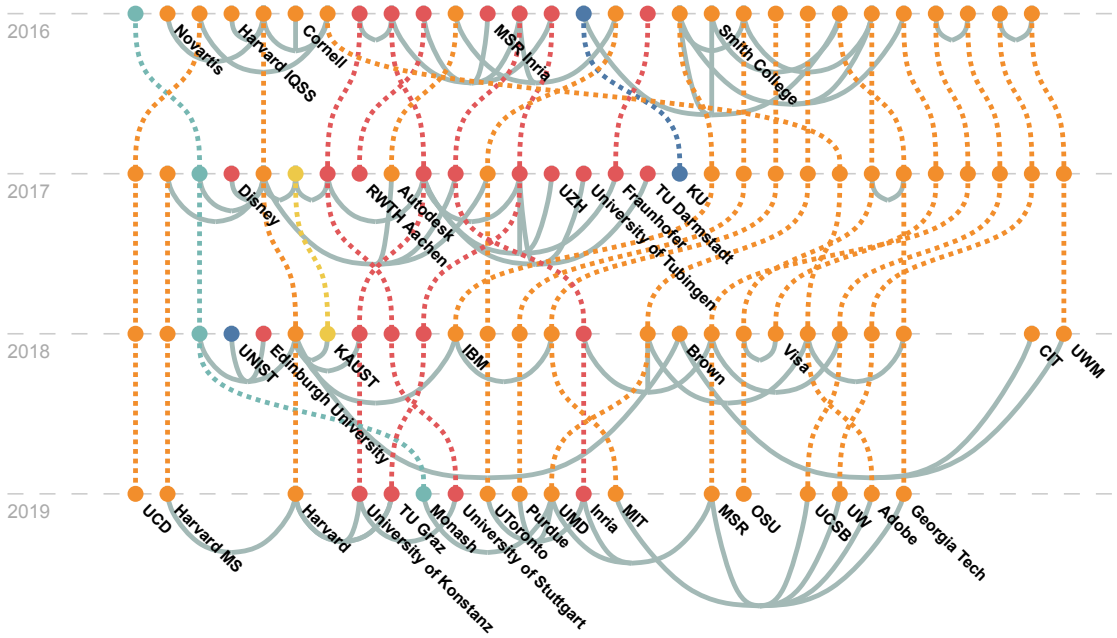


Figure 8: *Different ways of forming a path between vertices involved in a hyperedge, starting from the same graph. This is only a subset of the possible combinations.*



(a) *The original hypergraph, before any transformation or layout algorithm. It contains 45 hyperedges and 64 edges. The ordering of the vertices is the order in which they are read from the dataset with no modifications. This produces 450 crossings and an edge length of 879.*



(b) *The same graph as the image above, after a transformation and the application of the layout algorithm. In particular, this is the result obtained with aggregate-collapse. The higher readability is evidenced by the much lower amount of crossings (110) and much lower edge length (304).*

Figure 9: *Harvard University collaboration network. The graph includes all research groups within two degrees of separation from Harvard between 2016 and 2019. Different colors indicate different continents. The node label containing the name of the research group is only shown on the most recent node corresponding to the given research group.*