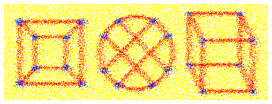
Journal of Graph Algorithms and Applications
http://jgaa.info/ vol. 23, no. 3, pp. 463–498 (2019)
DOI: 10.7155/jgaa.00499

Short Plane Supports for Spatial Hypergraphs

Thom Castermans[1] *Mereke van Garderen*[2] *Wouter Meulemans*[1]
Martin Nöllenburg[3] *Xiaoru Yuan*[4]

[1]TU Eindhoven, the Netherlands
[2]Universität Konstanz, Germany
[3]TU Wien, Vienna, Austria
[4]Peking University, Beijing, China

Abstract

A graph $G = (V, E)$ is a *support* of a hypergraph $H = (V, S)$ if every hyperedge induces a connected subgraph in G. Supports are used for certain types of hypergraph drawings, also known as set visualizations. In this paper we consider visualizing *spatial* hypergraphs, where each vertex has a fixed location in the plane. This scenario appears when, e.g., modeling set systems of geospatial locations as hypergraphs. Following established aesthetic quality criteria, we are interested in finding supports that yield plane straight-line drawings with minimum total edge length on the input point set V. From a theoretical point of view, we first show that the problem is NP-hard already under rather mild conditions, and additionally provide a negative approximability result. Therefore, the main focus of the paper lies on practical heuristic algorithms as well as an exact, ILP-based approach for computing short plane supports. We report results from computational experiments that investigate the effect of requiring planarity and acyclicity on the resulting support length. Furthermore, we evaluate the performance and trade-offs between solution quality and speed of heuristics relative to each other and compared to optimal solutions.

Submitted:	Reviewed:	Revised:	Accepted:
November 2018	January 2019	June 2019	June 2019
	Final:	Published:	
	July 2019	September 2019	
Article type:		Communicated by:	
Regular paper		T. Biedl and A. Kerren	

E-mail addresses: t.h.a.castermans@tue.nl (Thom Castermans) mereke.van.garderen@uni-konstanz.de (Mereke van Garderen) w.meulemans@tue.nl (Wouter Meulemans) noellenburg@ac.tuwien.ac.at (Martin Nöllenburg) xiaoru.yuan@pku.edu.cn (Xiaoru Yuan)

1 Introduction

A *hypergraph* $H = (V, S)$ is a generalization of a graph, in which each hyperedge in S is a nonempty subset of the vertex set V, that is, $S \subseteq \mathcal{P}(V) \setminus \{\emptyset\}$. Furthermore, we assume here that every element $v \in V$ is contained in at least one hyperedge $s \in S$. Hypergraphs arise in many domains to model set systems representing clusters, groups or other aggregations. To allow for effective exploration and analysis of such data, visualization is often used. Indeed, drawing hypergraphs relates to set visualization, an active subfield of information visualization (for more details see the recent survey of Alsallakh et al. [3]). Various methods have been developed to visualize set systems for elements fixed in (geo)spatial positions, such as Bubble Sets [9], LineSets [2], Kelp Diagrams [12] and KelpFusion [21]. These methods make different trade-offs between, e.g., Gestalt theory [26] and Tufte's principle of ink minimization [24] to visually convey the set structures; user studies have been performed to analyze the effectiveness of such trade-offs [21]. Rodgers et al. [23] performed a task-based evaluation of the above methods, but did not discover significant differences between them.

A hypergraph support is an important concept to model the drawing of hypergraphs [15]: a *support* of a hypergraph $H = (V, S)$ is a graph $G = (V, E)$ on the same vertex set V, such that every hyperedge $s \in S$ induces a connected subgraph in G. In other words, for every hyperedge s, the restriction of G to only edges that connect vertices in s is connected and spans all vertices in s. Figures 1(a–b) illustrate a hypergraph with a support. Hypergraph supports correspond to a prominent visualization style for geospatial sets, namely that of connecting all elements of a set using colored links, such as seen in LineSets [2] or Kelp-style diagrams [12, 21] (see also Figure 1(c)).

Thus, finding an embedded support that satisfies certain criteria readily translates into a good rendering of the spatial set system. A "good" support should avoid edge crossings, a standard quality criterion in the graph-drawing literature [22]. Figures 1(b–d) illustrate such plane supports, compared to the nonplanar supports of Figures 1(c,f). Moreover, as per Tufte's principle of ink minimization [24], a support should have small total edge length, where we use $||e||$ to denote the Euclidean length of an edge $e \in E$. Of course, one may argue that edges of the support that are used by multiple hyperedges do not significantly reduce the "ink" as compared to different edges for each hyperedge, and thus multiplicity should be considered. However, we observe that such edges show co-occurrences of elements and thus have a potential added value in the drawing—user studies that establish the validity of this reasoning are beyond the scope of this paper.

The shortest support may contain cycles. To further build on this idea of co-occurrences, one may want to restrict the support to be acyclic—a *support tree* (see e.g. Figures 1(d–e) compared to Figures 1(b,f)). The result of such a restriction is that the common intersection of any number of sets is a connected subgraph in the support. In other words, the intersection is visually shown as one component, rather than scattered across multiple pieces.

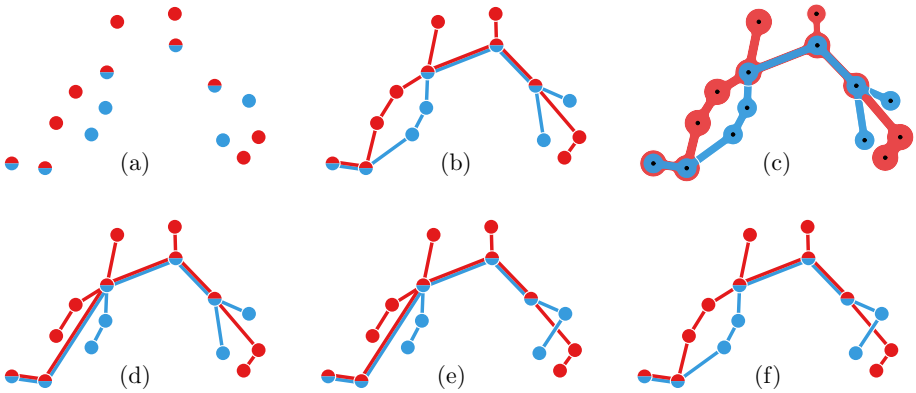

Figure 1: (a) A set system with colors indicating set membership. (b) The shortest plane support of the corresponding hypergraph. (c) A Kelp-style rendering of the set system. (d) The shortest plane support tree. (e) The shortest nonplane support tree. (f) The shortest nonplane support.

In many applications, the vertices have some associated (geo)spatial location, thereby prescribing their positions in the drawing of the support. We focus on this case where vertices have fixed positions in the plane and study supports that are embedded using straight-line edges. Figure 2 shows an example on real-world data of restaurants, similar to those used in [21], illustrating the result of various algorithms introduced in this paper.

Contributions The contributions of this paper are two-fold: on the one hand we fill some gaps in theoretical knowledge about computing plane supports and support trees; on the other hand, we perform computational experiments to gain more insight into the trade-offs on the complexity of the visual artifact for (implicit) support-based set visualization methods. Our focus is on the latter.

In Section 2 we explore computational aspects of the problem and introduce our algorithms. We observe that plane support trees always exist if at least one vertex is contained in all hyperedges, but show that length minimization is NP-hard. Deciding whether a planar support exists otherwise is still an open problem. Moreover, the natural approach to extend a Euclidean minimum spanning tree does not even yield a constant-factor approximation.

In Section 3 we present three algorithms. The first is a heuristic improvement upon a known approximation algorithm. It is based on iteratively computing minimum spanning trees for a hyperedge, where the weights are initially Euclidean but are later modified to promote using edges already in use by spanning trees of other sets. The second algorithm we present is a heuristic algorithm based on local search. The third is an exact algorithm via an integer linear program (ILP). Both the local-search method and the ILP can be configured such that they compute a plane or acyclic support (or both).

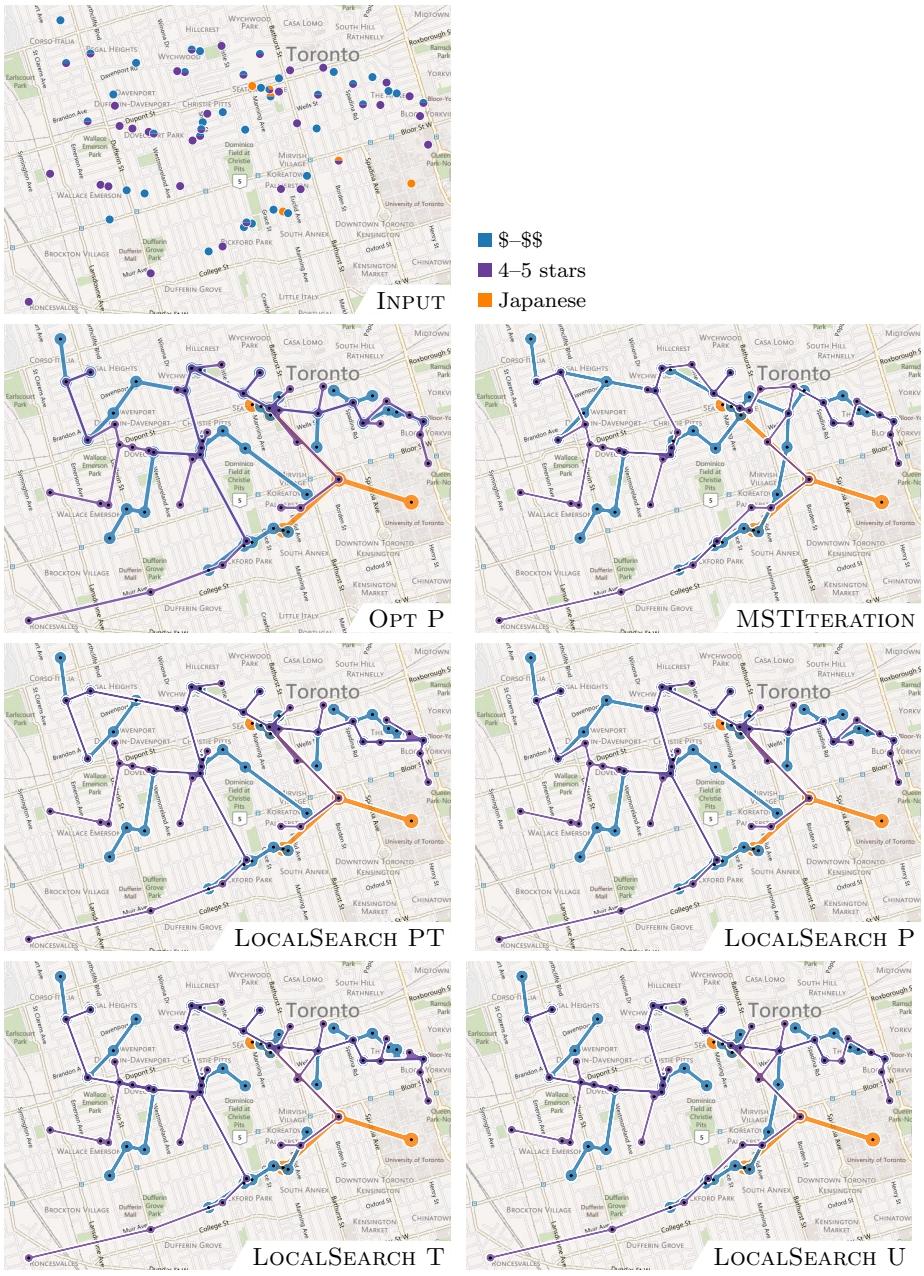

Figure 2: A set system of restaurants in downtown Toronto, visualized using Kelp-style rendering with the supports computed by the various algorithms used in this paper. The addition of the letters P and/or T, indicate the use of constraints forcing the support to be plane and/or a tree. LocalSearch P and Opt P are the same. MSTIteration and LocalSearch U solve the same (unconstrained) problem, but the former results in a support that is approximately 9.5% longer than the support computed by the latter.

In Section 4 we describe the results of two computational experiments.[1] The first experiment compares the performance of the two heuristic algorithms in terms of quality and speed. Whereas the local search achieves better quality, the approximation algorithm is faster. The second experiment compares how well these algorithms perform compared to the optimum, computed via the ILP, and investigates the cost in terms of edge length incurred by requiring planarity or acyclicity. The effect of planarity and acyclicity seems to be predictably influenced by the number of hyperedges and the number of incident hyperedges per vertex, but not by the number of vertices. Moreover, the experiment shows that local search often achieves an optimal result.

Related work Regarding supports for elements with fixed locations, some results are already known. The results of Bereg et al. [5] imply that existence of a plane support tree for two disjoint hyperedges can be tested in polynomial time; this implies the same result for a plane support. This problem has also been studied in a setting with additional Steiner points [4, 13]. Van Goethem et al. [25] enforce a stricter planarity than that of planar supports and investigate the resulting properties for elements on a regular grid, where only neighboring elements can be connected. However, solution length is of no concern in their results.

Without the planarity requirement, existence and length minimization of a (nonplane) support tree for fixed elements can be solved in polynomial time [17, 18]. Hurtado et al. [14] show that length minimization of a support for two hyperedges is solvable in polynomial time. However, for three or more hyperedges this problem is NP-hard [1]. We show that this is in fact NP-hard for two hyperedges if we do require planarity.

Planar supports without fixed elements have also received attention. Johnson and Pollak [15] originally showed that deciding whether a planar support exists is NP-hard; various restrictions have since been proven to be NP-hard [7]. Contrasting these reductions, our hardness result (Theorem 1) requires only two hyperedges, but uses length minimization. Buchin et al. [7] show that testing for a planar support tree with bounded maximum degree is solvable in polynomial time; testing for a planar support tree such that the induced subgraph of each hyperedge is Hamiltonian can also be done in polynomial time [6]. We summarize our results and previously known results in Table 1.

Various set-visualization methods [2, 12, 21] implicitly also compute supports, typically considering a combination of criteria such as length, detour, shape, crossings, and bends in their methods. There has also been some attention for visualizing sets and networks simultaneously, e.g. [11, 23]. Typically, this setting does not prescribe vertex locations.

[1] The source code and data for these experiments, as well as instructions on how to run the experiments, can be found on GitHub: https://github.com/Caster/spssh.

Table 1: A summary of results, with our results in bold. For two sets (or colors), a disjoint relation is represented by ●●, overlap by ◐, and containment by ◎. Condition ⋆ requires a non-empty intersection of all hyperedges.

			Planar		Nonplanar	
	k		existence	length min.	existence	length min.
Tree	2	●●	P [5]	**NP-hard**	yes	trivial
		◐	**yes**	**NP-hard**	yes	trivial
		◎	**yes**	**NP-hard**	yes	trivial
	3+		open (**yes**⋆)	**NP-hard**	P [18]	P [18]
Graph	2	●●	P [5]	**NP-hard**	yes	trivial
		◐	**yes**	**NP-hard**	yes	P [14]
		◎	**yes**	**NP-hard**	yes	trivial
	3+		NP-hard [7]	NP-hard [7]	yes	NP-hard [1]

2 Existence, Bounds, and Complexity

Existence The lemma below gives a sufficient (but not necessary) condition for the existence of a plane support tree. Bereg et al. [5] provide a necessary condition for two hyperedges ($|S| = 2$). The problem remains open for $|S| > 2$.

Lemma 1 *Consider a hypergraph $H = (V, S)$ with no three vertices in V on a line, such that $V_A = \bigcap_{s \in S} s \neq \emptyset$. Then H has a plane support tree that contains the Euclidean minimum spanning tree on V_A as a subtree.*

Proof: We first compute the Euclidean minimum spanning tree (EMST) T on V_A and then connect each vertex in $V \setminus V_A$ by a *new* edge to a closest vertex in V_A, see Figure 3. We argue that the resulting graph is a plane support tree. Obviously, the EMST is a plane tree. An edge that connects a vertex $v \in V \setminus V_A$ to its closest vertex $u \in V_A$ cannot cross an edge wz of T, as we know from the basic properties of the EMST that the circle with diameter $\|wz\|$ contains no other vertex of V_A for every edge wz of T. Therefore u would have larger Euclidean distance to v than w or z, which is a contradiction. Further observe that no two new edges can cross, as such a crossing implies that at least one of the crossing edges would not connect to the closest vertex in V_A. Finally, since there are no three collinear vertices, no added edge will contain a vertex and thus, no two added edges can be overlapping either. As we are attaching only leaves to a tree, the resulting graph remains a plane tree. □

If V_A is empty, one can immediately construct instances that enforce a crossing in *any* support, e.g., an X-configuration of two disjoint hyperedges.

Approximation In a support tree the subgraph induced by V_A must be a connected subtree to satisfy the support property for all hyperedges. Next we

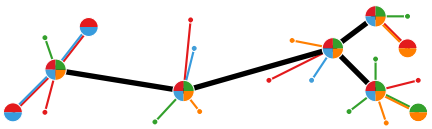

Figure 3: Example of a plane support tree constructed according to Lemma 1. The EMST on V_A is drawn with thick black edges.

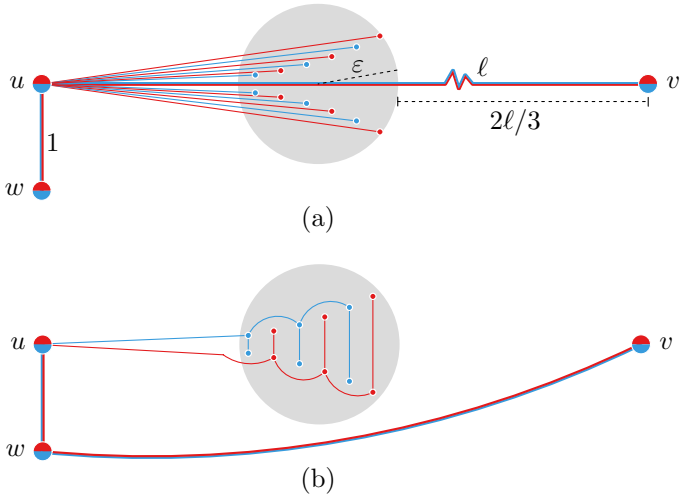

Figure 4: An n-point instance with approximation ratio $\Theta(n)$ if using an EMST on V_A. All edges are straight-line segments; curvature is used to emphasize the effect of the convex chain.

consider using the idea of Lemma 1 to start with a Euclidean minimum spanning tree (EMST) of V_A and extend it to a support tree. If we allow intersections, this leads to an $(\frac{\rho}{2}+1)$-approximation algorithm, where ρ is the Steiner ratio [14]. However, we show below that the planarity requirement can cause the resulting support length to exceed any constant factor of the length of the shortest plane support tree.

Lemma 2 *There is a family of n-vertex hypergraphs $H = (V, \{r, b\})$ with $V_A = r \cap b \neq \emptyset$ such that any plane support of H that includes an EMST of V_A is a factor $\Theta(|V|)$ longer than the shortest plane support tree.*

Proof: The hypergraph family is illustrated in Figure 4. The set $V_A = \{u, v, w\}$ consists of three vertices whose EMST T has length $\ell+1$ and is indicated by the heavier, two-colored edges in Figure 4a. The remaining vertices in $V \setminus V_A$ are indicated in red and blue (indicating membership of r and b) and placed inside a disk of radius ε just left of the midpoint of edge uv. The vertices alternate in colors from left to right and form two mirrored convex chains.

Since edge uv of T splits the vertices in $V \setminus V_A$ and by their placement on convex chains, the shortest extension of T into a plane support tree is to connect every vertex to u (Figure 4a). This yields a total length of the support tree of $\Theta(n) \cdot \ell$. If, however, V_A is connected by a slightly longer tree, the remaining vertices in $V \setminus V_A$ can be joined by two comb-shaped structures as shown in Figure 4b. The resulting plane support tree has a length of $\Theta(1) \cdot \ell$. □

Removing vertex w from the construction in Figure 4, we can similarly show that a plane support tree, which now necessarily includes the edge uv, is a factor $\Theta(n)$ longer than the shortest nonplane support tree.

Corollary 1 *There is a family of n-vertex hypergraphs $H = (V, \{r, b\})$ with $V_A = r \cap b \neq \emptyset$ such that any plane support tree of H is a factor $\Theta(n)$ longer than the shortest nonplane support tree.*

Computational complexity Unfortunately, the problem of finding the shortest plane support and several restricted variants are NP-hard, as captured in the theorem below.

Theorem 1 *Let $H = (V, S)$ be a hypergraph with vertices V having fixed locations in $\mathbb{R}^2$. Let $L > 0$. It is NP-hard to decide whether H admits a plane support with length at most L, even if $S = \{r, b\}$, $r \subseteq b$, and the support is required to be a tree.*

Proof: We first show the reduction for the most restricted case: $S = \{r, b\}$ and $r \subseteq b$, and the support is required to be a tree. We use a reduction from planar monotone 3-SAT [19]. Here, we are given a 3-CNF formula ϕ with n variables $v_1, \ldots, v_n$ and m clauses $c_1, \ldots, c_m$ such that every clause has either three positive literals or three negative literals. Moreover, we are given an embedding of ϕ as a graph, with rectangular vertices for variables on a horizontal line, and clauses as rectangles above or below the line (depending on whether the clause is positive or negative). Vertical edges connect clauses to the variables of their literals.

We must construct a spatial hypergraph $H = H(\phi) = (V, \{r, b\})$ such that $r \subseteq b$. In the remainder of the proof, we assign vertices to either r (red) or b (blue), understanding that every red vertex in r is also a blue vertex in b. Refer to Figure 5 for an illustration of the construction.

First, we place $3(n + 1)$ red vertices using coordinates $(3i \cdot (m + 1), y)$ for integers $i \in [0, n]$ and integers $y \in [-1, 1]$. Furthermore, we place $n \cdot (3m + 2)$ blue vertices using coordinates $(3i(m + 1) + j, 0)$ for integers $i \in [0, n - 1]$ and $j \in [1, 3m + 2]$.

We now place additional blue vertices for each clause c_a. We assume that this clause has positive literals for variable v_i, v_j, and v_k; the construction for clauses with negative literals is symmetric, using negative y-coordinates instead. First, we place $3a+1$ blue vertices from $(3(i-1)(m+1)+3p, 2)$ to $(3(i-1)(m+1) + 3p, 2 + 3a)$ at unit distance, to represent the incidence from c_a to variable

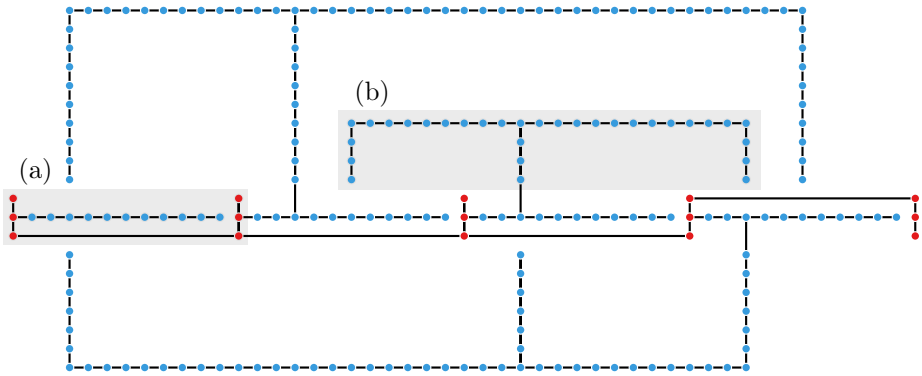

Figure 5: Construction for $\phi = (v_2 \vee v_3 \vee v_4) \wedge (\overline{v_1} \vee \overline{v_3} \vee \overline{v_4}) \wedge (v_1 \vee v_2 \vee v_4)$. Vertices in r and b are red, vertices in b are blue. A plane support tree with length at most L is given in black lines. (a) Representation of variable v_1; the solution sets v_1 to true. (b) Representation of the first clause.

v_i, using the given embedding to determine that c_a is the p^{th} clause incident from above to v_i. Analogously, we place the blue vertices for v_j and v_k. Now, we place further blue vertices at unit distance with y-coordinate $2+3a$ from the leftmost to the rightmost top vertex we just placed.

One clause requires at most $3(3m+1)$ vertices for the variable incidence and less than $3n \cdot (m+1)$ for the horizontal line connecting these. We can now readily measure the length of the minimum spanning tree on the blue vertices of one clause. We use L_a to denote this length; note that L_a is an integer at most $3(3m+1) + 3n \cdot (m+1)$. The value of L that we select is $2(n+1) + 3n \cdot (m+1) + n(3m+2) + 2m + \sum_{a \in [1,m]} L_a$.

This finalizes the construction. It is of polynomial size since we placed $3(n+1)$ red vertices and $n \cdot (3m-2)$ blue vertices for the variables and at most $m \cdot (3(3m+1)+3n \cdot (m+1))$ for the clauses: this is $O(nm^2)$ vertices. Moreover, we claim that our constructed hypergraph admits a plane support tree of length at most L, if and only if ϕ is satisfiable.

Assume we have a plane support tree of length at most L. First, we observe that all points in r must be connected: the minimal way of doing so connects the three vertices with the same x-coordinate and uses one horizontal segment to connect one triplet to the next. This has exactly length $2(n+1)+3n \cdot (m+1)$, corresponding to the first two terms defining L. The minimal way of connecting the blue points inside the variables to the red tree takes length $n(3m+2)$ in total: this is the third term defining L. Finally, to connect the clause vertices, we need length at least L_a per clause, the last term of L. We note that any solution must use these constructions on the blue vertices, since all vertices are at unit distance; other blue vertices are at distance at least 2. However, the support tree is connected: thus it must still have connections from each gadget

to either a red vertex or a blue vertex of a variable. The budget we have for this is $2m$ in total. Since each clause needs a connection of length at least 2, all clauses use exactly length 2. The only vertices within distance 2 of a clause are the three blue vertices of the variables with y-coordinate zero (one of each literal of the clause). Thus, each clause must have exactly one length-2 edge to one of these variable vertices. Since the support tree is plane, this cannot cross the horizontal links used to connect the red vertices. We can now readily obtain a satisfying assignment for ϕ from a plane support tree with length at most L, by looking at which of the two horizontal edges is used to connect the red vertices: if the one at the top is used, that variable is set to false; otherwise, it is set to true.

To prove the converse, assume that we have a satisfying assignment. Using the same reasoning as above, we construct a plane support tree of length L by picking the connecting horizontal edges for the red vertices according to the satisfying assignment: this readily implies that we can connect each clause using a length-2 connection for one of its satisfied literals that does not intersect the horizontal edges for the red vertices of the corresponding variable.

Our proof readily implies that the more general case with $|S| \geq 3$ or $r \not\subseteq b$ is also NP-hard. If we also admit supports that are not a tree, the same construction also works. The converse proof above does not change: we can still find the same support tree with length L. To show that a plane support of length at most L must imply a satisfiable formula, we may observe that any valid support of minimal length must be a tree. Consider a support of minimal length that is not a tree. Then it must have a cycle. If this cycle contains only vertices from r, then we can remove an arbitrary edge. If this cycle contains a vertex from $b \setminus r$, then we remove an incident edge of such vertex. Either case shortens the support while maintaining connected induced subgraphs for r and b (which also includes the vertices of r). This contradicts that the assumed support has minimal length. Thus the minimal-length support must be a tree. □

Note that the construction used in the reduction above is rather degenerate: it uses many collinear vertices with integer coordinates. This helps us bound the complexity of the reduction, that is, to show that we do not need many bits to encode each coordinate. However, the construction does not rely on this degeneracy. Slightly displacing all vertices keeps the structure intact.

3 Algorithms

We now turn to describe three algorithms. The first is an approximation algorithm, the second uses a local-search heuristic and the third computes optimal solutions via Integer Linear Programming.

3.1 Iterative Minimum Spanning Trees

Here we focus on computing short supports without requiring planarity. As described by Hurtado et al. [14], EMSTs can be used to find an approximation

Algorithm 1 MSTIteration(H, σ)

Input: hypergraph $H = (V, S)$ with vertices in the plane, computation sequence σ over S

Output: a support for H

1: Initialize T_s to $\emptyset$ for every $s \in S$
2: Initialize graph $G = (V, \emptyset)$, where each edge has a counter for the number of MSTs T_s that contain it
3: **for** $s \in \sigma$ **do**
4: For every $e \in T_s$, decrease the counter of e by 1; if it reaches 0, remove e from G
5: Compute an MST T_s of s, where edges in G have weight 0, and other pairs of vertices have weight equal to their Euclidean distance
6: For every $e \in T_s$, increase the counter on e by 1 if it already exists in G; otherwise, add e to G and set its counter to 1.
7: **return** G

of the shortest support. In particular, let $H = (V, S)$ be a hypergraph with n vertices and k hyperedges; by computing an EMST for each hyperedge and taking their union, we get a support that is a k-approximation[2] of the shortest support. This algorithm runs in $O(kn \log n)$ time.

Suppose that we compute the EMSTs $T_1, \ldots, T_k$ in that order, for the k hyperedges in S. The final support is the union of these trees: its length is not increased by using an edge in T_i that is already present in some T_j $(j < i)$. Hence, we can consider any pair of vertices that is adjacent in $T_1 \cup \ldots \cup T_{i-1}$ to have distance zero, when computing T_i. This heuristically reduces the length of the resulting support (though the approximation ratio remains the same). However, the order in which hyperedges are considered now matters for the result. To alleviate this issue, we iteratively recompute the minimum spanning trees.

Algorithm We define a *computation sequence* σ of a hypergraph $H = (V, S)$ as a sequence of hyperedges that contains each hyperedge in S at least once. Each item s in the sequence σ represents the computation of the (not-quite Euclidean) MST on the vertices of s, such that edges have weight 0 if they are part of the current support and a weight equal to the Euclidean distance between their vertices otherwise. We use T_s to denote the current MST for hyperedge $s \in S$; the support G is always the union over all T_s. As we compute a spanning tree for each hyperedge, G is a support for H when the algorithm terminates. Algorithm 1 provides pseudocode for the described algorithm.

Efficiency Implementing G with adjacency lists, we use $O(nk)$ storage as each of the k trees has $O(n)$ edges. To compute T_s, we use Lemma 3 below

[2] One can actually do slightly better, by computing spanning trees on the intersection of two hyperedges, yielding roughly a $(0.8k)$-approximation [14] for nonplanar supports.

to conclude that there are $O(nk)$ candidate edges, ensuring that Prim's MST algorithm runs in $O(nk + n \log n)$ time. To see that we can determine the weight without overhead, consider all vertices to be indexed with numbers from 1 to n. When adding a vertex u to the current tree in Prim's algorithm, we first process the neighbors of u in G (having a weight 0) and mark that these have been processed in an array using the above mentioned vertex index. Only then do we process all other vertices (having weight equal to the Euclidean distance) that are not marked and are not in the current tree. The total algorithm thus takes $O(|\sigma|(nk + n \log n))$ time and $\Theta(nk)$ space.

In the following, we mention finding the Euclidean MST of a point set, though the Euclidean MST is generally not unique. It being unique assumes either unique distances between all pairs of vertices, or a deterministic way of choosing which edge goes in the MST when multiple have the same minimum weight. The latter can easily be implemented in practice and is as such a reasonable assumption to make.

Lemma 3 *Let P be a point set and $F \subseteq P \times P$. Consider the MST T on P, based on edge weights 0 for edges in F and the Euclidean distance otherwise. Then T is a subset of the union of F and the Euclidean MST on P.*

Proof: Let T' denote the Euclidean MST on P. Assume that MST T has some edge e that is neither in F nor in T'. Since T is a tree, removing e from it partitions the tree into two connected components. By definition, T' contains an edge e' that connects the two components and by assumption $e' \neq e$. Since T' is the Euclidean MST, we know that $\|e'\| < \|e\|$. Since e is not in F, the weight it contributes to T is $\|e\|$ and thus we can find a shorter spanning tree T^*, by replacing e with e' in T. This contradicts that T is the MST, thus proving the lemma. □

Properties ($k = 2$) The main question that arises is how long a computation sequence σ must be such that that the result *stabilizes*, that is, any sequence that extends σ gives a support that has the same total length. We use G_σ to denote the support resulting from computation sequence σ. With Lemma 4, we prove that for $k = 2$, we need to recompute only one hyperedge: sequence $\sigma = \langle r, b, r \rangle$ or $\sigma = \langle b, r, b \rangle$ is sufficient to obtain a stable result. We can compute both sequences and use the result with smallest total edge length. In order to prove the lemma, we use the following observation.

Observation 1 *For any number of hyperedges, a computation sequence featuring two consecutive occurrences of the same hyperedge achieves the same result as the computation sequence in which these consecutive occurrences have been replaced by a single occurrence.*

Lemma 4 *Let $H = (V, \{r, b\})$ be a hypergraph. All computation sequences σ' with $|\sigma'| \geq 4$ have a shorter computation sequence σ with $|\sigma| = 3$ such that $G_\sigma = G_{\sigma'}$.*

Proof: By Observation 1 we may assume σ' to be an alternating sequence of r's and b's, so σ' starts either with $\langle r, b, r, b, \ldots\rangle$ or with $\langle b, r, b, r, \ldots\rangle$. Without loss of generality we consider σ' to start with $\langle r, b, r, b, \ldots\rangle$. We show that the subsequence σ consisting of the first three hyperedges of σ' achieves the same support as σ'. Consider all edges $(v_i, v_j) \in V \times V$. There are four cases:

- If both v_i and v_j are in both r and b, let the edge be in a set P of purple edges.
- Else, if v_i and v_j are both in r, let the edge be in a set R of red edges.
- Else, if v_i and v_j are both in b, let the edge be in a set B of blue edges.
- Else, the edge will never be a part of a support as the vertices do not share a color.

Let the support constructed after step i of σ be called G_i, so that we have G_1, G_2 and G_3. We show that $P(G_2) = P(G_3)$, where $P(G)$ denotes taking the subset of edges of G that are in P.

$P(G_3) \subseteq P(G_2)$**.** Let $e_p \in P(G_3)$. For a contradiction, assume $e_p \notin P(G_2)$. Edges in P are never removed from the support once they are added, since they have weight 0. This implies that also $e_p \notin P(G_1)$. As G_1 is the Euclidean MST of r, by the cut property of MSTs there is another edge $e \in R \cup P$ shorter than e_p in the cut induced by e_p that must be a part of the MST instead.[3] When constructing G_3, again e will be chosen over e_p, and thus $e_p \notin P(G_3)$. Contradiction.

$P(G_2) \subseteq P(G_3)$**.** Let $e_p \in P(G_2)$. We already established that edges in P are never removed from the support once they are added, hence $e_p \in P(G_3)$.

Next, we show that $G_4 = G_3$, i.e., $G_{\sigma'} = G_\sigma$.

$G_3 \subseteq G_4$**.** Take an edge $e \in G_3$. For a contradiction, assume $e \notin G_4$. As edges in P are not removed and edges in R remain untouched, $e \in B$. As $e \notin G_4$ and the fourth step calculates MST(b), the cut property tells us that some other edge $e' \in B \cup P$ is shorter and in MST(b) instead. But then e' would have been added in G_2 and hence $e \notin G_3$. Contradiction.

$G_4 \subseteq G_3$**.** Take an edge $e \in G_4$. For a contradiction, assume $e \notin G_3$. This means $e \notin R$, as such edges cannot be added when computing MST(b). Edges in P are never removed, thus $e \notin G_2$. The second step of σ computed MST(b), hence by the cut property there must be another edge e', shorter than e, part of MST(b) instead. Indeed, this implies $e \notin$ MST(b). However, as G_4 is computing an MST for b and we assumed $e \in G_4$, $e \in$ MST(b). Contradiction.

As we have now shown that $G_{\sigma'} = G_\sigma$, the lemma follows. □

[3] This requires the same assumption of unique distances or determinism as Lemma 3.

Choosing a computation sequence The question remains how to determine a good computation sequence σ for a hypergraph $H = (V, \{s_1, \ldots, s_k\})$. Based on the above, we use $\sigma = \langle s_1, s_2, s_1 \rangle$ for $k = 2$, knowing that longer sequences do not alter the result. It remains open whether we can prove similar statements for $k > 3$. That is, how can we strategically choose a computation sequence, to get good results with respect to all possible computation sequences? For our experiments, we use $\sigma = \langle s_1, \ldots, s_k \rangle^k$, that is, $\langle s_1, \ldots, s_k \rangle$ repeated k times.

3.2 Local Search

The algorithm described in Section 3.1 appears to perform well in practice, as shown in Section 4. However, it cannot guarantee that the resulting support is plane or acyclic. Moreover, one may wonder whether other commonly employed heuristic approaches outperform it even in the unrestricted case it solves. We therefore implement a local-search algorithm, specifically a hill-climbing heuristic, for comparison in the nonplanar case, but also to allow for computing supports that are plane or acyclic (or both).

Algorithm This approach assumes that in the given hypergraph $H = (V, S)$, at least one vertex $v \in V$ occurs in all hyperedges $s \in S$ such that Lemma 1 applies; let $V_A = \bigcap_{s \in S} s \neq \emptyset$. We need to initialize our hill climbing approach with a valid (plane), easy-to-find albeit possibly suboptimal solution. Following Lemma 1, we obtain this by first calculating an EMST of all vertices in V_A, and subsequently connecting all vertices $v \notin V_A$ to the nearest $v' \in V_A$.

Afterwards, we iteratively execute rounds until no further improvement is gained. We provide pseudocode in Algorithm 2. Each round consists of checking for each edge in the support if it can be removed, and if the hyperedges using it can be reconnected by (one or more) other edges that have a shorter total length than the removed edge without causing intersections. This check is nontrivial and done in a brute-force manner, improved by caching and pruning; more details are given below. At the end of each round, the edge replacement that reduces the total edge length most is actually executed. More rounds are evaluated until no single edge replacement reduces the total edge length.

As the initial state is a plane support tree, we can also readily enforce acyclicity, or relax the constraints to allow intersections. In the provided pseudocode, this implies the following changes. To enforce a plane support, we need to only change Line 6 to include only edges that do not intersect an edge in $E \setminus \{e\}$. To enforce a support tree, we need to only change this same line, to include only edges that connect the two components of *all* hyperedges in D. Note that Line 7 can also be simplified in the support tree case: R is now simply the shortest edge in C and no longer needs the brute-force approach. Though observe that our implementation readily ensures this same computation with only minor overhead.

Algorithm 2 LocalSearch(H)

Input: hypergraph $H = (V, S)$ with vertices in the plane, and $V_A = \bigcap_{s \in S} s \neq \emptyset$
Output: a support for H

```
1: Initialize graph G = (V, E), where E is union of the EMST on V_A and, for
   each u ∈ V \ V_A, a shortest edge to a v ∈ V_A
2: while G is changed do
3:    Remember the best replacement (e*, R*, ℓ*) to (nil, nil, 0)
4:    for each e ∈ E do
5:       Determine the hyperedges D ⊆ S that do not induce a connected sub-
         graph in G if e was removed
6:       Find all candidate edges C ⊆ V² \ E that connect two components of
         some hyperedge in D
7:       Find the replacement R ⊆ C with minimum total length that recon-
         nects all hyperedges in D
8:       Set ℓ to ‖e‖ − Σ_{r∈R} ‖r‖
9:       If ℓ > ℓ*, update (e*, R*, ℓ*) to (e, R, ℓ)
10:   Perform the best replacement (e*, R*, ℓ*), if (e*, R*, ℓ*) ≠ (nil, nil, 0)
11: return G
```

Finding the best replacement We are given a set C of candidate edges, and aim to compute the subset R with minimal length that reconnects the hyperedges in D. We use the length of a set of edges to refer to its sum of edge lengths. To this end, we implement a recursive branch-and-bound algorithm, that builds a set R', by considering each edge $c \in C$ in order of increasing length. We use D' to denote the set of hyperedges of D that is *not* reconnected by R'. If D' is empty, R' reconnects all hyperedges in D and is a solution; we update R if the length of R exceeds that of R'. If c reconnects one or more hyperedges in D', we add c to R', update D' accordingly and recurse. Regardless, we recurse with the original R' and D', that is, without adding c to R'.

We prune the recursive search as follows. When the length of R' plus $\|c\|$ exceeds that of R, the algorithm will never find a better solution anymore and thus the recursion stops. To improve the best replacement found so far, we must find a subset R such that $\|e\| - \sum_{r \in R} \|r\| > \|e^*\| - \sum_{r \in R^*} \|r\|$. That is, we eventually use R, only if the length of R is at most $\|e\| - \|e^*\| + \sum_{r \in R^*} \|r\|$; we use this value to compare to, when no R has been found yet.

Analysis We analyze the complexity of one iteration of our local-search algorithm, that is, the time it takes to perform one replacement. We assume a hypergraph with n vertices and k hyperedges. We use a straightforward implementation. For a given edge $e \in E$, we find D by running a linear-time breadth-first search for every hyperedge that contains both endpoints of e; this takes $O(kn)$ time. To find the $O(n^2)$ candidate edges C, we can check whether a candidate edge reconnects the induced graph of a hyperedge in $O(1)$ time, by inspecting whether one endpoint was reached during the BFS and the other

was not; thus this takes $O(kn^2)$ time in total. We then sort the edges of C according to length, in $O(n^2 \log n)$ time. We have now spent $O(n^2(k + \log n))$ time to prepare for the brute-force search.

We can straightforwardly test in $O(k)$ time whether an edge reconnects any hyperedges in D', by storing the reconnecting set for each candidate. Moreover, a candidate edge may give two recursive calls, but only when $|D'|$ is reduced by one; and the recursion stops when $|D'| = 0$. Therefore, in the binary recursion tree any path from root to leaf has length at most $O(|C|)$ and at most $|D| \leq k$ nodes of degree 2. Thus, this tree has $O(2^k|C|)$ nodes; processing a node takes $O(k)$ time. As a result, the total running time is $O(k2^k|C|) = O(k2^k n^2)$ per iteration.

Note that there may be room to improve this algorithm by first reducing C such that for every candidate edge $c \in C$, there is not a shorter edge $c' \in C$ that reconnects (a superset of) the hyperedges reconnected by c. However, we did not implement this for our experiments. This gives an upper bound of $O(2^k)$ to $|C|$, which might be beneficial for low values of k. However, this would yield significant improvements, only if some hyperedges are reconnected only by the "longer edges" in C.

3.3 Integer Linear Program

Theorem 1 implies that several variants of computing the shortest plane support are NP-hard. Here we sketch how to obtain an *integer linear program* (ILP) for a hypergraph $H = (V, S)$, allowing us to leverage effective ILP solvers, which can provide exact solutions, albeit not in polynomial time.

We introduce variables $e_{u,v} \in \{0, 1\}$, indicating whether edge uv is selected for the support or not. This allows us to represent a graph with fixed vertices. Because the vertex locations are fixed, we can precompute edge lengths $d_{u,v}$ as well as which pairs of edges intersect. This gives the following basic ILP

$$\begin{array}{ll} \text{minimize} & \sum_{u,v \in V} d_{u,v} \cdot e_{u,v} \\ \text{subject to} & e_{u,v} + e_{w,x} \leq 1 \qquad \text{for all } u, v, w, x \in V \text{ if } uv \text{ and } wx \text{ intersect.} \end{array}$$

What remains is to ensure that the graph is also a support: we need additional constraints that imply that each hyperedge in S induces a connected subgraph. To this end, we construct a flow tree for each hyperedge s. We pick an arbitrary sink for the hyperedge, $\sigma_s \in s$, that may receive flow, and let the remaining vertices in s generate one unit of flow that needs to reach σ_s using only edges of s. To formalize this, we introduce variables $f_{s,u,v} \in \{0, 1, \ldots, |s| - 1\}$ for each $s \in S$ and $u, v \in s$ with $u \neq v$. We now need the following constraints: (a) the incoming flow at σ_s is exactly $|s| - 1$; (b) the outgoing flow at σ_s is zero; (c) except for σ_s, each vertex in s sends out one unit of flow more than it receives; (d) flow can be sent only over selected edges.

(a)	$\sum_{u \in s \setminus \{\sigma_s\}} f_{s,u,\sigma_s} = \lvert s \rvert - 1$	for all $s \in S$
(b)	$f_{s,\sigma_s,v} = 0$	for all $s \in S, v \in s \setminus \{\sigma_s\}$
(c)	$\sum_{v \in s \setminus \{u\}} (f_{s,u,v} - f_{s,v,u}) = 1$	for all $s \in S, u \in s \setminus \{\sigma_s\}$
(d)	$f_{s,u,v} \leq e_{u,v} \cdot (\lvert s \rvert - 1)$	for all $s \in S, u, v \in s$ with $u \neq v$

In the analysis below, let $n = |V|$ denote the number of vertices and $k = |S|$ the number of hyperedges. The basic program already has $O(n^2)$ variables and $O(n^4)$ constraints: each potential edge – a pair of vertices with a common set – results in a variable and every pair of such edges that intersect cause a constraint. The flow trees to ensure connectivity add, for each edge-set combination, another variable to capture the flow of that set through that edge. They also add a constraint for each variable, to limit the flow through the edge (constraint (d)). More constraints are added ((a)–(c)), but their number is far fewer than one per edge-set combination. In total we obtain $O(kn^2)$ variables and $O(n^4 + kn^2)$ constraints. The flow trees are essential for a correct answer, but we may expect that short supports often avoid many of the potential intersections automatically. Therefore, we implement the intersections as lazy constraints, being added to the ILP only if a solution is found that violates the constraint.

The analysis above implicitly assumes that each set spans $O(n)$ vertices. For hypergraphs with relatively few hyperedges, this may be a reasonable assumption. But we may use a more fine-grained analysis to get better bounds on the complexity. Specifically, since we add potential edges only if the endpoints have a common set, we have $O(|s|^2)$ potential edges in one set and the same number of variables for the flow tree. The number of candidate edges is thus upper bounded by $\sum_{s \in S} |s|^2$ as well as by n^2. Note that the former may exceed the latter, as the summation counts edges with k' sets in common between its endpoints k' times, whereas we add only one candidate edge. The number of candidate edges is at most $\min\{n^2, \sum_{s \in S} |s|^2\}$. Thus, the number of variables becomes $O(\sum_{s \in S} |s|^2)$ and the number of constraints $O(\min\{n^2, \sum_{s \in S} |s|^2\}^2 + \sum_{s \in S} |s|^2) = O(\min\{n^2, \sum_{s \in S} |s|^2\}^2)$. For example, if each set spans only $O(\sqrt{n})$ vertices and the number of sets is small ($k = O(n)$), we find that we have $O(kn)$ variables and $O((kn)^2)$ constraints.

Variants The described ILP results in the shortest plane support for H. It is easily modified to allow nonplanar supports by omitting the intersection constraints. Similarly, the ILP can be extended to penalize or admit a limited number of intersections, by adding an indicator variable for each intersecting pair. However, this increases the number of variables from quadratic to quartic.

For each of these variants, we can also enforce the support to be a tree (or forest). To do so, we enforce that the number of selected edges is exactly $n - c$, where c is the number of connected components of the graph of potential edges. Since the flow trees enforce that each such connected component is connected in the computed support, this is in fact sufficient. Note that if there is a vertex that is contained in all sets, then c is always equal to one.

4 Experiments

As discussed above, there are various ways of defining and computing good supports. In this section we discuss several computational experiments that were performed to gain insight into the trade-offs between the different methods and properties. In particular, we use two different setups. In the first experiment, we exclude exact but slow algorithms to extensively compare the heuristic algorithms. In the second experiment, we include exact algorithms to answer questions about the effect of requiring planarity or support trees, and to investigate how well heuristic algorithms approximate the optimal solution, albeit on smaller data sets. We provide selected plots of the experimental data in this section; the detailed plots of all experiments are found on pages 491 to 494.

Algorithms We shall study four algorithms under various conditions in these experiments. In particular, we use MSTApproximation to refer to the simple approximation algorithm of computing a minimum spanning tree for each hyperedge and then taking their union [14]. We refer to our heuristic improvement as MSTIteration (Section 3.1). Finally, we use LocalSearch to indicate our local search algorithm (Section 3.2) and Opt to denote an exact algorithm for computing optimal solutions. The latter two allow four different conditions, by requiring a plane support, a support tree, both (i.e., a plane support tree) or neither (unrestricted). We append P, T, PT and U to denote these conditions.

Data generation We generate a random hypergraph $H = (V, S)$ via the procedure below. We use $n = |V|$ and $k = |S|$ to denote the desired number of vertices and hyperedges, respectively. Moreover, we specify one of four degree-distribution schemes: even, mid, low or high. Finally, the spatial distribution of the vertices is determined in one of two ways: uniform or clustered. As this placement method does not influence the generation process structurally, we describe this further at the end.

1. Initialize an array $D[1 \dots k]$ such that $\sum_{i=1}^{k} D[i] = n$, in which $D[i]$ indicates that we wish to generate $D[i]$ vertices of degree i. To this end, we define four schemes, where we always restrict the degrees to be between 1 and k.

   even All degrees occur equally frequently. If $n \bmod k \neq 0$, then degrees one through $n \bmod k$ occur once more than the others.

   mid We generate n random degrees using a normal distribution. We draw a random value g from $\mathcal{N}(0.5, 2/9)$ and map this to degree $1 + \lfloor kg \rfloor$. The distribution of degrees is expected to look like a Gaussian curve with its peak on $k/2$.

   low Similar to the mid scheme, we draw a random value g from $\mathcal{N}(0, 2/5)$ and map this to degree $1 + \lfloor k|g| \rfloor$. The distribution of degrees is expected to look like a Gaussian curve with its peak on 1.

HIGH Similar to the MID scheme, we draw a random value g from $\mathcal{N}(0, 2/5)$ and map this to degree $k - \lfloor k|g| \rfloor$. The distribution of degrees is expected to look like a Gaussian curve with its peak on k.

2. If $D[k] = 0$, decrease the maximal degree i for which $D[i] > 0$ by one and set $D[k]$ to one.

3. While $\sum_{i=1}^{k} i \cdot D[i] < 2k$, decrease the minimal degree i for which $D[i] > 0$ by one and increase $D[i+1]$ by one.

4. While $\sum D[i] > 0$, let i be a degree such that $D[i] > 0$, chosen uniformly at random. Generate vertex v with a random position and add it to V. Pick i hyperedges uniformly at random from those hyperedges that have less than two vertices; if there are no such hyperedges left, pick from all hyperedges instead. Decrease $D[i]$ by one.

To explain the four steps in this process, we treat them in reverse order.

4. We generate all desired n vertices and assign them to hyperedges. We first pick from those hyperedges that have less than two vertices, to ensure that each hyperedge contains at least two vertices. This ensures that all hyperedges have influence on the support. We pick a random degree, to avoid biasing small hyperedges towards low degree or high degree vertices.

3. We ensure that the sum over all degrees (over all nodes) is at least $2k$. We need this lower bound on the sum of degrees, to ensure that we are able to pick at least two vertices for every hyperedge.

2. We ensure that there is at least one vertex that occurs in all hyperedges; this step is optional but necessary to ensure that our local search algorithm can be initialized. It guarantees that a planar solution exists, see Section 3.2. In our experiments, this step is always performed.

1. We decide on the distribution over the degrees. That is, how many vertices shall we have of degree i? This can be done according to various schemes. The four schemes used in this paper are described in the main text.

We place vertices in one of two ways. In the UNIFORM method, we place a vertex uniformly at random in a square of side length 100. Following the method of Meulemans [20], the CLUSTERED method aims at generating placements that are more spatially structured as one may expect with real data. Specifically, for a given instance, we first generate five helper points uniformly at random in a square of side length 100 and compute the Euclidean minimum spanning tree on these. We add one extra edge, namely the one that reduces the dilation the most, that is, that decreases the maximum ratio between the path length in the tree and the Euclidean distance, over all pairs of helper points. This skeleton is then used to place the vertices: each placement consists of uniformly at random picking one of the five edges, and then positioning the vertex close to the chosen edge. Specifically, for edge (a, b) with vector $r = a - b$ and r' being r rotated

by 90 degrees, we place the point at $a + \lambda \cdot r + \mu \cdot r'$ where λ is drawn uniformly at random from $[-0.1, 1.1]$ and μ from a standard Gaussian distribution with mean 0 and standard deviation 1.

Analysis Following recommended practices for statistical analysis, we analyzed the results with an estimation-based approach using confidence intervals (CIs) [10]. Unless indicated otherwise, each of the plots in the following subsections show the estimated means (black dots) and 99% CIs (colored bars) for each condition based on 1000 trials. If the CIs of two conditions are disjoint, the results are significantly different ($\alpha = 0.01$).

Moreover, rather than analyzing length directly, we always divide lengths by the length of the Euclidean minimum spanning tree of the vertices. We refer to this value as the *edge length ratio* of a solution. Though the EMST is typically not a support, it provides a lower bound on a support length and thus allows us to normalize. It eliminates undesirable effects of scale or precise vertex placement.

4.1 Experiment 1: Comparison of Heuristics

Here we focus on answering the following three questions: (1) how much does the spanning tree iteration help to reduce the length of the support, compared to computing the minimum spanning trees in isolation; (2) which heuristic algorithm performs best in terms of support length; (3) which heuristic algorithm performs best in terms of computation time?

Setup For each combination of $n \in \{20, 40, 60, 80, 100\}$, $k \in \{2, 3, 4, 5, 6, 7\}$, $d \in \{\text{EVEN}, \text{MID}, \text{LOW}, \text{HIGH}\}$ and $p \in \{\text{UNIFORM}, \text{CLUSTERED}\}$, we generate 1000 random hypergraphs with n vertices and k hyperedges according to degree distribution scheme d and placement method p. For each hypergraph, we run six algorithms: MSTApproximation and MSTIteration as well as LocalSearch U/T/P/PT. This experiment was run on one machine, sequentially in a single thread to also allow for comparison of runtime performance. The machine was an HP ZBook with an Intel Core i7-6700HQ CPU, 24 GB RAM and running Windows 8.1.

Results We first consider question (1) and compare MSTApproximation and MSTIteration. Since MSTIteration can only improve upon MSTApproximation, we express this as a ratio between 0 and 1. In Figure 6 we show the results for $n = 20, 60, 100$ (Figure 11 on page 491 provides the chart for all cases). Interestingly, the median gain remains roughly equal as we increase the number of vertices, though the variance becomes lower. Increasing the number of hyperedges gradually increases the relative gain of MSTIteration. We also observe a dependency on the degree distribution. In particular, MID and EVEN systematically benefit more from iteration than LOW and HIGH. We explain this by observing that in the extreme cases MSTApproximation is optimal:

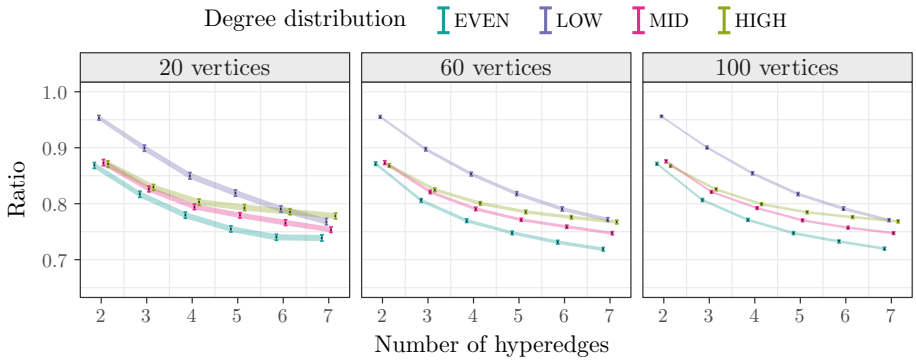

Figure 6: Ratio between the support length computed by MSTIteration and by MSTApproximation. Lower values indicate a higher gain of the iteration method. Shown are the 99% confidence intervals based on 1000 trials with uniform placement.

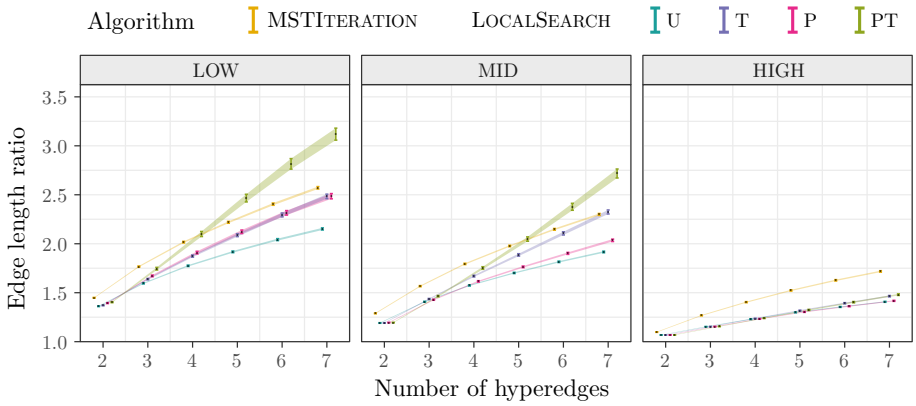

Figure 7: The support length computed by the algorithms as a ratio to the EMST length. Shown are the 99% confidence intervals based on 1000 trials with 100 vertices and uniform placement.

if all vertices have degree 1, then the optimal support is simply the union of all (disjoint) minimum spanning trees; if all vertices have degree k, then the optimal support is also simply the minimum spanning tree on the vertices. Difficulties arise when having many vertices that are part of multiple but not all hyperedges. This corresponds to the MID and EVEN schemes.

Let us now turn towards question (2), and consider the resulting support length of the LocalSearch algorithm as well. We omit MSTApproximation from these comparisons, since MSTIteration always performs at least as well. In Figure 7 we show the results for $n = 100$ (Figure 12 on page 492 provides the chart for all cases). As one may expect, the length increases gradually with more hyperedges, as the support must use more edges to ensure that each hyperedge induces a connected subgraph. Moreover, we see that LocalSearch U consistently outperforms MSTIteration. To be exact, this is the case in 98.5% of all trials; the average ratio of LocalSearch U to MSTIteration (including those trials in which MSTIteration performs better) is 0.877, that is, the support length is over 12% shorter on average. The effect of degree distribution also stands out. In LOW and MID, requiring planarity or a support tree has a large effect on the support length, whereas this is not the case in EVEN and HIGH. To explain this, observe that the minimum spanning tree on vertices that are in many or all hyperedges is planar and likely a part of the computed solution; in the EVEN and HIGH cases, there are comparatively many such vertices which can then serve as places to connect the other vertices in the support. In the LOW and MID cases, there are only few such vertices and thus the shortest connections that can be used to connect these to such a "backbone" structure are likely to intersect other connections. Though the number of vertices has little effect on MSTIteration and LocalSearch U, this does exacerbate the above problem: more vertices leads to a larger increase in support length when we enforce planarity or a support tree.

Finally, we briefly consider question (3) and compare the computation times of the various algorithms (see Figure 8, or Figure 14 on page 494). We see that the number of hyperedges impacts the computation only slightly, whereas the number of vertices has a much stronger effect. MSTIteration clearly outperforms the LocalSearch variants, running on average 95.11% faster than LocalSearch U over all trials (98.73% faster on trials with $n = 100$). Another clear pattern is that requiring planarity with LocalSearch increases the running time significantly (272.64% slower over all trials, 354.06% on trials with $n = 100$); the number of steps to arrive at a local minimum is not sufficiently reduced to compensate for the time spent on checking intersections. In fact, the number of iterations may increase as shorter replacements may become admissible later. Interestingly, the running time slightly decreases for the PT case as the number of hyperedges increases. This is likely due to the severe restrictions this setting poses: an edge must be replaced by exactly one other edge that is shorter, free of intersections *and* connects at least the same sets as the original edge.

In the above, the supporting figures showcase results of the UNIFORM placement, but our observations generally apply to the CLUSTERED case also. There

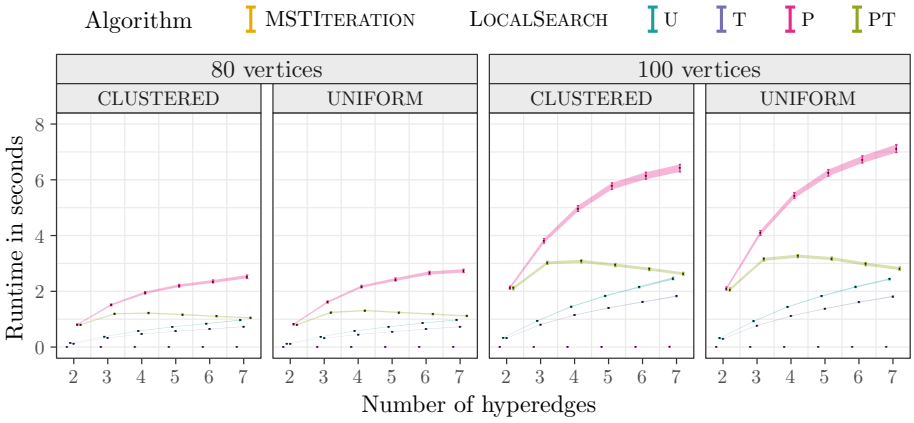

Figure 8: Computation time of the various algorithms. Shown are the 99% confidence intervals based on 1000 trials with degree distribution MID.

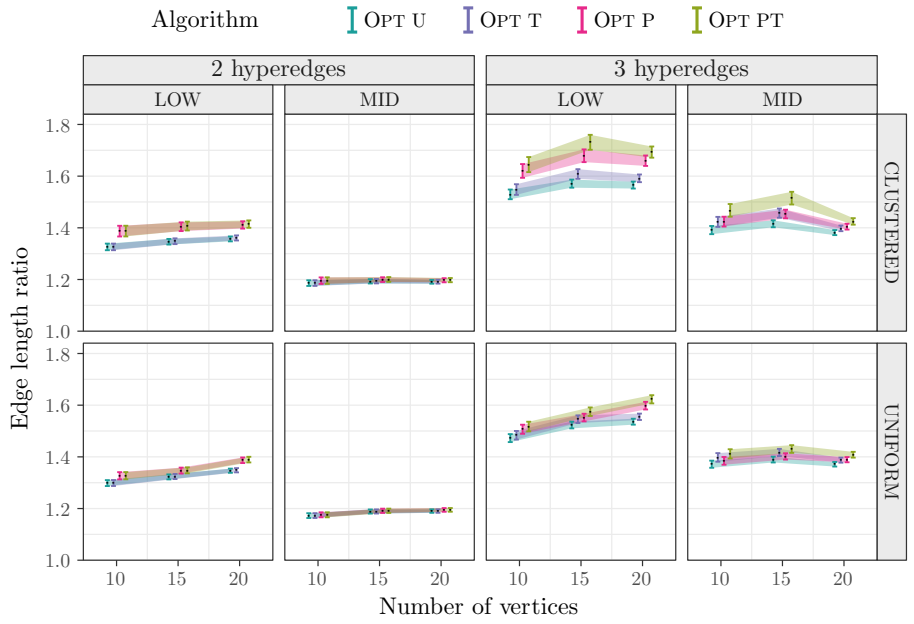

Figure 9: The support length achieved by Opt in the four conditions U/T/P/PT as a ratio to the EMST length.

are some small differences comparing the two settings. The edge length ratio for CLUSTERED tend to be slightly higher than for UNIFORM (see Figures 12 and 13). On the other hand, the computation times for CLUSTERED are lower than for UNIFORM for LOCALSEARCH (see Figure 14). Both phenomena can likely be explained by the clustered nature of CLUSTERED placement: distances within the clusters (close in the skeleton) are expected to be notably shorter than distances between clusters. Thus, when edges between clusters are necessary to obtain a support, this increases the total edge length more significantly. At the same time, this implies that candidate edges for LOCALSEARCH are often longer compared to the current edges and replacements are less likely to be performed.

4.2 Experiment 2: Approximation of Optimality

Here we focus on answering two questions: (1) how is the support length affected by additionally requiring that the support is a tree and/or is planar; (2) how well do the heuristic algorithms approximate the optimal solution?

Setup For each combination of $n \in \{10, 15, 20\}$, $k \in \{2, 3\}$, $d \in \{\text{LOW}, \text{MID}\}$ and $p \in \{\text{UNIFORM}, \text{CLUSTERED}\}$, we generate 1000 random hypergraphs with n vertices, k hyperedges according to degree distribution scheme d and placement method p. For each hypergraph, we run the LOCALSEARCH U/T/P/PT and compute an optimal solution OPT U/T/P/PT[4]. To obtain a large enough number of trials, these experiments were run on different machines simultaneously and in concurrent threads. As such, we refrain from analyzing algorithm speed in this experiment.

Results Let us first compare the optimal solutions according to the four different restrictions. In Figure 9 we show the results. For two hyperedges, we see that there is no significant effect of requiring support trees; however, there is a small effect on the worst-case edge length ratio observed for the LOW case. For three hyperedges, we see that the effects become slightly larger. Most noticeable is that enforcing support trees has now a slight effect; but this is only significant for the CLUSTERED- MID case. In terms of plane supports, we see a similar pattern as before, that is, that of an increase particularly in the LOW case, but also some in the MID case. Note that the effects for $n = 20$ are potentially underestimated; see the discussion in the next section.

Let us now turn towards how well LOCALSEARCH performs with respect to the optimal solution. Our results indicate that in a majority of the cases, our heuristic actually achieves optimal results (see Figure 10; or Figure 15 for a split according to placement method). For $n = 10$, 15 we see a clear decrease of this percentage for plane supports and trees; we attribute the apparent increase at $n = 20$ to the failed trials (see discussion in the next section). To further see how well LOCALSEARCH performs if it fails to achieve optimal results, we look

[4]For $n \in \{10, 15\}$ and $p =$ UNIFORM, this is a simple branch and bound algorithm; for $n = 20$ or $p =$ CLUSTERED we use the ILP solution, solved with IBM ILOG CPLEX 12.6.3.

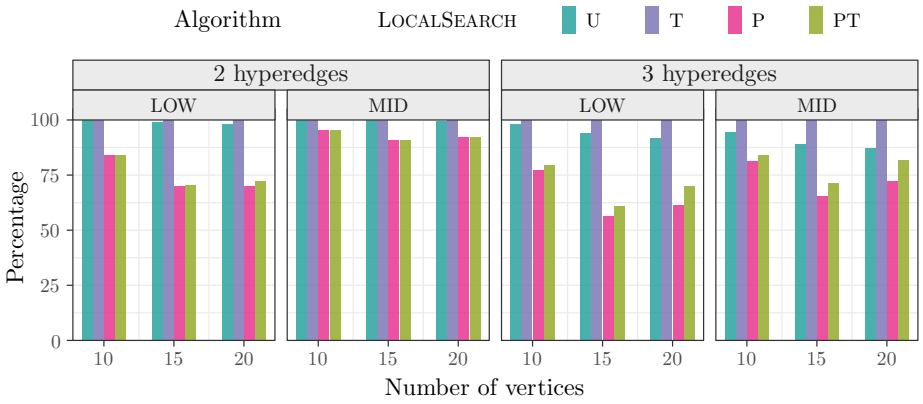

Figure 10: Percentage of trials of LocalSearch that achieve the optimal solution over 2000 trials (1000 per placement method). Note that LocalSearch T always achieves optimal results.

at the ratio between the support length it achieves and the optimal support length. In all cases, we observe a ratio of less than 1.61. The 90-, 95-, and 99-percentile of this ratio was worst for LocalSearch PT, being 1.05, 1.09, and 1.19, respectively. Again, we have to keep in mind that the data for $n = 20$ likely exclude some of the more difficult cases and thus the trend in the increasing ratio might extend further for a larger number of vertices. Shown by the P and PT cases, uniform placement yields more difficult instances than clustered placement: the former achieves optimal results less often than the latter.

5 Discussion and Limitations

Failed trials For the experiments described in Section 4.2, not all trials could be completed. CPLEX was allocated 24GB of RAM and 64GB of file storage. Nonetheless, the CPLEX computation would run out of memory and therefore not finish successfully for some cases with $n = 20$; all instances with $n \in \{10, 15\}$ solved successfully. For the uniform case, we have therefore ran 1730 trials for each of the four conditions ($k \times d$) with four settings for Opt; 941 runs out of these $27,680$ runs failed (approximately 3.4%). This is shown in Table 2. We filtered out erroneous trials, leaving 1138 trials, 1000 of which were used for the analysis of the results to match the cases for $n = 10, 15$. The same approach was taken for clustered placements.

This may bias the results towards only including the "easier" cases on which CPLEX was successful; this should be taken into consideration when interpreting the results. To localize and quantify this bias, we counted which conditions failed and, for each condition, measured the average length of the LocalSearch results in the successful and failed trials (see Table 2). We note that the tree

Table 2: Number of failed trials for $n = 20$ per condition for UNIFORM placement. Ratio indicates the average length of LocalSearch on failed trials, divided by the average length of LocalSearch on successful trials.

		Opt U		Opt T		Opt P		Opt PT		all	
k	d	#	ratio	#	ratio	#	ratio	#	ratio	#	ratio
2	LOW	2	1.11	7	1.26	2	1.12	15	1.23	26	1.22
	MID	7	1.10	3	1.26	7	1.10	5	1.22	22	1.15
3	LOW	0		61	1.20	0		264	1.26	325	1.25
	MID	18	1.13	169	1.18	20	1.11	361	1.23	568	1.20
all		27	1.09	240	1.24	29	1.07	645	1.33	941	1.29

and plane tree cases are impacted most. We also see that the ratio is mostly well above one, suggesting that indeed the more difficult cases have now been excluded from the analysis.

Implications for set visualization Our work is motivated by the problem of visualizing set systems with fixed vertex locations. As mentioned earlier, supports are implicitly used in various techniques to structure such visualizations [2, 12, 21]. These methods often treat the various sets as independent, either completely or by fixing their order and computing the support for each set only after computing (and thereby fixing) the prior sets. Our theoretical results further support such approaches, as the computational problem where even only two sets influence each other is NP-hard when minimizing the length of the support while avoiding crossings (by Theorem 1).

We also considered an approximation algorithm and a simple local-search method to compute high-quality supports (see Figure 2 for some results on real data, using a set-visualization rendering). The experiments show that these methods are indeed effective. Especially the local-search approach can handle a variety of constraints, while often giving near-optimal solutions. As such, this type of algorithm seems suitable for computing good supports. At the same time, we should acknowledge that good supports often make a trade-off between various criteria beyond length and planarity. Edges of the support need not be line segments, but may indeed be routed to avoid ambiguity when a segment would be too close to a vertex that is not the set. When we ignore crossings, such routings can be precomputed and considered in, e.g., our local-search method. But the problem becomes more complex when we also consider intersections, as routes of different length may be needed depending on which other edges are part of the support. Yet, if the data set is not too dense, the deviations necessary to reroute can be small. This still allows our methods to be used to compute a good basic support, of which the edge geometry can be modified in a postprocessing step to include other criteria.

6 Conclusion

Motivated by finding structures to visualize set systems, we studied the problem of computing good supports for hypergraphs. Specifically, we focussed on length minimization combined with planarity or acyclicity requirements. We showed that this problem is NP-hard even for two hyperedges with one containing the other. The acyclic case requires that the elements that are contained in all sets form a connected subgraph of the eventual support. We showed that the existence of such elements guarantee that a plane support exists. However, we also showed that extending the Euclidean minimum spanning tree on those elements cannot lead to an $o(n)$-approximation algorithm on a hypergraph with n vertices.

Motivated by the NP-hardness of computing shortest plane supports, we conducted a computational experiment to investigate the quality of supports that can be computed via heuristic algorithms. To this end we introduced two heuristic algorithms and evaluated these with respect to each other and to exact solutions computed by an integer linear program. Our experiments showed that the heuristic LocalSearch often achieves the optimal solution, and otherwise computes a support that is less than 20% longer than the optimal solution in 99% of the cases. Moreover, our experiments showed that LocalSearch performs better than MSTIteration, which in turn is a k-approximation for k hyperedges. But recall that MSTIteration does not support computing planar or acyclic supports. We can also guarantee that LocalSearch (without restrictions) is a k-approximation by initializing it using either MSTApproximation or MSTIteration. However, it is not clear whether this change will generally improve the result of LocalSearch. There is a trade-off between speed and support length, where MSTIteration is faster and LocalSearch yields shorter supports. We also observed that the increase in support length caused by additional requirements, depends both on the number of sets and the number of set memberships per element, but this behavior seems predictable and not to depend on the number of elements.

Future work From the theoretical side, several questions remain open. For example, can we efficiently decide whether a plane support tree exists? We currently know how to answer this only for two hyperedges (using Lemma 1 and a result of Bereg et al. [5]). In terms of length minimization of a plane support, the question of whether the problem can be efficiently solved for two disjoint hyperedges remains open.[5] Recently, Kindermann et al. [16] studied this problem of disjoint hyperedges in the case of many small hyperedges, as opposed to few large hyperedges. Finally, we may also investigate how many iterations we need for MSTIteration with more than two hyperedges to guarantee that the computation stabilizes.

Our experiments indicate that our local search algorithm does not always

[5] In an earlier version [8], we claimed that this was NP-complete. However, a flaw was found in the construction.

perform optimally, especially when requiring plane supports. It is, however, based on simple hill climbing. Can we employ better search techniques such as simulated annealing to efficiently find better solutions?

Finally, we chose to generate random hypergraphs for our experiments, as to not depend on particular properties of (geospatial) configurations that may be inherent to some real-world data sets. While this reduces the explanatory power with respect to real-world data sets, it provides us with more insight into the structural problem, unbiased by unknown or hidden structures of real-world data. We leave it to future work to further dive into real-world data sets, to see if similar trends and patterns emerge or more difficult structures arise and to evaluate in a user study the impact of the different heuristics on readability.

Acknowledgments

This work started at Dagstuhl seminar 17332 "Scalable Set Visualizations". The authors would like to thank Nathalie Henry Riche for providing the data for Figure 2. TC was supported by the Netherlands Organisation for Scientific Research (NWO, 314.99.117). MvG received funding from the European Union's Seventh Framework Programme (FP7/2007-2013) under ERC grant agreement n^o 319209 (project NEXUS 1492) and the German Research Foundation (DFG) within project B02 of SFB/Transregio 161. WM was partially supported by the Netherlands eScience Centre (NLeSC, 027.015.G02).

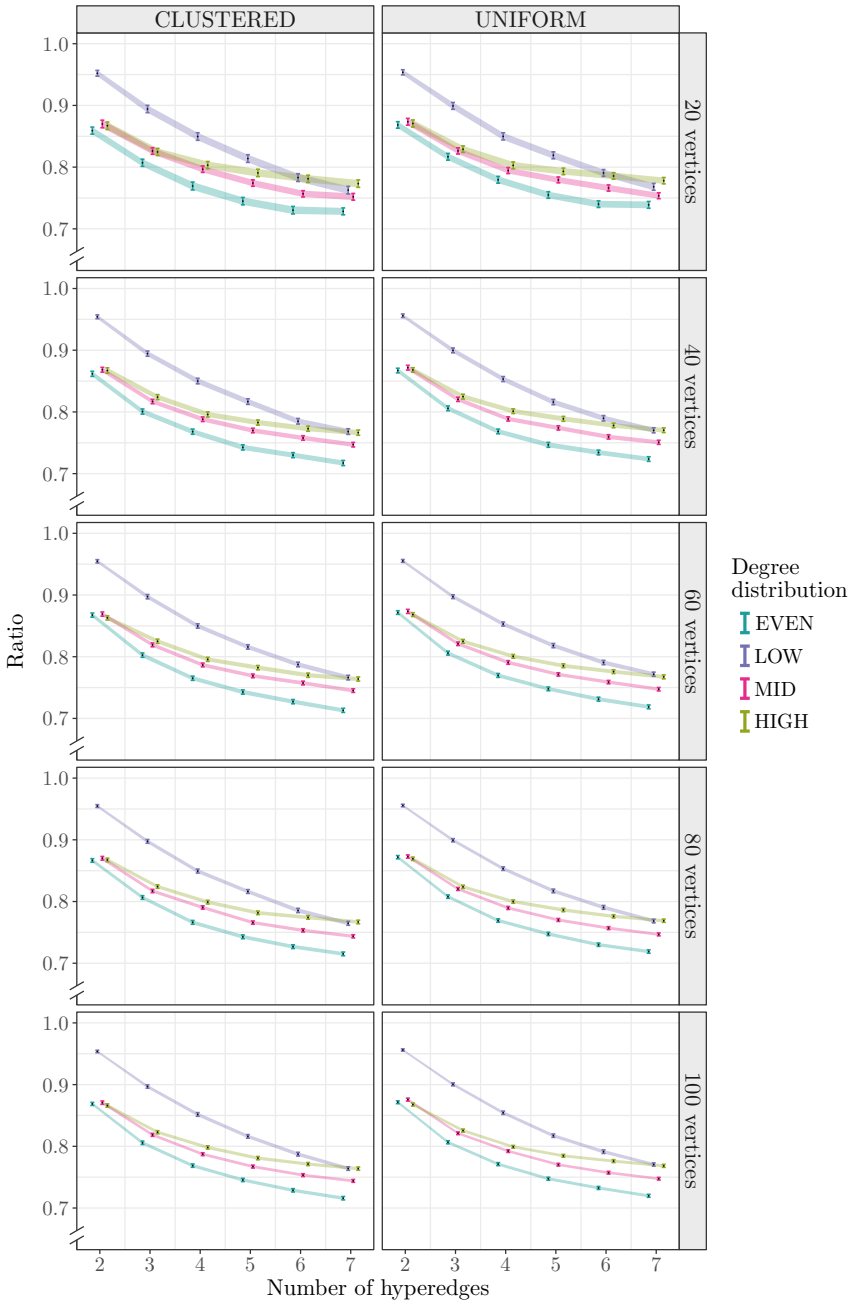

Figure 11: Ratio between the support length computed by MSTIteration and by MSTApproximation. Lower values indicate a higher gain of the iteration method.

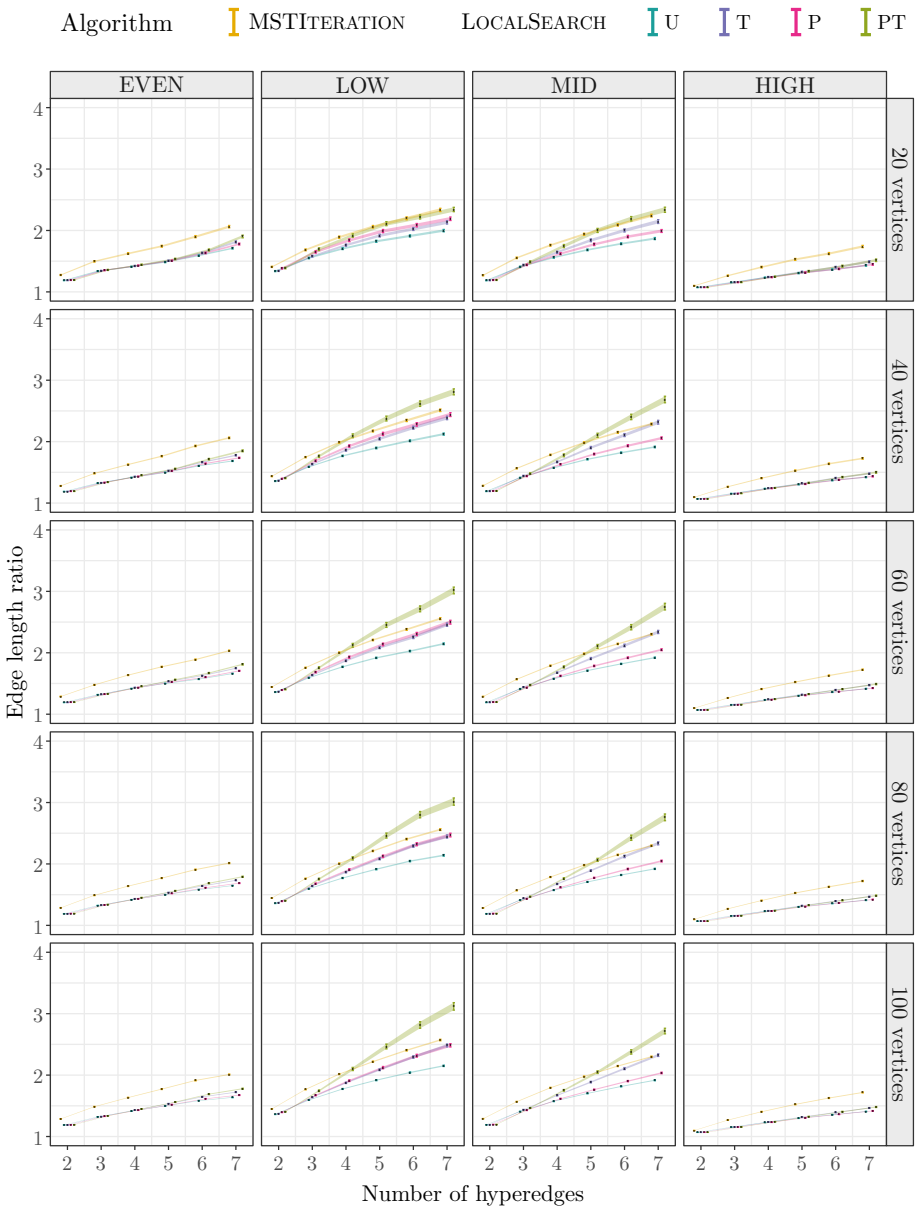

Figure 12: The support length computed by the various algorithms as a ratio to the EMST length for varying values of n, k and d with UNIFORM placement.

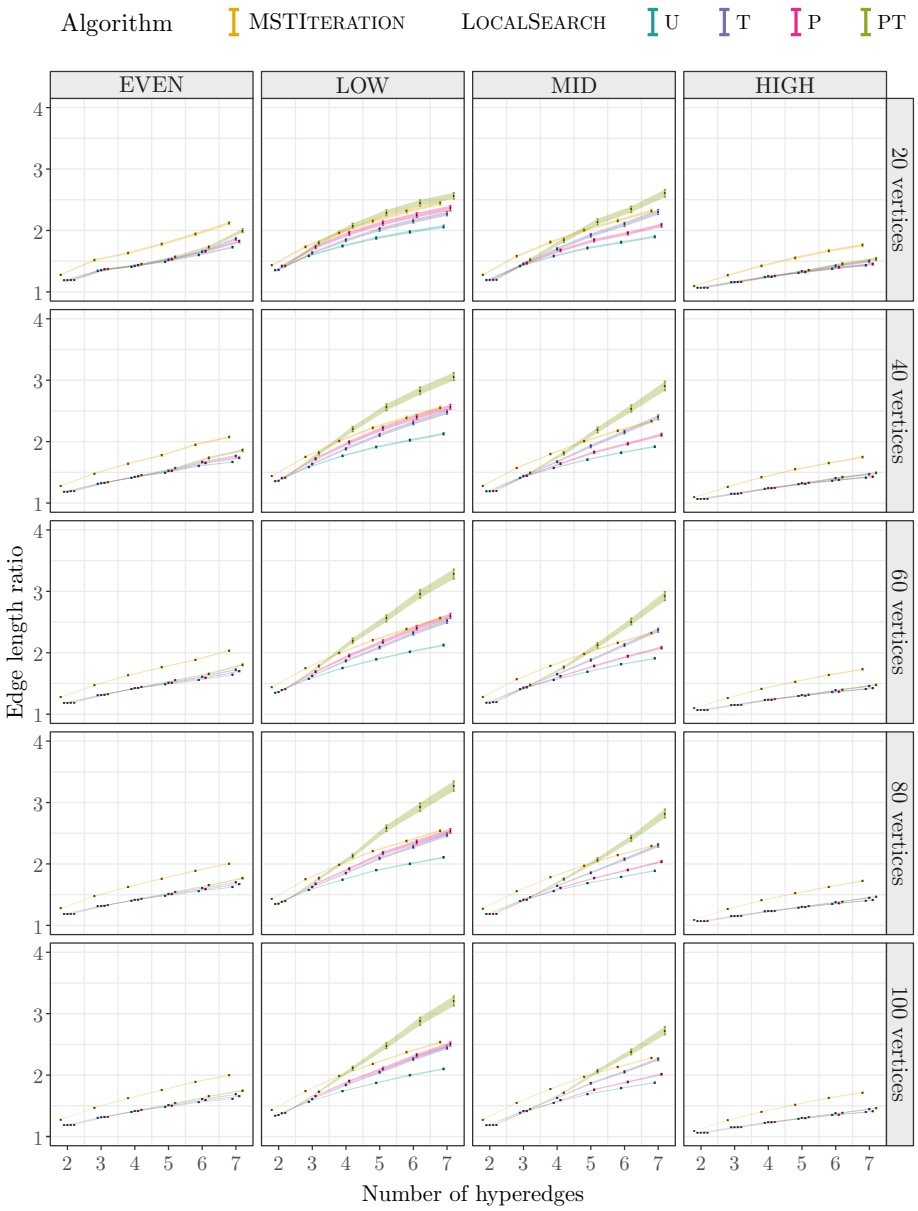

Figure 13: The support length computed by the various algorithms as a ratio to the EMST length for varying values of n, k and d with CLUSTERED placement.

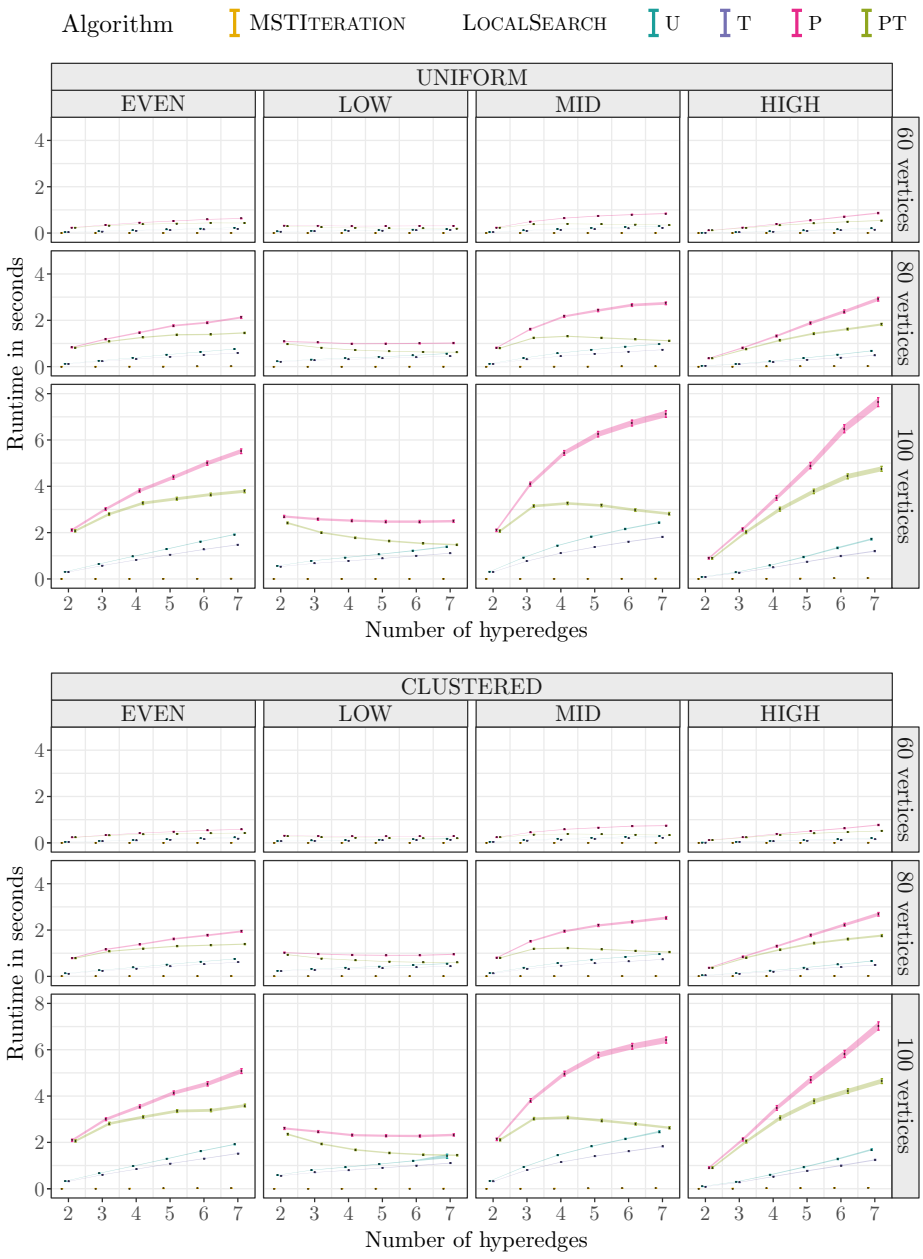

Figure 14: Computation time of the various algorithms for varying values of n, k and d. Computation times for $n = 20$ and $n = 40$ are all close to 0 (below 0.036 seconds for n=20 and below 0.425 seconds for $n = 40$).

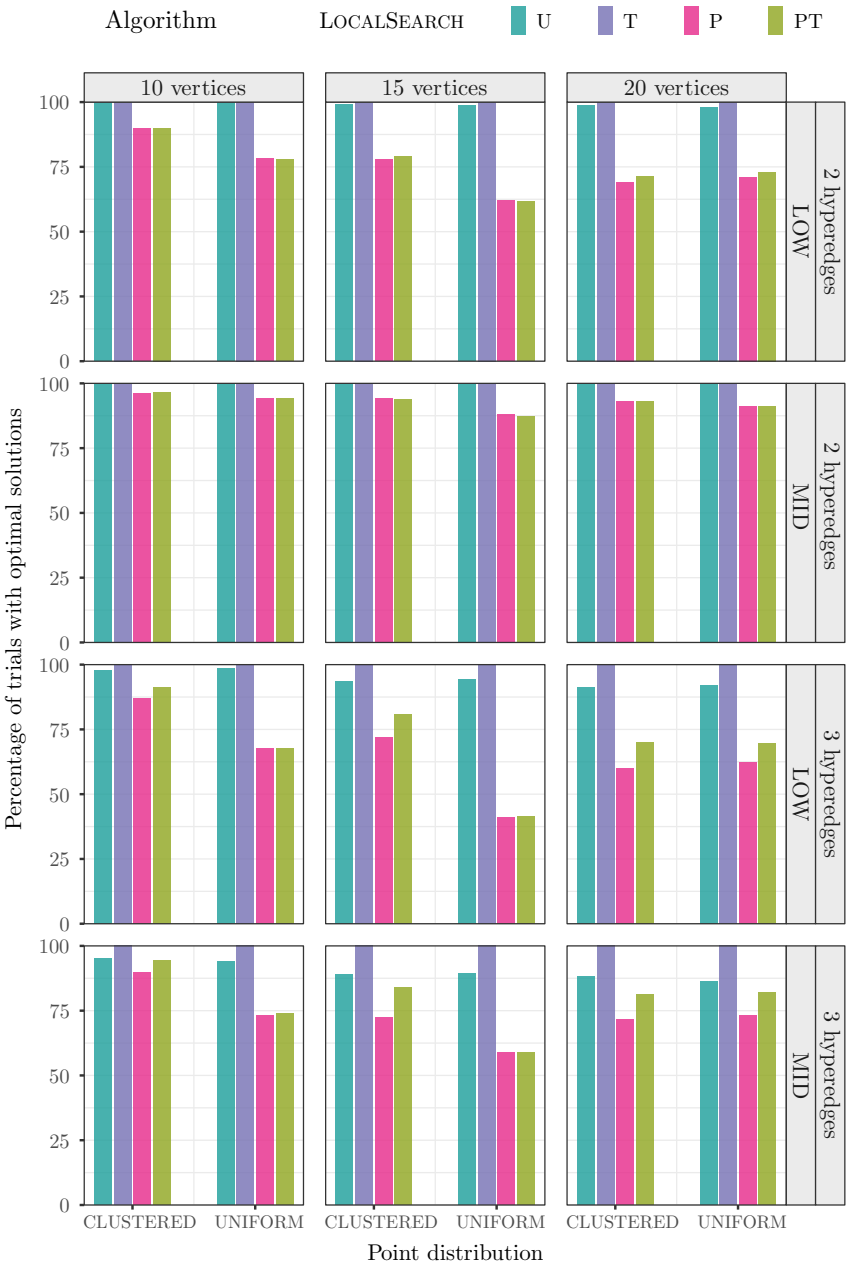

Figure 15: Percentage of the trials where LocalSearch found the optimal solution.

References

[1] H. A. Akitaya, M. Löffler, and C. D. Tóth. Multi-colored spanning graphs. In *Graph Drawing and Network Visualization (GD'16)*, volume 9801 of *LNCS*, pages 81–93. Springer, 2016. doi:10.1007/978-3-319-50106-2_7.

[2] B. Alper, N. Henry Riche, G. Ramos, and M. Czerwinski. Design study of LineSets, a novel set visualization technique. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2259–2267, 2011. doi:10.1109/TVCG.2011.186.

[3] B. Alsallakh, L. Micallef, W. Aigner, H. Hauser, S. Miksch, and P. Rodgers. The state of the art of set visualization. *Computer Graphics Forum*, 35(1):234–260, 2016. doi:10.1111/cgf.12722.

[4] S. Bereg, K. Fleszar, P. Kindermann, S. Pupyrev, J. Spoerhase, and A. Wolff. Colored non-crossing Euclidean Steiner forest. In *Algorithms and Computation (ISAAC'15)*, volume 9472 of *LNCS*, pages 429–441. Springer, 2015. doi:10.1007/978-3-662-48971-0_37.

[5] S. Bereg, M. Jiang, B. Yang, and B. Zhu. On the red/blue spanning tree problem. *Theoretical Computer Science*, 412(23):2459–2467, 2011. doi:10.1016/j.tcs.2010.10.038.

[6] U. Brandes, S. Cornelsen, B. Pampel, and A. Sallaberry. Path-based supports for hypergraphs. *Journal of Discrete Algorithms*, 14:248–261, 2012. doi:10.1016/j.jda.2011.12.009.

[7] K. Buchin, M. van Kreveld, H. Meijer, B. Speckmann, and K. Verbeek. On planar supports for hypergraphs. *Journal of Graph Algorithms and Applications*, 15(4):533–549, 2011. doi:10.7155/jgaa.00237.

[8] T. Castermans, M. van Garderen, W. Meulemans, M. Nöllenburg, and X. Yuan. Short plane supports for spatial hypergraphs. In *Graph Drawing and Network Visualization (GD'18)*, volume 11282 of *LNCS*, pages 1–14. Springer, 2018. doi:10.1007/978-3-030-04414-5_4.

[9] C. Collins, G. Penn, and S. Carpendale. Bubble Sets: Revealing set relations with isocontours over existing visualizations. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1009–1016, 2009. doi:10.1109/TVCG.2009.122.

[10] G. Cumming. *Understanding the new statistics: Effect sizes, confidence intervals, and meta-analysis*. Routledge, 2013.

[11] K. Dinkla, M. El-Kebir, C.-I. Bucur, M. Siderius, M. J. Smit, M. A. Westenberg, and G. W. Klau. eXamine: Exploring annotated modules in networks. *BMC Bioinformatics*, 15:201, 2014. doi:10.1186/1471-2105-15-201.

[12] K. Dinkla, M. van Kreveld, B. Speckmann, and M. Westenberg. Kelp Diagrams: Point set membership visualization. *Computer Graphics Forum*, 31(3pt1):875–884, 2012. doi:10.1111/j.1467-8659.2012.03080.x.

[13] A. Efrat, Y. Hu, S. G. Kobourov, and S. Pupyrev. MapSets: Visualizing embedded and clustered graphs. *Journal of Graph Algorithms and Applications*, 19(2):571–593, 2015. doi:10.7155/jgaa.00364.

[14] F. Hurtado, M. Korman, M. van Kreveld, M. Löffler, V. Sacristán, A. Shioura, R. I. Silveira, B. Speckmann, and T. Tokuyama. Colored spanning graphs for set visualization. *Computational Geometry: Theory and Applications*, 68:262–276, 2018. doi:10.1016/j.comgeo.2017.06.006.

[15] D. S. Johnson and H. O. Pollak. Hypergraph planarity and the complexity of drawing Venn diagrams. *Journal of Graph Theory*, 11(3):309–325, 1987. doi:10.1002/jgt.3190110306.

[16] P. Kindermann, B. Klemz, I. Rutter, P. Schnider, and A. Schulz. The partition spanning forest problem. *Computing Research Repository (arXiv.org)*, 1809.02710, 2018. URL: https://arxiv.org/abs/1809.02710.

[17] B. Klemz, T. Mchedlidze, and M. Nöllenburg. Minimum tree supports for hypergraphs and low-concurrency Euler diagrams. In *Algorithm Theory (SWAT'14)*, volume 8503 of *LNCS*, pages 253–264. Springer, 2014. doi:10.1007/978-3-319-08404-6_23.

[18] E. Korach and M. Stern. The clustering matroid and the optimal clustering tree. *Mathematical Programming*, 98(1–3):385–414, 2003. doi:10.1007/s10107-003-0410-x.

[19] D. Lichtenstein. Planar formulae and their uses. *SIAM Journal on Computing*, 11(2):329–343, 1982. doi:10.1137/0211025.

[20] W. Meulemans. Efficient optimal overlap removal: Algorithms and experiments. *Computer Graphics Forum*, 38(3):713–723, 2019. doi:10.1111/cgf.13722.

[21] W. Meulemans, N. Henry Riche, B. Speckmann, B. Alper, and T. Dwyer. KelpFusion: A hybrid set visualization technique. *IEEE Transactions on Visualization and Computer Graphics*, 19(11):1846–1858, 2013. doi:10.1109/TVCG.2013.76.

[22] H. Purchase. Metrics for graph drawing aesthetics. *Journal of Visual Languages and Computing*, 13(5):501–516, 2002. doi:10.1006/jvlc.2002.0232.

[23] P. J. Rodgers, G. Stapleton, B. Alsallakh, L. Micallef, R. Baker, and S. J. Thompson. A task-based evaluation of combined set and network visualization. *Information Sciences*, 367–368:58–79, 2016. doi:10.1016/j.ins.2016.05.045.

[24] E. Tufte. *The Visual Display of Quantitative Information*. Graphics Press, 2001.

[25] A. van Goethem, I. Kostitsyna, M. van Kreveld, W. Meulemans, M. Sondag, and J. Wulms. The painter's problem: covering a grid with colored connected polygons. In *Graph Drawing and Network Visualization (GD'17)*, volume 10692 of *LNCS*. Springer, 2018. doi:10.1007/978-3-319-73915-1_38.

[26] M. Wertheimer. Untersuchungen zur Lehre von der Gestalt. *Psychologische Forschung*, 4(1):301–350, 1923.