Journal of Graph Algorithms and Applications
http://jgaa.info/ vol. 21, no. 5, pp. 873–913 (2017)
DOI: 10.7155/jgaa.00443

Block Crossings in Storyline Visualizations

Thomas C. van Dijk[1] *Martin Fink*[2] *Norbert Fischer*[1]
Fabian Lipp[1] *Peter Markfelder*[1] *Alexander Ravsky*[3] *Subhash Suri*[2]
Alexander Wolff[1]

[1] Lehrstuhl für Informatik I, Universität Würzburg, Germany
http://www1.informatik.uni-wuerzburg.de/en/staff
[2] University of California, Santa Barbara, USA
[3] Pidstryhach Institute for Applied Problems of Mechanics and Mathematics, National Academy of Sciences of Ukraine, Lviv, Ukraine

Abstract

Storyline visualizations help visualize encounters of the characters in a story over time. Each character is represented by an x-monotone curve that goes from left to right visualizing progression of time. A meeting is represented by having the characters that participate in the meeting run close together for some time. In order to keep the visual complexity low, rather than just minimizing pairwise crossings of curves, we propose to count *block crossings*, that is, pairs of intersecting bundles of lines. In a block crossing, two blocks of parallel lines intersect each other, which is less distracting than the same number of individual crossings being spread over the drawing.

In this paper, we show that minimizing the number of block crossings is NP-hard, even if all meetings are of size 2. For this special case, we present a greedy heuristic, which we evaluate experimentally. We show that the general case is fixed-parameter tractable. Our main results is a constant-factor approximation algorithm for meetings of bounded size. The algorithm is based on (approximately) solving a hyperedge deletion problem on hypergraphs that may be of independent interest.

Submitted:	Reviewed:	Revised:	Reviewed:	Revised:
December 2016	March 2017	May 2017	June 2017	July 2017
	Accepted:	Final:	Published:	
	August 2017	August 2017	October 2017	
	Article type:		Communicated by:	
	Regular paper		Y. Hu and M. Nöllenburg	

A preliminary version of this paper appeared in the Proceedings of the 24th International Symposium on Graph Drawing and Network Visualization (GD 2016) [17].

E-mail addresses: thomas.van.dijk@uni-wuerzburg.de (Thomas C. van Dijk) fink@cs.ucsb.edu (Martin Fink) norbert.fischer@stud-mail.uni-wuerzburg.de (Norbert Fischer) fabian.lipp@uni-wuerzburg.de (Fabian Lipp) peter.markfelder@stud-mail.uni-wuerzburg.de (Peter Markfelder) alexander.ravsky@uni-wuerzburg.de (Alexander Ravsky) suri@cs.ucsb.edu (Subhash Suri) orcid.org/0000-0001-5872-718X (Alexander Wolff)

1 Introduction

A storyline visualization is a convenient abstraction for visualizing the complex narrative of interactions among people, objects, or concepts. The motivation comes from the setting of a movie, novel, or play where the narrative develops as a sequence of interconnected scenes, each involving a subset of characters. See Fig. 1 for an example.

The storyline abstraction of characters and events occurring over time can be used as a metaphor for visualizing other situations, from physical events involving groups of people meeting in corporate organizations, political leaders managing global affairs, and groups of scholars collaborating on research to abstract co-occurrences of "topics" such as a global event being covered on the front pages of multiple leading news outlets, or different organizations turning their attention to a common cause.

A storyline visualization maps a set of characters of a story to a set of curves in the plane and a sequence of meetings between the characters to regions in the plane where the corresponding curves come close to each other. While Minard's visualization of Napoleon's Russian campaign [11] can be seen as an early (and extremely stark) form of storyline visualization (combining time and location on a map), the current form of storyline visualizations seems to have been invented by Munroe [14] (compare Fig. 1), who used it to visualize, in a compact way, which subsets of characters meet over the course of a movie. Each character is shown as an x-monotone curve. Meetings occur at certain times from left to right. A meeting corresponds to a point in time where the characters that meet are next to each other with only small gaps between them. Munroe highlights meetings by underlying them with a gray shaded region, while we use a vertical line for that purpose. Hence, a storyline visualization can be seen as a drawing of a hypergraph whose vertices are represented by the curves and whose edges come in at specific points in time.

A natural objective for the quality of a storyline visualization is to minimize unnecessary *crossings* among the character lines. The number of crossings alone, however, is a poor measure: two blocks of "locally parallel" lines crossing each other are far less distracting than an equal number of crossings randomly scattered throughout the drawing. Therefore, instead of pairwise crossings, we

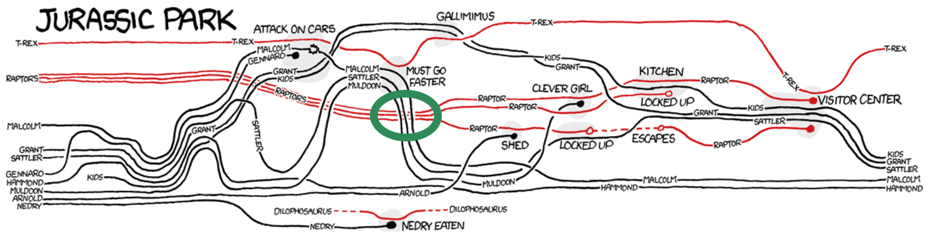


Figure 1: Storyline visualization for *Jurassic Park* by `xkcd` [14] with several block crossings (one of which we highlighted by a bold green ellipse).

focus on minimizing the number of *block crossings*; each block crossing involves two arbitrarily large sets of locally parallel lines forming a crossbar, with no other line in the crossing area; see Fig. 1 for an example.

Previous Work. The InfoVis community has quickly embraced Munroe's effective visualization technique. Kim et al. [8] used storylines to visualize genealogical data; meetings correspond to marriages and special techniques are used to indicate child–parent relationships. Tanahashi and Ma [15] computed storyline visualizations automatically and showed how to adjust the geometry of individual lines to improve the aesthetics of their visualizations. Muelder et al. [13] visualized clustered, dynamic graphs as storylines, summarizing the behavior of local networks that surround user-selected foci.

Only recently, a more theoretical and principled study was initiated by Kostitsyna et al. [10], who considered the problem of minimizing pairwise crossings (that is, *not* block crossings) in storylines. They proved that the problem is NP-hard in general, and showed that it is fixed-parameter tractable (FPT) with respect to the (total) number of characters. Their FPT algorithm runs in time $O(k!^2 k \log k + k!^2 n)$ and uses space $O(k!^2 + n)$, where k is the number of characters and n is the number of meetings. For the special case of 2-character meetings without repetitions where the meeting graph—whose edges describe the meetings of pairs of characters—is a tree, they showed that $\Theta(k \log k)$ crossings always suffice and are sometimes necessary.

Very recently, Gronemann et al. [7] formulated an integer linear program that minimizes the total number of crossings for storylines with meetings of arbitrary cardinality. Their approach can solve instances with 10–20 characters and up to about 50 meetings optimally in a few seconds. Due to the nature of integer linear programming, however, their approach becomes unusable for large instances.

Our work builds on the problem formulation of Kostitsyna et al. [10] but we considerably extend their results by designing (approximation) algorithms for general meetings—for a different optimization goal: we minimize the number of *block crossings* rather than the number of pairwise line crossings. Block crossings were introduced by Fink et al. [5] for visualizing metro maps; in their setting, block crossings happen between metro lines that run on top of an embedded graph, the metro network.

Problem Definition. While a block crossing is visually easily recognizable by a human reader, we start by carefully defining the problem. In the course of doing so, we will already gain further insight in what minimizing the number of block crossings amounts to. A storyline $\mathcal{S}$ is a pair (C, M) where $C = \{1, \ldots, k\}$ is a set of *characters* and $M = [m_1, m_2, \ldots, m_n]$ with $m_i \subseteq C$ and $|m_i| \geq 2$ for $i = 1, 2, \ldots, n$ is a sequence of *meetings* of at least two characters. We call any set $g \subseteq C$ of characters that has at least one meeting, a *group* (i.e., $g = m_i$ for some $1 \leq i \leq n$). We define the *group hypergraph* $\mathcal{H} = (C, \Gamma)$ as follows: its vertices are the characters and its hyperedges are the groups. This hypergraph does

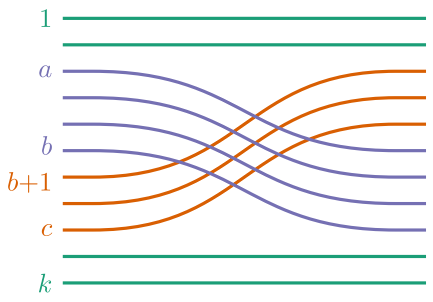


Figure 2: Block crossing (a, b, c)

not include the temporal aspect of the storyline—it models only the structure of groups participating in the storyline. Each group is represented by a single hyperedge, even if it meets multiple times. The group hypergraph can be built by sorting M and filtering the multiply-occurring meetings. This can be done in $O(nk)$ time by representing the meetings as bitsets and using radix sort.

Note that we do not encode the exact times of the meetings: in a given visualization, at any time t, there is a unique vertical order π_t of the characters. Without changing π_t by crossings, we can increase or decrease vertical gaps between lines. If a group g forms a contiguous interval in π_t, then we can bring the lines in g within a short vertical distance δ_{group} without any crossing, and also make sure that all other lines are at a distance of $\delta_{\text{sep}} > \delta_{\text{group}}$; see Fig. 12. Since any group must be supported at a time just before its meeting starts, computing an output drawing consists mainly of changing the permutation of characters over time so that during a meeting its group is supported by the current permutation. We therefore focus on changing the permutation by crossings over time, and only have to be concerned about the order of meetings; the final drawing can be obtained by a simple post-processing from this discrete set of permutations.

Suppose that we locally renumber the characters from top to bottom from 1 to k. Then we can define, for $a \leq b < c$, a *block crossing* (a, b, c) to be the exchange of the two consecutive blocks $\langle a, \dots, b\rangle$ and $\langle b+1, \dots, c\rangle$. This exchange maps the permutation $\langle 1, \dots, a, \dots, b, \dots, c, \dots, k\rangle$ to the permutation $\langle 1, \dots, a-1, b+1, \dots, c, a, \dots, b, c+1, \dots, k\rangle$; see Fig. 2 for an illustration. The same definition was used by Fink et al. [5].

A meeting m *fits* a permutation π (or a permutation π *supports* a meeting m) if the characters participating in m form an interval in π. In other words, there is a permutation of m that is part of π. If we apply a sequence B of block crossings to a permutation π in the given order, we denote the resulting permutation by $B(\pi)$.

Problem 1 (Storyline Block Crossing Minimization (SBCM)) *Given a storyline instance (C, M) with $M = [m_1, m_2, \dots, m_n]$, find a solution consisting of a start permutation π_0 of C and a sequence $B = (B_1, B_2, \dots, B_n)$ of (possibly empty) sequences of block crossings such that the total number of block crossings is minimized and $\pi_i = B_i(\pi_{i-1})$ supports m_i for $1 \leq i \leq n$.*

Note that in our problem definition no characters arrive after the start of the drawing or disappear before the end; this is different in the example in Fig. 1. We also consider d-SBCM, a special case of SBCM where meetings involve groups of size at most d, for an arbitrary constant d. For example, 2-SBCM allows only 2-character meetings, a setting that was also studied by Kostitsyna et al. [10].

Our Results. We observe that a storyline has a crossing-free visualization if and only if its group hypergraph is an *interval hypergraph*, that is, if there exists a permutation of the character set such that each hyperedge corresponds to a contiguous block of characters in this permutation. This is equivalent to the hypergraph having *path support* [1]. A hypergraph can be tested for the interval property in $O(n^2)$ time, where n is the number of hyperedges. We show that 2-SBCM is NP-hard (see Sect. 3) and that SBCM is fixed-parameter tractable with respect to k (Sect. 4). One of our FPT algorithms can be modified to handle pairwise crossings instead of block crossings. The modified algorithm is faster than the algorithm of Kostitsyna et al. [10] by a factor of $k!$. We carry out some experiments evaluating this algorithm and other approaches in a follow-up paper [18].

The case of size-2 meetings is of interest; recall that Kim et al. [8] have used a storyline-like visualization to trace genealogical data. We investigate structural properties and present a greedy algorithm for 2-SBCM that runs in $O(k^3 n)$ time for k characters. For $k = 3$, the greedy algorithm yields optimal solutions. We experimentally compare greedy solutions to optimal solutions; see Sect. 5. One main result is a constant-factor approximation algorithm for d-SBCM for the case that d is bounded and that meetings cannot be repeated; see Sect. 6.

Our algorithm is based on a solution for the following NP-complete hypergraph problem, which may be of independent interest: given a hypergraph $\mathcal{H}$, delete the minimum number of hyperedges so that the remainder is an interval hypergraph. For this problem, we develop a $(d+1)$-approximation algorithm, where d is the maximum cardinality of a hyperedge in $\mathcal{H}$; see Sect. 7. Finally, we list some open problems in Sect. 8.

2 Preliminaries and Basic Observations

First, we consider the special case where every meeting consists of two characters. For these restricted instances, any meeting can be realized from any permutation by a single block crossing.

Observation 1 *Given an instance of 2-SBCM, there is a solution with at most one block crossing before each of the meetings. In particular, there is a solution with at most n block crossings in total.*

Proof: Let π' be an arbitrary permutation, and let $m = \{c, c'\} \in M$ be the next meeting. Let i and j be the positions of the characters c and c', respectively, in the permutation π', that is, $\pi'(i) = c$ and $\pi'(j) = c'$. Without loss of generality,

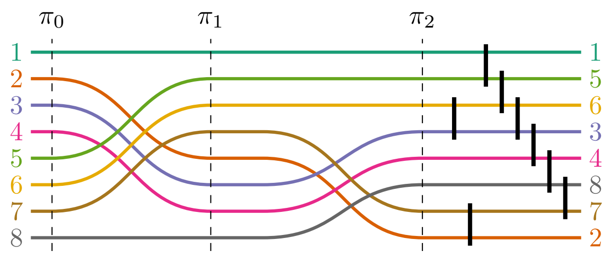


Figure 3: Optimal solution for $\mathcal{S}$ from the proof of Lemma 1.

assume $i < j$. If π' does not support m, we can realize it using the block crossings $(i, i, j-1)$, that is, moving the line of c directly above that of c'. $\square$

Observation 1 shows that there is a solution with at most one block crossing before any meeting. This raises the question whether there is also an *optimal* solution that fulfills this condition. The answer is negative.

Lemma 1 *There is an instance $\mathcal{S}$ of 2-SBCM and a start permutation π_0 such that there is no optimal solution (π_0, B) of $\mathcal{S}$ that starts with π_0 and uses at most one block crossing before the first and between each pair of consecutive meetings.*

Proof: We prove the claim by contradiction. Consider the instance $\mathcal{S} = (C, M)$ with

$$C = \{1,2,3,4,5,6,7,8\} \text{ and}$$
$$M = [\{6,3\}, \{7,2\}, \{1,5\}, \{5,6\}, \{6,3\}, \{3,4\}, \{4,8\}, \{8,7\}].$$

Let $\pi_0 = \langle 1,2,3,4,5,6,7,8\rangle$ be the start permutation. There is a solution with only two block crossings, namely (π_0, B) with $B = [(2,4,7), (4,5,8)]$; see Fig. 3. Let π_1 be the permutation after the first block crossing of B on π_0, and π_2 the permutation after both block crossings. The permutation π_2 supports all meetings in M. The first meeting $\{6,3\}$ in M fits neither π_0 nor π_1, that is, both block crossings must occur before the first meeting.

Now assume that there is another solution (π_0, B') with $|B'| \le 2$ that has at most one block crossing before each meeting. Starting from π_0 there are exactly nine feasible block crossings that allow the first meeting. They yield the following permutations (where the bars indicate the crossing blocks):

- $\langle 1,2,\overline{4,5,6},\overline{3},7,8\rangle$
- $\langle 1,2,\overline{5,6},\overline{3,4},7,8\rangle$
- $\langle 1,2,\overline{6},\overline{3,4,5},7,8\rangle$
- $\langle \overline{4,5},\overline{1,2,3},6,7,8\rangle$
- $\langle 1,\overline{4,5},\overline{2,3},6,7,8\rangle$
- $\langle 1,2,\overline{4,5},\overline{3},6,7,8\rangle$
- $\langle 1,2,3,\overline{6},\overline{4,5},7,8\rangle$
- $\langle 1,2,3,\overline{6,7},\overline{4,5},8\rangle$
- $\langle 1,2,3,\overline{6,7,8},\overline{4,5}\rangle$

None of these permutations supports the second meeting $\{7,2\}$. So we need the second block crossing before this meeting. This second block crossing needs to prepare all of the remaining meetings because otherwise $|B'| > 2$. These meetings can only be supported by the permutation $\sigma = \langle 1,5,6,3,4,8,7,2\rangle$ or

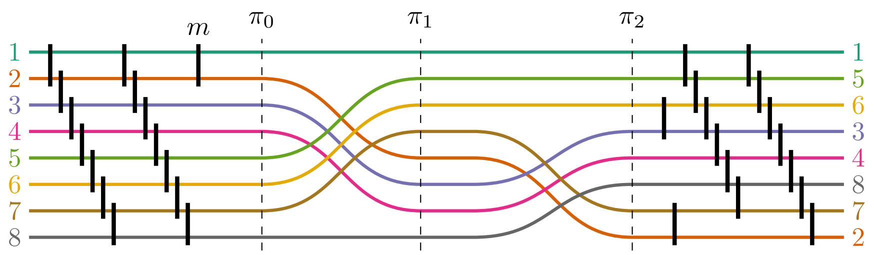


Figure 4: Optimal solution for $\mathcal{S}$ from the proof of Theorem 1.

its reverse permutation σ^{R}. It remains to show that none of the permutations yielded by the feasible first block crossing can be transformed to σ or σ^{R} by one additional block crossing. All permutations containing $\langle 3, 6\rangle$ as a subsequence are infeasible because there is only one block crossing that swaps two neighboring characters and it does not produce σ. For permutations starting with $\langle 1, 2\rangle$, there is only one possible block crossing to bring character 2 to the end of the permutation while character 1 stays at the first position, which also does not yield σ. Similarly, we can show that there is also no block crossing after any of the feasible block crossings for the first step that leads to σ^{R}. □

Now we modify this instance such that we do not need to prescribe the start permutation.

Theorem 1 *There is an instance $\mathcal{S}$ of 2-SBCM such that there is no optimal solution (π_0, B) of $\mathcal{S}$ that uses at most one block crossing between each pair of consecutive meetings.*

Proof: We prove the claim by constructing an example. Consider the instance $\mathcal{S} = (C, M)$ depicted in Fig. 4. The instance has no solution with less than two block crossings, but there is a solution with two block crossings, namely (π_0, B) with start permutation $\pi_0 = \langle 1, 2, 3, 4, 5, 6, 7, 8\rangle$ and block crossings $B = [(2, 4, 7), (4, 5, 8)]$; see Fig. 4. Let π_1 be the permutation after the first block crossing of B on π_0, and let $\pi_2 = \langle 1, 5, 6, 3, 4, 8, 7, 2\rangle$ be the permutation after the second block crossing in B. The permutation π_2 supports all meetings in the right half of M. The first occurrence of meeting $\{6, 3\}$ in M (right after the third dashed line in Fig. 4) fits neither π_0 nor π_1, that is, both block crossings occur before that meeting.

Now we argue that there is no solution with two block crossings that are separated by at least one meeting. Let m be the third occurrence of meeting $\{1, 2\}$ (right before the first dashed line in Fig. 4). At some point in time before m, either permutation π_0 or π_0^{R} must occur, otherwise we would need at least two block crossings to realize the meetings before m and at least one block crossing after m to realize the last seven meetings (because the permutation at m is not π_2 or π_2^{R}).

We distinguish three cases for the two block crossings.

a) Both crossings occur before m: This is not possible because the remaining meetings cannot be realized with characters 1 and 2 being adjacent.

b) Both crossings occur after m: Then the start permutation needs to be π_0 or π_0^{R}, which is exactly the case handled in Lemma 1.

c) The first crossing is before m, the second one after m. Since we must realize the last seven meetings with only one block crossing, the final permutation is π_2 or π_2^{R}. Assume that it is π_2. Then going backwards from π_2, we have to realize m by one block crossing. There are exactly nine such block crossings yielding the following permutations:

- $\langle\overline{5,6,3,4,8,7,2},\overline{1}\rangle$
- $\langle\overline{4,8,7,2},\overline{1,5,6,3}\rangle$
- $\langle\overline{2},\overline{1,5,6,3,4,8,7}\rangle$
- $\langle\overline{6,3,4,8,7,2},\overline{1,5}\rangle$
- $\langle\overline{8,7,2},\overline{1,5,6,3,4}\rangle$
- $\langle 1,\overline{2},\overline{5,6,3,4,8,7}\rangle$
- $\langle\overline{3,4,8,7,2},\overline{1,5,6}\rangle$
- $\langle\overline{7,2},\overline{1,5,6,3,4,8}\rangle$
- $\langle\overline{5,6,3,4,8,7},\overline{1},2\rangle$

It is easy to check that none of them contains as a subinterval what we call a *triplet*; three consecutive numbers in their natural (or inverted) order (for instance, $\langle 3,4,5\rangle$ or $\langle 5,4,3\rangle$). Recall that at some point in time before m, permutation π_0 or π_0^{R} must occur. Now note that any permutation reachable from π_0 or π_0^{R} by one block crossing contains a triplet (since we have eight characters). Thus we obtain a contradiction. The case when the final permutation is π_2^{R} can be treated similarly. $\square$

Detecting Crossing-Free Storylines. If a storyline admits a crossing-free visualization, then the vertical permutation of the character lines remains the same over time, and all meetings involve groups that form contiguous subsets in that permutation. (The visualization can be obtained by placing characters along a vertical line in the correct permutation and for each meeting bringing its lines together for the duration of the meeting and then separating them apart again.) In other words, a single permutation supports each group of $\mathcal{H} = (C, \Gamma)$. This holds if and only if $\mathcal{H}$ is an *interval hypergraph.* Recall that this is the case if there exists a permutation $\pi = \langle v_1, \dots, v_k\rangle$ of C such that each hyperedge $e \in \Gamma$ corresponds to a contiguous block of characters in this permutation.

Note that a graph is an interval hypergraph if and only if all of the graph's connected components are paths. Hence, a graph being an interval hypergraph is *not the same* property as being an interval graph.

An interval hypergraph can be visualized by placing all of its vertices on a line, and drawing each hyperedge e as an interval that includes all vertices of e and no vertex of $V \setminus e$. Checking whether a k-vertex hypergraph is an interval hypergraph takes $O(k^2)$ time [16]. Recall that we can build $\mathcal{H}$ in $O(nk \log n)$ time.

Theorem 2 *Given the group hypergraph $\mathcal{H}$ of an instance of SBCM with k characters, we can check in $O(k^2)$ time whether a crossing-free solution exists. If this is the case, a permutation realizing a crossing-free solution can also be found within the same time bound.*

For 2-SBCM we only need to check (in $O(k)$ time) whether $\mathcal{H}$ is a collection of vertex-disjoint paths; this is dominated by the time ($O(n)$) for building $\mathcal{H}$, as long as $k = O(n)$.

3 NP-Completeness of SBCM

In this section we prove that SBCM is NP-complete. This is known for the problem Block Crossing Minimization (BCM), introduced by Fink et al. [5]. But SBCM is not simply a generalization of BCM because in SBCM we can choose an arbitrary start permutation. Therefore, the idea of our hardness proof is to force a certain start permutation by adding some characters and meetings. We reduce from Sorting by Transpositions (SBT), which has also been used to show the hardness of BCM [5]. In SBT, one is given a permutation π and an integer k, and the task is to decide whether there is a sequence of transpositions (which are equivalent to block crossings) of length at most k that transforms π to the identity. SBT was recently shown NP-hard by Bulteau et al. [2].

We show hardness for 2-SBCM, which implies that SBCM is NP-hard, too. It is easy to see that the decision version of SBCM is in NP: Obviously, the maximum number of block crossings needed for any number of characters and meetings is bounded by a polynomial in k and n. Therefore also the size of a possible certificate is bounded by a polynomial. To test the feasibility of a solution efficiently, we simply test whether the permutations between the block crossings support the meetings in the right order from left to right. We will use the following obvious fact.

Observation 2 *If permutation π needs c block crossings to be sorted, any permutation containing π as a subsequence needs at least c block crossings to be sorted.*

Theorem 3 *2-SBCM is NP-complete.*

Proof: It remains to show the NP-hardness. We reduce from SBT. Given an instance of SBT, that is, a permutation π of $\{1, \dots, k\}$, we show how to construct a corresponding instance of 2-SBCM.

We extend the set of characters $\{1, 2, \dots, k\}$ to $C = \{1, \dots, k, c_1, c_2, \dots, c_{2k}\}$. Correspondingly, we extend the permutation $\pi = \langle \pi(1), \pi(2), \dots, \pi(k) \rangle$ to $\pi' = \langle c_1, \dots, c_{2k}, \pi(1), \dots, \pi(k) \rangle$ and ι to $\iota' = \langle c_1, c_2, \dots, c_{2k}, 1, 2, \dots, k \rangle$. Let $M_{\pi'}$ and $M_{\iota'}$ be the sequences of meetings of all neighboring pairs in π' and ι', respectively. Let M_1 and M_2 be the concatenations of $k + 1$ copies of $M_{\pi'}$ and $M_{\iota'}$, respectively. This yields the instance $\mathcal{S} = (C, M)$ of 2-SBCM, where M is the concatenation of M_1 and M_2; see Fig. 5.

We show that the number of block crossings needed for the 2-SBCM instance $\mathcal{S}$ equals the number of transpositions to solve instance π of SBT, that is, to transform π to the identity $\iota = \langle 1, 2, \dots, k \rangle$. Note that π can be sorted by at most k block crossings. So k is an upper bound for an optimal solution of instance π of SBT.

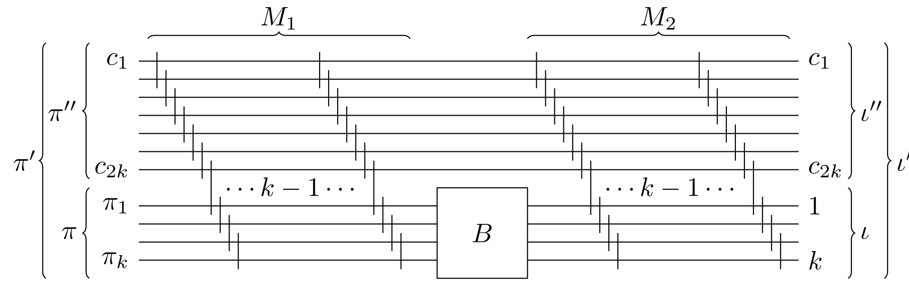


Figure 5: Solution for the 2-SBCM instance $\mathcal{S}$ corresponding to a solution B of instance π of SBT. The box B represents the block crossings.

First, let B be a shortest sequence of block crossings to sort π. Then, (π', B) is a feasible solution for $\mathcal{S}$. The start permutation π' supports all meetings in M_1 without any block crossing. Using B, the lines are sorted to ι', and this permutation supports all meetings in M_2 without any further block crossings; see Fig. 5. Hence, the number of block crossings in any solution of π is an upper bound for the minimum number of block crossings needed for $\mathcal{S}$.

For the other direction, let (π^*, B^*) be an optimal solution for $\mathcal{S}$. In the input of SBCM, we do not fix a certain order of the characters. So any solution of 2-SBCM gives rise to a symmetric solution that is obtained by reversing the order of the characters. In the following, without loss of generality, we assume that π' (rather than the reverse permutation π'^{R}) *occurs* in M, that is, there is a time t in the visualization such that the vertical order of the characters at t is described by π'.

Next, we show that the start permutation π' occurs somewhere in M_1 and that ι' occurs somewhere in M_2. If there is a sequence $M_{\pi'}$ of meetings between which there is no block crossing, the permutation at this position can only be the start permutation π' or its reverse. For a contradiction, assume that π' does not occur during M_1 in the layout induced by (π^*, B^*). Then there is no such sequence without any block crossing in it. As this sequence is repeated $k+1$ times, the solution would need at least $k+1$ block crossings. This contradicts our upper bound, which is k. Analogously, we can show that the permutation ι' or its reverse occurs in M_2.

We now want to show that the unreversed version of ι' occurs in M_2. For a contradiction, assume the opposite. We forget about the lines $1, \dots, k$ and only consider the sequence $\pi'' = \langle c_1, \dots, c_{2k} \rangle$ in π' which is reversed to $\iota''^{\mathrm{R}} = \langle c_{2k}, \dots, c_1 \rangle$ in ι'^{R}. Eriksson et al. [4] showed that we need $\lceil (l+1)/2 \rceil$ block crossings to reverse a permutation of l elements. This implies that we need $k+1$ block crossings to transform π'' to ι''^{R}. As π' and ι'^{R} contain these sequences as subsequences, Observation 2 implies that the transformation from π' to ι'^{R} also needs at least $k+1$ block crossings. As the optimal solution uses at most k block crossings, we know that we cannot reach ι'^{R} and thus the sequence of permutations contains π' and ι'.

The sequence of block crossings that transforms π' to ι' yields a sequence B of block crossings of the same length that transforms π to ι. Therefore the length of a solution for $\mathcal{S}$ is an upper bound for the length of an optimal solution of the corresponding SBT instance π. Thus, the two lengths are equal. $\square$

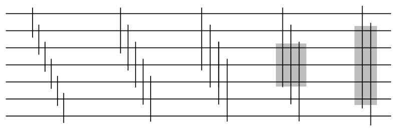
Hardness Without Repetitions. With arbitrarily large meetings, we can slightly modify our hardness proof, and show that minimizing the number of block crossings is also hard without repeating the same meeting many times. The idea to change our reduced SBCM instance, is to replace the repeated sequence of 2-character meetings so that in each repetition the group size is increased by one for all meetings; see Fig. 6.

Due to the overlapping structure of the groups in a single sequence, they can only be all supported at the same time if also the 2-character meetings that they replaced are supported. The only thing that we have to be careful about is that when the groups get larger than $3k/2$, that is, half of the number of characters, there is a growing set of characters in π' (or ι') that are contained in exactly the same groups, and their relative order does not matter for the meetings; see Fig. 6(a). We will avoid this situation in the following way (shown in Fig. 6(b)).

Since we have $k+1$ sequences of repeated meetings at the beginning as well as at the end of the timeline, and we keep increasing the group sizes, we have groups of $2k+3$ characters in the end. We replace $c_1, \dots, c_{2k}$ by a new sequence $c_1, \dots, c_{10k}$ of characters without changing the structure further. Then, we can increase the group size up to $2k+3$ while in the end still less than half of all characters are involved in each group. Since the growing meetings completely simulate the desired 2-character meetings, the rest of the reduction and its proof stay the same, and we get the following result.

Theorem 4 *SBCM is NP-hard even if meetings are not repeated.*

Note that in this reduction (different from Theorem 3) we use meetings whose size depends on the input. So for this variant without repeated meetings we do not show hardness for d-SBCM for any fixed d.


(a) Instance in which the groups are too large for the number of characters. The area highlighted in gray shows the characters involved in all meetings of a certain size. Therefore, the relative order of these characters does not matter anymore.

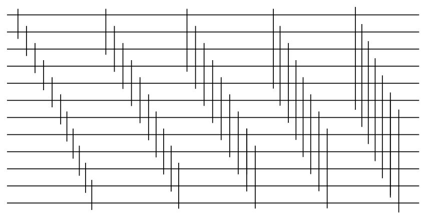

(b) By adding characters (and the corresponding meetings), we avoid characters being involved in every meeting.

Figure 6: We simulate repeated 2-character meetings by using groups of increasing size.

4 Exact Algorithms

We present two exact algorithms for SBCM. Conceptually, both build up a sequence of block crossings while keeping track of how many meetings have already been accomplished. The first uses polynomial space; the second improves the runtime at the cost of exponential space. The latter generalizes to improve a result by Kostitsyna et al. [10] about minimizing pairwise crossings (rather than block crossings).

We start with a data structure to keep track of permutations, block crossings and meetings. It is initialized with a given permutation and has two operations. The CHECK operation returns whether a given meeting fits the current permutation. The BLOCKMOVE operation performs a given block crossing on the permutation and then returns whether the most-recently CHECKed meeting now fits.

Lemma 2 *A sequence of arbitrarily interleaved* BLOCKMOVE *and* CHECK *operations can be performed in $O(\beta + \mu)$ time, where β is the number of calls to* BLOCKMOVE *and μ is sum of cardinalities of the meetings given to* CHECK*. The space usage is $O(k)$, where k is the number of characters.*

Proof: Represent the permutation as a doubly-linked list.[1] First consider a 2-meeting. It takes constant time to check whether it fits: check the previous/next pointers. Since a block crossing changes at most 6 adjacencies, BLOCKMOVE can update the linked list in constant time. (This requires giving out pointers into the linked-list representation, since we need to find the characters at the edges of the block in constant time.) Note that all possible block crossings can be enumerated in constant time each.

Now we look at a meeting of cardinality m. Interpret the linked list as a path and consider the subgraph induced by the nodes in the meeting. If the meeting fits the permutation, this subgraph is connected and, being a path, has $m - 1$ edges; if the meeting does not fit, this subgraph has more components and therefore fewer edges. The CHECK operation on a meeting of size m can be performed in $O(m)$ time by counting at every node in the meeting whether zero, one or two of its neighbors are also in the meeting. For the amortized runtime over a sequence of operations, remember this count: BLOCKMOVE can update it in constant time, since again at most six adjacencies change.

In terms of space, there is only the doubly linked list and the count. □

Now we provide an output-sensitive algorithm for SBCM, the runtime of which depends on the number of block crossings required by the optimum.

Theorem 5 *An instance $\mathcal{S} = (C, M)$ of SBCM can be solved in $O(k! \cdot (\frac{k^3-k}{6})^\beta \cdot (\beta + \mu))$ time and $O(\beta k)$ working space if a solution with β block crossings exists, where $\mu = \sum_{m \in M} |m|$.*

[1] We assume that the meetings given to CHECK are represented by references to the nodes in this list; if necessary, this representation can be constructed efficiently in preprocessing.

Proof: Consider a branching algorithm that starts from a permutation of the characters and keeps trying all possible block crossings. The instance of SBCM is solved by finding the shortest sequence of block crossings that supports the meetings.

A block crossing can be represented by indices (a, b, c) with $1 \leq a \leq b < c \leq k$. The number of possible block crossings is given by adding the number of block crossings with $a = b$ and the number of block crossings with $a \neq b$; hence, there are $\binom{k}{2} + \binom{k}{3} = \frac{k^3-k}{6}$ distinct block crossings on a permutation of length k. We can enumerate these in constant time each by enumerating all appropriate triples (a, b, c).

We use depth-first iterative-deepening search [9] from all possible start permutations, applying block crossings, until we find a sequence of permutations that fulfills all meetings. While searching, the algorithm keeps track of how many meetings fit the current sequence of permutations using the data structure from Lemma 2. Correctness follows from the iterative deepening of the search, since we want an (unweighted) shortest sequence of block crossings. The runtime and space bounds follow from the standard analysis of iterative-deepening search, observing that a node uses $O(k)$ space and it takes $O(\beta + \mu)$ time in total to evaluate a path from root to leaf. $\square$

Note that μ is $O(kn)$: there are n meetings and each consists of at most k characters.

At the cost of exponential space, we can improve the runtime and get rid of the dependence on β, showing the problem to be fixed parameter linear for k. The following algorithm can easily be adapted to optimize pairwise crossings rather than block crossings. In that case we improve upon the algorithm of Kostitsyna et al. [10] by a factor of roughly $k!$, in terms of both running time and space consumption: recall that their FPT algorithm runs in $O(k!^2 k \log k + k!^2 n)$ time and uses $O(k!^2 + n)$ space.

Theorem 6 *An instance of SBCM can be solved in $O(k! \cdot k^3 \cdot n)$ time and $O(k! \cdot k \cdot n)$ space.*

Proof: The algorithm is based on the following question: how many block crossings are required for the first ℓ meetings given that we end on permutation π? Formally, let $f(\pi, \ell)$ be the optimal number of block crossings in a solution to the given instance when restricted to the first ℓ meetings and to have π as its final permutation. Note that by definition the solution for the actual instance is given by $\min_{\pi^*} f(\pi^*, n)$, where the minimum ranges over all possible permutations. As a base case, $f(\pi, 0) = 0$ for all π, since the empty set of meetings is supported by any permutation. Let π and π' be permutations that are one block crossing apart and let $0 \leq \ell \leq \ell'$. If the meetings $\{m_{\ell+1}, \ldots, m_{\ell'}\}$ all fit π', then we have $f(\pi', \ell') \leq f(\pi, \ell) + 1$: if we can support the first ℓ meetings and end on π, then with one additional block crossing we can support the first ℓ' meetings and end on π'.

We now model the above observation in a graph. Let G be an unweighted directed graph on nodes (π, ℓ), where π is a permutation of characters and

$0 \leq \ell \leq n$. Call a node *start node* if $\ell = 0$. There is an arc from (π, ℓ) to (π', ℓ') if and only if

- π and π' are one block crossing apart,
- $\ell \leq \ell'$, and
- the meetings $\{m_{\ell+1}, \dots, m_{\ell'}\}$ fit π'.

Note that we allow $\ell = \ell'$ since we may need to allow block crossings that do not immediately achieve an additional meeting (cf. Proposition 1), so G is not acyclic. By construction we have that $f(\pi, \ell)$ equals the minimum graph distance from a start node to the node (π, ℓ). Call a path from a start node that realizes this distance *optimal*.

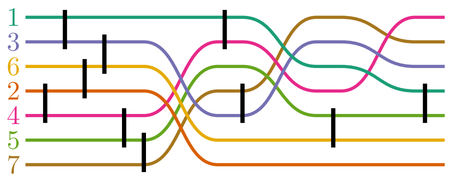
In G, consider any length-3 path $[(\pi_1, \ell_1), (\pi_2, \ell_2), (\pi_3, \ell_3)]$ with strict inequality $\ell_3 > \ell_2$. If meeting $\ell_2 + 1$ fits π_2, then $[(\pi_1, \ell_1), (\pi_2, \ell_2 + 1), (\pi_3, \ell_3)]$ is also a path. Repeating this transformation shows that for all π, the node (π, n) has an optimal path in which every arc maximally increases ℓ. Let G' be the graph where we drop all arcs from G that do not maximally increase ℓ. Note that G' still contains a path that corresponds to the global optimum.

The graph G' has $O(k! \cdot n)$ nodes. Each node has outdegree $O(k^3)$, since any block crossing contributes at most one out-arc to a node. Then a breadth-first search from all start nodes to any node (π^*, n) achieves the claimed time and space bounds, assuming we can enumerate the outgoing arcs of a node in constant time each.

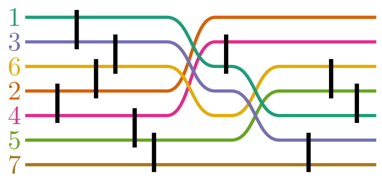
For a given node (π, ℓ) we can enumerate all possible block crossings in constant time each, as before. To generate its outgoing arcs in G', we also need to know the maximum ℓ' such that all meetings $\ell + 1$ up to ℓ' fit π', where π' is the permutation resulting from the block crossing. Note that ℓ' only depends on ℓ and π'; in particular, it does not depend on π. We can therefore precompute a table $M(\pi', \ell)$ that gives this value. Computing $M(\pi', \ell)$ for a given π' and all ℓ takes a total of $O(kn)$ time: first compute for every m_i whether it fits π', then compute the implied 'forward pointers' using a linear scan. So using $O(k! \cdot kn)$ preprocessing time and $O(k! \cdot n)$ space, we have an efficient implementation of the breadth-first search. The theorem follows. $\square$

5 A Greedy Heuristic

In this section we develop an $O(kn)$-time greedy algorithm to quickly draw good storyline visualizations for 2-SBCM. Given an instance $\mathcal{S} = (C, M)$, we reserve a list $B = [\,]$ that the algorithm will use to store the block crossings. The algorithm starts with an arbitrary permutation π_0 of the characters. In every step the algorithm removes all meetings from the beginning of M that are supported by the current permutation π_i. Subsequently, the algorithm picks a block crossing b such that the resulting permutation $\pi_{i+1} = b(\pi_i)$ supports the maximum number of meetings from the beginning of M, and b is appended to the list B. This process repeats until M is empty. The algorithm returns the solution (π_0, B).



(a) Greedy solution



(b) Optimal solution

Figure 7: The greedy algorithm is not optimal.

Recall that, from any particular permutation, there are $O(k^3)$ block crossings. To find the appropriate block crossings, the algorithm could simply check all of them. However, most of those will result in permutations that do not even support the next meeting, which cannot be the greedy choice described above. Hence, our algorithm enumerates only the relevant block crossings in the following sense: block crossings yielding a permutation that supports the upcoming meeting. Let $\{c, c'\}$ be the next meeting in M. If x and y are the positions of c and c' in the current permutation (i.e., $\pi_i(x) = c$ and $\pi_i(y) = c'$; without loss of generality, assume $x < y$), the relevant block crossings are:

$$\{(z, x, y-1)\colon 1 \le z \le x\} \cup \{(x, z, y)\colon x \le z < y\} \cup \{(x+1, y-1, z)\colon y \le z \le k\}.$$

We see that the number of relevant block crossings in each step is $k + 1$. Let n_i be the maximum number of meetings at the beginning of M we can support by one of these block crossings. We use the data structure in Lemma 2 and check for each relevant block crossing how many meetings can be done with this permutation. Hence, we can identify a block crossing achieving the maximum number in $O(kn_i)$ time since we have to check $k + 1$ options, each lasting up to n_i meetings each. The numbers of meetings n_i in each iteration of the algorithm sum up to n and therefore the algorithm runs in $O(kn)$ total time.

In the description above, this greedy algorithm starts with an arbitrary permutation. Instead, we could start with a permutation that supports the maximum number of meetings before the first block crossing needs to be done. In other words, we could to find a maximal prefix M' of M such that (C, M') can be represented without any block crossings. We can find M' in $O(kn)$ time: start with an empty graph on the characters and successively add edges for the meetings. After each addition we check whether the graph is still a collection of paths, which can be done in $O(k)$ time. After this process terminates, we construct a permutation that supports all meetings in M', which is easy given the collection of paths. See Fig. 7 for an example that uses the heuristic start permutation. While this is a sensible heuristic, we do not prove that it reduces the total number of block crossings. Indeed, we experimentally observe that, while this heuristic for the start permutation is generally good, it is not always the best; this is discussed in the experimental evaluation later in this section.

The greedy algorithm actually yields optimal solutions for special cases of 2-SBCM. For the following lemma, we assume that no two subsequent meetings in the input are the same. We call an instance *normal* if this is the case. An instance can be normalized by simply dropping the repeated meetings. This does not affect the optimal number of block crossings or the behavior of the greedy algorithm, but note that it does lower n.

Lemma 3 *A normal instance of 2-SBCM with $k = 3$ can be solved using at most $\lceil n/2 \rceil - 1$ block crossings.*

Proof: Note that there are only three possible meetings, namely $\{1, 2\}$, $\{1, 3\}$, and $\{2, 3\}$. Any permutation supports precisely two of these and not the third, and is equivalent in this sense to its reverse. For example, the permutation $\langle 1, 2, 3 \rangle$ and its reverse both support the meetings $\{1, 2\}$ and $\{2, 3\}$, but not $\{1, 3\}$. Let π and π' be distinct permutations. Case distinction shows that it is always possible with a single block crossing to get from π to either π' or to the reverse of π'.

For the analysis, we partition the sequence of meetings into *epochs* as follows. We start from the first meeting and keep going until the third distinct meeting occurs: these meetings form the first epoch. That is, an epoch alternates between two different meetings. Repeating this process partitions the entire sequence of meetings into epochs, possibly with a single remaining meeting as final epoch. A solution can choose the start permutation π_0 that supports the first epoch. After that it can always get to a permutation that supports the entire next epoch with one block crossing. (Note that the greedy algorithm uses exactly this strategy.) In the worst case all epochs have length 2, and we need $\lceil n/2 \rceil - 1$ block crossings. □

Since the greedy algorithm can choose an arbitrary start permutation it does not need to perform a block crossing for the first sequence of meetings because it can build a start permutation using the same strategy as described above. This leaves us with at most $\lceil n/2 - 1 \rceil$ block crossings.

Theorem 7 *For $k = 3$, the greedy algorithm produces optimal solutions.*

Proof: We look at the epochs from Lemma 3 again. The greedy algorithm produces one block crossing fewer than the number of epochs.

Consider any epoch except the last one and include the meeting after it. By construction, this is the third distinct meeting and therefore these meetings together cannot fit a single permutation. Then in any solution to the problem, a block crossing must occur after at least one of the meetings in the epoch. This holds for all epochs except the last one and since they are disjoint, the number of epochs reduced by one is a lower bound for the optimum number of block crossings. The result of the greedy algorithm realizes this bound. □

We note here that the greedy algorithm does not sensibly generalize to SBCM with arbitrary meetings. Consider the step that finds a block crossing that supports the maximum number of subsequent meetings. With arbitrary meetings (as opposed to only cardinality-2 meetings), it may be the case that this maximum is zero—that is, there may not exist any single block crossing that supports the next meeting. In this case our algorithm has no greedy way make progress.

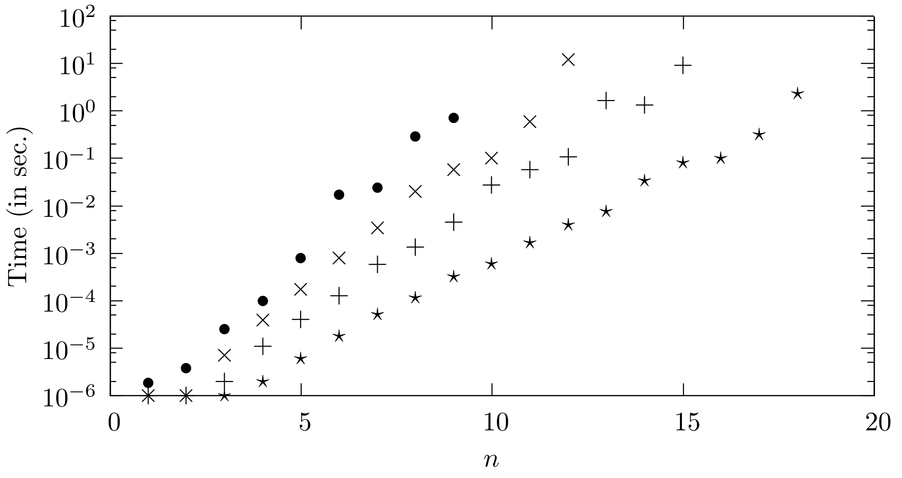


Figure 8: Runtime of the exact algorithm of Theorem 5 on random instances with $k = 4(\star), 5(+), 6(\times), 7(\bullet)$. Each data point is the average of 50 random instances.

Experimental Evaluation. In this section, we report on some preliminary experimental results for 2-SBCM. We have generated random instances as follows. Given n and k, we pick n pairs of characters as meetings, uniformly at random using rejection sampling to ensure that consecutive meetings are different. (This means that the generated instances are normal in the sense of Lemma 3.)

First, we consider the exact algorithm of Theorem 5. As expected, its runtime depends heavily on k (Fig. 8). Perhaps somewhat unexpectedly, we observe exponential runtime in n. This is actually a property of our random instances, in which β tends to increase linearly with n. Note that this experiment does not invalidate the algorithm since in practical applications we may be interested mainly in instances for which β is indeed small.

We have also generated instances with small solutions as follows. Pick k, n and β, then sample a uniformly-random start permutation and β uniformly-random block crossings. Consider the sequence of permutations resulting from these block crossings. We randomly sample n meetings by picking, for each one independently, one of the permutations at random and then two adjacent characters from this permutation. An instance constructed in this way might have a solution with fewer than β block crossings, but by construction the optimum is at most β. On these instances, the runtime of the algorithm of Theorem 5 scales as expected. Since the construction of these instances is somewhat complicated and arbitrarily, we now return to the more natural first type of random instances.

The exact algorithm is feasible only for rather small instances, so we now shift our focus to the greedy algorithm. Recall that it starts with an arbitrary

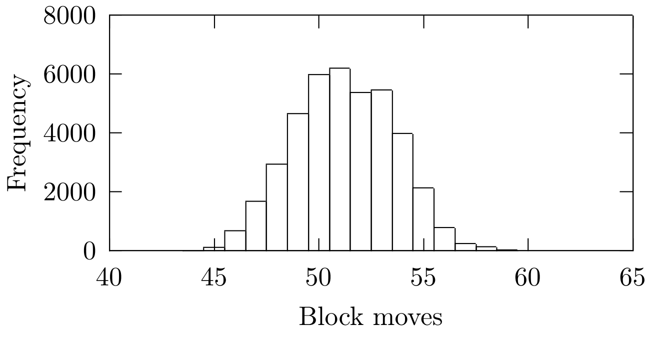


Figure 9: Histogram of the number of block crossings used by the greedy algorithm for all $k!$ different start permutations, on a single random instance with $n = 100$ and $k = 8$. The best greedy solution uses 45 block crossings; the average is 51.2, with standard deviation 2.39.

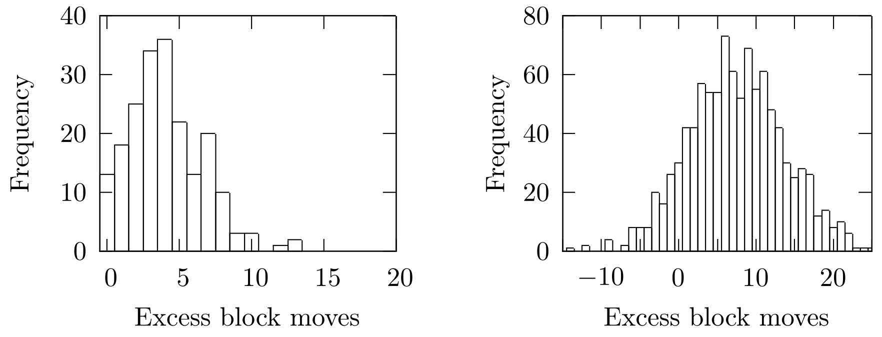


Figure 10: Left: histogram of HeuristicGreedy minus BestGreedy, 200 instances with with $k = 7$ and $n = 100$. Right: histogram of RandomGreedy minus HeuristicGreedy, 1000 instances with $k = 30$ and $n = 200$.

permutation and proceeds greedily. The histogram in Fig. 9 shows the number of block crossings used by the greedy algorithm for each of the $k!$ possible start permutations, for a single fixed instance. This bell curve is typical for instances generated by our random model, and is shown for illustration. We see, qualitatively, that there indeed exist "rare" start permutations that do noticeably better than almost all others (here with 45 block crossings). Indeed, for the reported instance, a random start permutation does 7.2 block crossings worse in expectation than the best possible start permutation, while the number of block crossings for different start permutations for this instance has standard deviation of 2.4.

We call the best possible result of the greedy algorithm over all start permutations BestGreedy, which we calculate by brute force. This is deterministic

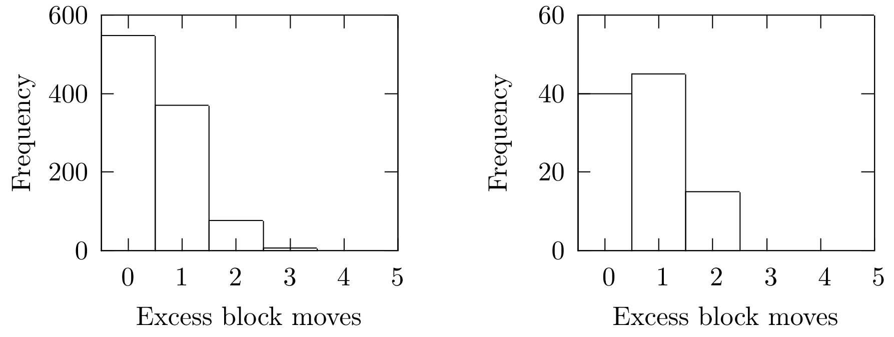


Figure 11: Left: Heuristic-greedy minus OPT with $k = 5$ and $n = 12$, 1000 runs. Right: the same, except $k = 6$ and 100 runs.

given the instance. Let RandomGreedy be the result of the greedy algorithm starting with a permutation chosen uniformly at random, and let Heuristic-Greedy be the result of starting with the heuristic permutation that we have described above. The histogram in Fig. 10 (left) shows how many more block crossings HeuristicGreedy uses than BestGreedy on random instances. This distribution is heaviest near zero (meaning the heuristic is often close to optimal), but there are instances where the difference is large. On average the heuristic is 4.1 block crossings worse than best possible start permutation on these instances, which is an improvement over random start permutations; see Fig. 10, right. Note that we do not know how to compute BestGreedy efficiently, but this experiment demonstrates that there is room for improvement within the greedy algorithm by picking a good start permutation.

Lastly, we compare the greedy algorithm to the optimum, which (because of runtime) we only do for small k and n. On 1000 random instances with $k = 5$ and $n = 12$, HeuristicGreedy gave an optimal solution 56% of the time. It needed sometimes one (38%), two (5%), or three (1%) block crossings more, but never more than that. With $k = 6$, we get similar numbers; see Fig. 11. This is a promising behavior, but clearly cannot be extrapolated verbatim to larger instances.

Based on these experiments, we recommend HeuristicGreedy as an efficient, reasonable heuristic for the 2-SBCM problem when it arises. We have not run experiments to visualize real storylines (movies, books, et cetera) since the restriction to cardinality-2 meetings is too severe. The genealogical visualizations of Kim et al. [8] do fit this restriction, but require further visualization work on top of solving the SBCM instance, which we have not performed. Instead, we have focused these experiments on some combinatorial properties of random instances.

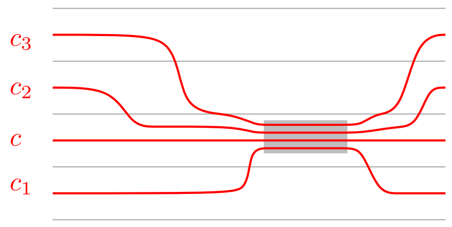


Figure 12: Meeting $\{c, c_1, c_2, c_3\}$

6 Approximation Algorithm

We now develop a constant-factor approximation algorithm for d-SBCM where d is a constant. We initially assume that each group meeting occurs exactly once, but later show how to extend our results to the setting where the same group can meet a bounded number of times.

Overview. Our approximation algorithm has the following three main steps.

1. Reduce the input group hypergraph $\mathcal{H} = (C, \Gamma)$ to an *interval hypergraph* $\mathcal{H}_\mathrm{f} = (C,\ \Gamma \setminus \Gamma_\mathrm{p})$ by deleting a subset $\Gamma_\mathrm{p} \subseteq \Gamma$ of the edges of $\mathcal{H}$.

2. Choose a permutation π_0 of the characters that supports all groups of this interval hypergraph $\mathcal{H}_\mathrm{f}$. Thus, π_0 is the order of characters at the beginning of the timeline.

3. Incrementally create support for each deleted meeting of Γ_p in order of increasing time, as follows. We arbitrarily fix one character c of the meeting and move the other characters of the meeting next to it. After the meeting we move the characters back to their original positions; see Fig. 12.

Step 2 is straightforward: Section 2 shows how to find a permutation supporting all the groups for an interval hypergraph. The main technical parts of the algorithm are Step 1 and an analysis to charge at most a constant number of block crossings in Step 3 to a block crossing in the optimal visualization. Step 1 requires solving a hypergraph problem; this is technically the most challenging part, and consumes the entire Section 7.

Bounds and Analysis. We call the edges in Γ_p *paid* edges, and the edges in $\Gamma_\mathrm{f} = \Gamma \setminus \Gamma_\mathrm{p}$ *free* edges. Intuitively, a free edge can be realized without a block crossing because $\mathcal{H}_\mathrm{f}$ is an interval hypergraph, while each edge $e \in \Gamma_\mathrm{p}$ must be charged to block crossings of the optimal drawing. We initialize the drawing by placing the characters in the vertical order π_0, which supports all the groups in Γ_f. Now we consider the paid edges in left-to-right order. Suppose that the next meeting involves a group $g \in \Gamma_\mathrm{p}$. We have $|g| \leq d$ and fix one character of g. To bring the remaining characters of g in this character's vicinity, we need at most $(d-1)$ block crossings, one per line. When the meeting is over, we again use up to $(d-1)$ block crossings to revert the lines back to their original position prescribed by π_0; see Fig. 12.

We do this for each paid hyperedge, giving rise to at most $2(d-1)|\Gamma_\mathrm{p}|$ block crossings. We now prove that this bound is within a constant factor of optimal. We first establish a lower bound on the optimal number of block crossings assuming a fixed start permutation.

Lemma 4 *Let π be a permutation of the characters, let Γ_f be the set of groups supported by π, and let $\Gamma_\mathrm{p} = \Gamma \setminus \Gamma_\mathrm{f}$. Any storyline visualization with start permutation π needs at least $2|\Gamma_\mathrm{p}|/(3d^2)$ block crossings.*

Proof: Let $g \in \Gamma_\mathrm{p}$. Since g is not supported by π, the optimal drawing does not contain the characters of g as a contiguous block initially. However, in order to support this meeting, these characters must eventually become contiguous before the meeting starts. The order changes only through (block) crossings; we bound the number of groups that can become supported after each block crossing.

After a block crossing, at most three pairs of lines that were not neighbors before can become neighbors in the permutation: after the blocks $C_1, C_2 \subseteq C$ cross, there is one position in the permutation where a line of C_1 is next to a line of C_2, and two positions with a line of C_1 (C_2, respectively) and a line of $C \setminus (C_1 \cup C_2)$. Any group that was not supported, but is supported after the block crossing, must contain one of these pairs. We can describe each such group in the new permutation by specifying the new pair and the numbers d_1 and d_2 of characters of the group above and below the new pair in the permutation. Since the group size is at most d, we have $d_1 + d_2 \leq d$. For any $d_1 \geq 1$, there are $d - d_1$ possible choices for d_2. Together with $d_1, d_2 \geq 1$ (since the new pair is included), we get at most $\sum_{d_1=1}^{d-1}(d-d_1) = \sum_{d_1=1}^{d-1} d_1 = d(d-1)/2 \leq d^2/2$ possible groups for each new pair. Thus, the total number of newly supported groups after a block crossing is at most $3d^2/2$, which shows that the optimal number of block crossings is at least $2|\Gamma_\mathrm{p}|/(3d^2)$, completing the proof. □

We now bound the loss of optimality caused by not knowing the initial permutation used by the optimal solution. The key idea here is to use a constant-factor approximation for the problem Interval Hypergraph Edge Deletion: delete the minimum number of hyperedges from $\mathcal{H}$ so that $\mathcal{H}$ becomes an interval hypergraph. We prove the following theorem in Section 7.

Theorem 8 *We can find a $(d+1)$-approximation for* Interval Hypergraph Edge Deletion *on group hypergraphs with m hyperedges of rank d in $O(m^2)$ time.*

With the help of Theorem 8, we can show the main result of this paper.

Theorem 9 *d-SBCM admits a $(3d^2(d^2-1))$-approximation algorithm that runs in $O(kn)$ time.*

Proof: Let Γ_{p} be the set of paid edges in our algorithm, and let Γ_{OPT} be the set of paid edges in the optimal solution, that is, the set of edges that are not supported by the initial permutation of the optimal solution. By Theorem 8, we have $|\Gamma_{\mathrm{p}}| \leq (d+1)|\Gamma_{\mathrm{OPT}}|$, since even Γ_{OPT} cannot be smaller than an optimum solution for the interval hypergraph edge deletion problem.

Let ALG and OPT be the numbers of block crossings for our algorithm and the optimal solution, respectively. Our algorithm ensures that $\mathrm{ALG} \leq 2(d-1)|\Gamma_{\mathrm{p}}| \leq 2(d-1)(d+1)|\Gamma_{\mathrm{OPT}}|$. On the other hand, by Lemma 4, we have $\mathrm{OPT} \geq 2|\Gamma_{\mathrm{OPT}}|/(3d^2)$, which gives $|\Gamma_{\mathrm{OPT}}| \leq 3d^2/2 \cdot \mathrm{OPT}$. Combining the previous inequalities, we get $\mathrm{ALG} \leq 3d^2(d^2-1) \cdot \mathrm{OPT}$ as desired.

Now we analyze the time complexity. We have to consider the permutation (of length k) of characters before and after each of the n meetings, as well as after each of the $O(n)$ block crossings. This results in $O(kn)$ time for the last part of the algorithm, but this is dominated by the time ($O(n^2)$) needed for finding Γ_{p} and for determining the start permutation.

We can improve the running time to $O(kn)$ by a slight modification: using the approximation algorithm for Interval Hypergraph Edge Deletion is only necessary for sparse instances. If $\mathcal{H}$ has sufficiently many edges, any start permutation will yield a good approximation. Since no meeting involves more than d characters, no start permutation can support more than dk meetings. If $n \geq 2dk$, then even the optimal solution must therefore remove at least half of the edges. Hence, taking an arbitrary start permutation yields an approximation factor of at most $2 < d + 1$.

We now change the algorithm to use an arbitrary start permutation if $n \geq 2dk$ and only use the approximation for Interval Hypergraph Edge Deletion otherwise. In particular, we use the approximation only if there are $n \in O(k)$ edges. Hence, for sparse instances we have $O(n^2) \subseteq O(kn)$, and for dense instances, we skip the costly $O(n^2)$-time step mentioned above. This yields the desired running time of $O(kn)$, which is worst-case optimal since the output complexity is of the same order. □

Remark. We assumed that each group meets only once, but we can extend the result if each group can meet α times, for constant α. Our algorithm then yields a $(\alpha \cdot 3d^2(d^2-1))$-factor approximation; each repetition of a meeting may trigger a constant number of block crossings not present in the optimal solution.

Improvements for 2-SBCM. By using specific structures for 2-character meetings we can improve approximation factor and runtime; note that the general algorithm yields a 36-approximation.

For 2-character meetings the group hypergraph is a graph; an interval hypergraph in this setting is a collection of vertex-disjoint paths. Our algorithm for Interval Hypergraph Edge Deletion for $d = 2$ yields a 3-approximation. We develop a better approximation using the following observation. Consider a character c in the collection of paths supported in the beginning of some solution. If c has two neighbors c_1 and c_2 in its path, but c's first meeting is with a character $c_3 \notin \{c_1, c_2\}$, then at the beginning of that meeting c can only be neighbor to one of the two, say, to c_1, even in an optimal solution; the meeting with c_2 then must later be reconstructed by block crossings. Hence, the effective set of meetings supported in the beginning is in fact a collection of paths with the additional restriction that each character is adjacent to at most one character except for the one he meets first. Without changing the rest of the analysis, we can approximate this new problem for finding the start permutation with an approximation factor of 2.

We first consider, for each vertex c, all edges incident to c except for the one describing c's first meeting. If there are $\ell \geq 2$ such edges, we know that even the optimal solution can support at most one of them and, hence, has to remove $\ell - 1$ of them. We remove all ℓ of them, which preserves the intended approximation factor of 2 as $\ell/(\ell - 1) \leq 2$. Eventually, all vertices have degree 2 or less and the connected components are paths and cycles. For each cycle, we remove one arbitrary edge, so that we end up with a collection of paths. This second step does not change the approximation factor since the optimal solution has to remove at least one edge per cycle as well. This part of the algorithm easily runs in linear time, which yields a total running time of $O(kn)$.

Theorem 10 *We can find a 24-approximate solution for 2-SBCM without repetitions in $O(kn)$ time.*

For 2-SBCM, too, the approximation algorithm generalizes to the case where each group of characters can meet up to α times for a constant α; the approximation factor then is 12α.

7 Interval Hypergraph Edge Deletion

We now describe the main missing piece from our approximation algorithm: how to approximate the minimum number of edges whose deletion reduces a hypergraph to an interval hypergraph, i.e., how to solve the following problem.

Problem 2 (Interval Hypergraph Edge Deletion) *Given a hypergraph $\mathcal{H} = (V, E)$ find a smallest set $E_\mathrm{p} \subseteq E$ such that $\mathcal{H}_\mathrm{f} = (V, E \setminus E_\mathrm{p})$ is an interval hypergraph.*

For standard graphs, both vertex and edge deletion problems for different graph properties have been considered in the past. Recall that a *hereditary* property is a property that is preserved while deleting any vertex from a given graph satisfying that property. The vertex deletion problem for a property asks for theminimum number of vertices whose deletion results in an induced subgraph satisfying that property. By a famous result of Yannakakis [19], all vertex deletion problems for hereditary properties are NP-hard. This implies that interval hypergraph vertex deletion is NP-hard.

A seemingly related class graph is that of standard *interval graphs*, the intersection graphs of intervals on the real line. For this graph class, edge deletion

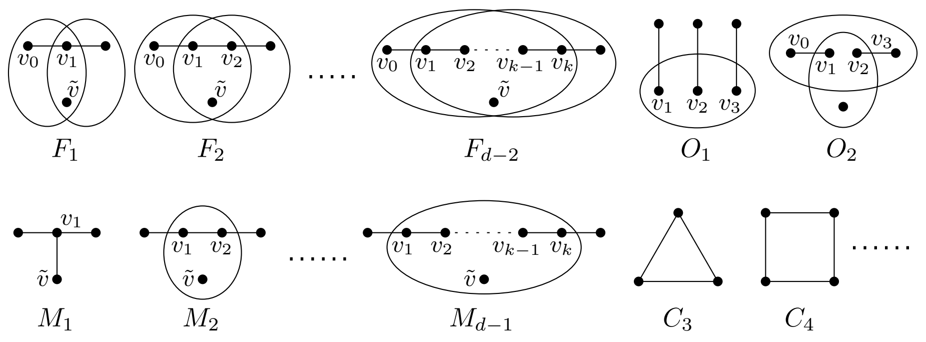


Figure 13: List of patterns that are forbidden in interval hypergraphs: none of these can occur as a subhypergraph of an interval hypergraph. Line segments represent hyperedges of size 2, whereas ellipses represent larger hyperedges. Note that each hyperedge may contain further vertices of the full graph that are not shown since they are not part of the pattern. The names of the vertices are used in the runtime analysis and in Algorithms 1–4.

is known to be NP-hard [6]. Note, however, that this result for interval graphs does not imply anything about interval hypergraphs since the two properties are entirely different. Still, it is easy to see that interval hypergraph edge deletion is NP-hard as follows.

A graph is an interval hypergraph if and only if all its connected components are paths. In particular, an n-vertex graph with $n-1$ edges is an interval hypergraph if and only if it is a path (note that this is not true for interval graphs!). Since a graph contains a Hamiltonian path if and only if one can remove all but $n-1$ edges so that a single path remains, our problem is hard even for graphs.

Theorem 11 Interval Hypergraph Edge Deletion *is NP-hard.*

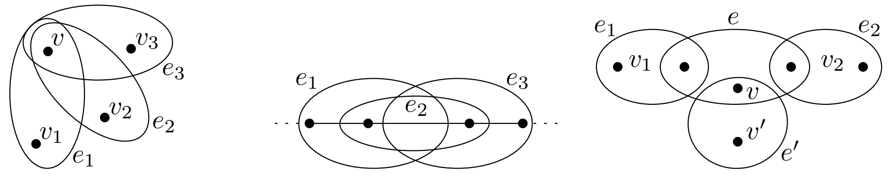
We now present a $(d+1)$-approximation algorithm for rank-d hypergraphs, that is, where each hyperedge has at most d vertices.

For our algorithm, we use the following known characterization: a hypergraph is an interval hypergraph if and only if it contains none of the hypergraphs shown in Fig. 13 as a subhypergraph [16, 12]. Cycles are the only arbitrarily large forbidden subhypergraphs in our setting: the families F_k and M_k are finite since we consider only bounded-rank hypergraphs. (F_{d-2} and M_{d-1} are the largest relevant members of these families.) From now on, let

$$\mathcal{F} = \{O_1, O_2, F_1, \ldots, F_{d-2}, M_1, \ldots, M_{d-1}, C_3, \ldots, C_{d+1}\}.$$

We call the elements of $\mathcal{F}$ *patterns*, and we say that a hypergraph is *$\mathcal{F}$-free* if it does not contain any pattern as a subhypergraph.

There are various definitions of cycles in hypergraphs; we use the following (fairly common) option. A cycle C in a hypergraph is a sequence of $\ell \geq 3$



(a) three edges form M_1 (b) three edges consecutive on C (c) four edges form O_1

Figure 14: Illustrations for the proof of Lemma 5.

hyperedges $e_1, \ldots, e_\ell$ with the property that there are vertices $v_1, \ldots, v_\ell$ such that, for $2 \leq i \leq \ell$, (i) $v_i \in e_{i-1} \cap e_i$ (and $v_1 \in e_\ell \cap e_1$), and (ii) edge e_i contains no vertex of $v_1, \ldots, v_\ell$ except for v_i and v_{i+1} (and e_1 contains only v_ℓ and v_1). As usual, the *length* of the cycle C is the number ℓ of hyperedges. We write $V(C) = \bigcup_{e \in C} e$ for the vertex set of C.

Our approximation algorithm consists of two steps. First, we search for patterns from $\mathcal{F}$, and remove all edges involved in these hypergraphs. In the second step, we break remaining (longer) cycles by removing some more hyperedges based on the structure of connected components. We analyze the approximation ratio as follows. Any pattern in $\mathcal{F}$ consists of at most $d + 1$ hyperedges. A given optimal solution must remove at least one of these hyperedges; we remove all of them instead, which yields a ratio of at most $d + 1$. The second step will not negatively affect this approximation ratio.

Intuitively, allowing long cycles while forbidding patterns of $\mathcal{F}$ results in a generalization of interval hypergraphs where the vertices may be placed on a cycle rather than a vertical line. This is not exactly true, but we will see that, after the first step, the connected components have a structure similar to this, which will help us find a small set of edges whose removal destroys all remaining long cycles.

The following lemma shows that, in an $\mathcal{F}$-free hypergraph, any vertex is contained in at most three hyperedges of a cycle, where the case of three hyperedges with a common vertex occurs only if a hyperedge is contained in the union of its two neighbors in the cycle.

Lemma 5 *Let $\mathcal{H} = (V, E)$ be an $\mathcal{F}$-free hypergraph and let C be a cycle in $\mathcal{H}$. Then two edges of C can have a common vertex only if they are either consecutive in C or if they share a common neighbor in C.*

Proof: According to the definition of a cycle, no edge of C can fully contain another edge of C. Let $e_1, e_2, e_3 \in C$ be three edges of C, and assume that there is a vertex v with $v \in e_1 \cap e_2 \cap e_3$. If there are "private" vertices $v_1 \in e_1 \setminus (e_2 \cup e_3)$, $v_2 \in e_2 \setminus (e_1 \cup e_3)$, and $v_3 \in e_3 \setminus (e_1 \cup e_2)$, the three hyperedges form pattern M_1 (with v_1, v_2, v_3, and v serving as vertices); see Fig. 14(a).

On the other hand, if one of the three edges does not have a private vertex, say e_2, we have $e_2 \subseteq e_1 \cup e_3$, and one easily checks that this can only be the case if the three edges are consecutive on the cycle; see Fig. 14(b).

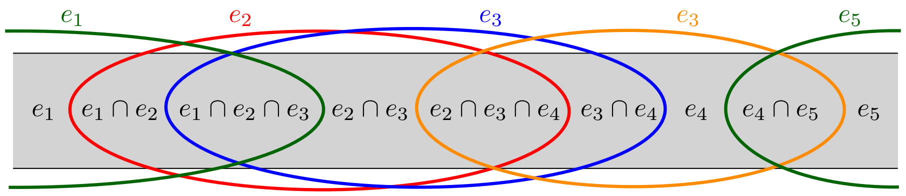


Figure 15: Cycle-sets and their relative order. The gray band represents all vertices of the cycle while the ellipses represent hyperedges of the cycle. Note that hyperedges not included in an intersection are meant to be excluded; e.g., $e_1 \cap e_2$ stands for $(e_1 \cap e_2) \setminus (e_3 \cup e_4 \cup e_5)$.

Now, assume that there are two edges that are neither consecutive nor have a common neighboring hyperedge in C, but share a vertex $v \in e \cap e'$. Due to the observations in the first part of the lemma, v cannot be contained in any of the neighbors of e and e' in C. Let e_1 and e_2 be the neighbors of e in C. If either of the two intersects with e', we find pattern C_3, a contradiction.

Hence, each of e', e_1, and e_2 has a private vertex (v', v_1, and v_2 in Fig. 14(c), respectively). Together with vertices in the intersections with e (that is, vertices in $e \cap e'$, $e \cap e_1$, and $e \cap e_2$), the private vertices form pattern O_1 with respect to edge e. This again contradicts $\mathcal{H}$ being $\mathcal{F}$-free. □

Let e_1, e_2, and e_3 be three consecutive edges of a cycle C. If all three edges are present in an interval representation for some of the edges of $\mathcal{H}$, we know that we will encounter vertices in the following order: first vertices that are contained only in e_1, then vertices that are in $(e_1 \cap e_2) \setminus e_3$, then vertices that are in $e_1 \cap e_2 \cap e_3$, followed by vertices of $(e_2 \cap e_3) \setminus e_1$, and vertices of $e_3 \setminus (e_1 \cup e_2)$. Some of the sets (except for the pairwise intersections) may be empty. We do not know the relative order of vertices within one set, but we know the relative order of any pair of vertices of different sets; see Fig. 15. By generalizing this to the whole cycle, we get a cyclic order—describing the local order in a possible interval representation—of sets defined by containment in one, two, or three hyperedges. We call these sets *cycle-sets* and their cyclic order the *cycle-order* of C.

The following lemma effectively shows that, given two cycles, their vertex sets are either identical or disjoint.

Lemma 6 *Let $\mathcal{H} = (V, E)$ be an $\mathcal{F}$-free hypergraph, and let C be a cycle in $\mathcal{H}$. Then, for any hyperedge $e \in E$, it holds that either $e \subseteq V(C)$ or $e \subseteq V \setminus V(C)$.*

Proof: Assume to the contrary that there is a hyperedge e with $e \not\subseteq V(C)$ but $e \cap V(C) \neq \emptyset$. If e contains at least one vertex that lies in the intersection of two edges of C, then we find pattern M_k (for some $k \in \{1, \ldots, d-1\}$) as follows. Assume that $v' \in e \cap e_1 \cap e_2$ with edges e_1 and e_2 consecutive on C. From v' on, we follow C in both directions as long as we find vertices in the intersection of consecutive cycle edges that also belong to e. This process must stop eventually since e can contain at most $d-1$ vertices of cycle edges whereas C has length at

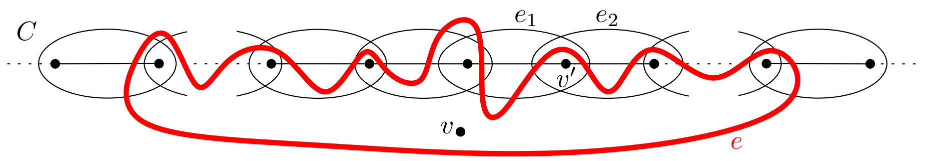


Figure 16: If edge e contains vertices of cycle C and vertices beyond C, pattern M_k occurs.

least $d + 2$ (due to the fact that $\mathcal{H}$ is $\mathcal{F}$-free). Together with two vertices of the next intersections of cycle edges (that are not in e), we have found a path that, together with e and v, forms pattern M_k; see Fig. 16.

Now, we know that e cannot contain a vertex that lies in two cycle edges, but there could still be an edge e' of C with a vertex $v' \in e \cap e'$. This, however, would represent pattern O_1 with respect to e, e', and the two neighbors of e' in C (see Fig. 14(c)) and would again contradict $\mathcal{H}$ being $\mathcal{F}$-free. □

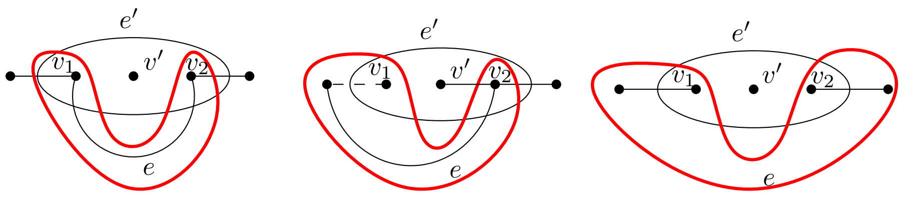
Due to Lemma 6 two cycles have either identical or disjoint vertex sets. Therefore any connected component of an $\mathcal{F}$-free hypergraph is either acyclic or forms the ground set for a set of cycles. We now consider the structure of cycles on such a connected component in order to break all remaining cycles optimally.

If two cycles share their vertex sets, we can analyze how an edge of one cycle relates to the structure—the cycle sets and their order—of the other cycle. Recall that we know the relative order of the cycle-sets, but not the internal order of vertices within the same cycle-set. Another edge can contain a cycle-set completely, can be disjoint from it, or can contain only some of its vertices. We call a consecutive sequence of cycle-sets contained in edge e—potentially starting and ending with cycle-sets partially contained in e—an *interval* of e on C. The following lemma shows that every edge forms only a single interval on a given cycle.

Lemma 7 *Let $\mathcal{H} = (V, E)$ be an $\mathcal{F}$-free hypergraph, let C be a cycle in $\mathcal{H}$, and let $e \in E$ be a hyperedge with $e \subseteq V(C)$. If e intersects two cycle-sets of C, then e must fully contain the vertices of all cycle-sets of C in between the two cycle-sets, in one of the two directions along cycle C.*

Proof: Assume that the claim is not true, that is, e consists of a collection of at least two intervals of (partially) contained cycle-sets, where any two such intervals are separated by a vertex that lies in a cycle-set but not in e. We distinguish cases similar to the proof of the previous lemma.

1. First, assume that one of the intervals contains a vertex that is part of two consecutive edges of the cycle. We follow the cycle in both directions from that vertex, as long as we find a vertex of e in the intersection of the current edge with the next edge along the cycle. We call these vertices *internal vertices*. Since e has at most d vertices but C has length at least $d + 2$, this process will eventually stop, thus forming a path of at most d



(a) M_2 as a pattern. (b) M_1 as a pattern. (c) O_2 as a pattern.

Figure 17: Vertex $v' \notin e$ in a gap between two intervals of e.

vertices in e and at most $d-1$ edges in C. At both ends of this path, we add a vertex from the two incident edges in C, which are not in e (due to the maximality of the path we constructed). Thus, we get a path Π of length at least two and at most $d+1$. The intersection of every pair of consecutive edges of Π contains at least one vertex of e. The first and the last edge of Π both contain at least one vertex that is not in e and private to them; they do not contain vertices of e that are private to them among the edges of Π. (Note that Π cannot be a cycle since this would be pattern C_k for some $k \in \{3, \dots, d+1\}$.)

(a) Now, assume that e has a private vertex $v \in e \setminus V(\Pi)$. Then, this yields pattern M_k for some $k \in \{1, \dots, d-1\}$; see Fig. 16.

(b) Hence, e does not have such a private vertex. Then, if the path Π has length 2, the union of its two edges e_1, e_2 contains e. It is easy to check that our assumption fails in this case: The only violation to the interval property would be that there is a vertex $v' \in e_1 \cap e_2 \cap e$ and vertices $v_1 \in e_1 \cap e \setminus e_2$ and $v_2 \in e_2 \cap e \setminus e_1$. However, in this case the edges form the forbidden pattern C_3 with v', v_1, v_2 as defining vertices.

If Π has length at least 3, then there are at least two internal vertices, all of which belong to e by construction. If, for each pair of consecutive internal vertices, all cycle-sets separating the two are fully contained in e, then e forms a single interval. Hence, there must be a vertex $v' \notin e$ in a cycle-set separating two consecutive internal vertices v_1 and v_2 (connected by an edge e') of Π; see Fig. 17(a).

i. If none of the neighbors of v_1 and v_2 along the cycle lies in e, we have found pattern M_2 as in Fig. 17(a).

ii. If the neighbor of only one of them, say, v_1 is in e but the neighbor of v_2 isn't, then by disregarding v_1 we find pattern M_1 centered at v_2; see Fig. 17(b).

iii. On the other hand, if both neighbors lie in e, then we have pattern O_2; see Fig. 17(c).

2. If e does not contain any vertex that lies in the intersection of two consecutive cycle edges, then we take vertices $v \in e$ and $v' \in e$ from two different

intervals; $v \in e_1$ and $v' \notin e_1$ for a cycle edge e_1. Let e_0 and e_2 be the neighbors of e_1 in C. We have $v' \notin e_0 \cup e_2$ since otherwise there would be a triangle, that is, pattern C_3. But then e_0, e_1, e_2, and e (via v') form pattern O_1. □

Since e forms only a single interval of cycle-sets, we know that by opening the cycle at a single position within a cycle-set not contained in e, $C + e$ forms an interval hypergraph. Edge e adds further information on the relative order within some cycle-sets: if only some of the vertices of a cycle-set are contained in e and also vertices of the next cycle-set in one direction, we know that the vertices of e in the first cycle-set should be next to the second cycle-set.

We use this to refine the cycle-sets to a cyclic order of *cells*, the *cell order*. A cell is a set of vertices that should be contiguous in the cyclic order prescribed by hyperedges. Initially, the cells are the cycle-sets. Then, in each step we refine the cell-order by inserting an edge containing vertices of more than one cell, possibly splitting two cells into two subcells each. If after refining the cell order, it is still true that each remaining edge forms a single interval, then this results in a final refined cell order, where each remaining edge of the connected component must be fully contained in one of the cells.

Lemma 8 *For any time in the refinement process and any connected component K of overlapping hyperedges that do not form a cycle on the cell order at that time, there is another edge overlapping with the connected component.*

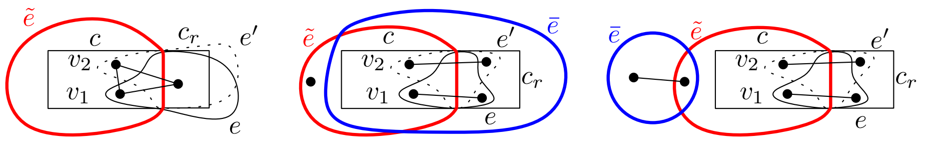
Proof: Assume there were a connected component K for which the claim is not true, that is, there is no currently inserted hyperedge overlapping with K. Then, K must be fully contained in a single cycle-set of C; especially, no edge of C participates in K. No, consider the edge e of K that was inserted first. Since e is no edge of C, at the point of insertion of e, there must have existed an edge overlapping with e, otherwise e would not have been inserted. Either e overlaps with K, or K fully contains e. In the latter case, we extend K with e (as the new oldest edge) and continue the search. This process must evenutally stop since there is only a finite number of inserted edges. The last edge of the process must therefore overlap with K. □

The following lemma shows that, during the process of refinements, the interval property is indeed preserved as an invariant.

Lemma 9 *Let $\mathcal{H} = (V, E)$ be an $\mathcal{F}$-free hypergraph, and let C be a cycle in $\mathcal{H}$. We initialize the cell order with the cycle-sets of C. Then, we iteratively refine the structure by considering an edge that contains vertices of at least two different cells (our* insertion criterion*). Then, the following* interval property *holds at any refinement step for any hyperedge $e \in E$ with $e \subseteq V(C)$:*

If e intersects two cells, then e must fully contain the vertices of all cells lying in between the two cells in one of the two directions along the cyclic order.

Proof: We show the interval property by induction on the number of insertions. Due to Lemma 7 the property holds in the beginning. Now, assume that the



(a) C_3 in case 1.1.1. (b) O_2 in case 1.1.2.1. (c) O_1 in case 1.1.2.2.

Figure 18: Edges e and e' overlapping in cell c from the same direction (I).

property holds for the cell order after inserting a set of edges. We show that after refining the cells by considering another edge e', the property still holds.

Assume to the contrary that for the refined cells the interval property does not hold for some edge e. Since the property did hold for the cells of the previous step, the only problem can occur in a cell c of the previous step that is only partially contained by both e and e'. Let c_r and c_l be the cells right and left of c, respectively. Without loss of generality, we can assume that e' also contains elements of c_r. Let $c_1 = c \setminus e'$ and $c_2 = c \cap e'$ be the (nonempty) cells that result from splitting c. There are two basic cases in which the interval property could be violated for e.

1. First, if e contains also elements of c_r, then we have a violation only if there are vertices $v_1 \in c_1 \cap e$ and $v_2 \in c_2 \setminus e$. We distinguish cases based on the right boundary of cell c, which—from left to right—can either be closing or opening at least one hyperedge $\tilde{e}$.

   1.1. First assume that the right boundary of c is closing $\tilde{e}$, that is, $\tilde{e}$ contains c and c_l, but is disjoint from c_r.

   1.1.1. If there is a common vertex of e and e' in c_r, then we find pattern C_3 with respect to e, e', and $\tilde{e}$; see Fig. 18(a).

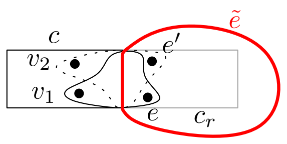
   1.1.2. Otherwise, there are vertices in c_r that are unique for e and e', respectively.
   Since we never inserted an edge completely contained in cells (due to our insertion criterion), this must also have hold for $\tilde{e}$. Therefore, there must be an edge $\bar{e}$ (apart from e and e') containing some (but not all) cells of $\tilde{e}$ and cells either to the left or to the right of $\tilde{e}$.

   1.1.2.1. If $\bar{e}$ contains cells to the right, then in particular it must contain cells c and c_r. Together with a vertex in $\tilde{e} \setminus \bar{e}$, this is pattern O_2; see Fig. 18(b).

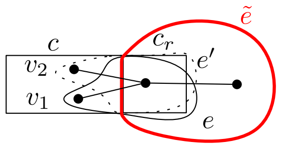
   1.1.2.2. On the other hand, if $\bar{e}$ contains cells of $\tilde{e}$ and cells to the left of $\tilde{e}$, then we have pattern O_1, using a vertex in $\tilde{e} \cap \bar{e}$ (not in c) and a vertex in $\bar{e} \setminus \tilde{e}$; see Fig. 18(c).

   1.2. Now, assume that $\tilde{e}$ is opening on the right boundary of c, that is, $\tilde{e}$ contains no vertex of c, but all of c_r.

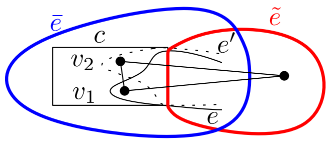
   1.2.1. If c_r contains no common element of e and e', the situation is symmetric to the one we had before by swapping the roles of c and c_r; see Fig. 19(a).



(a) Symmetric situation in case 1.2.1.



(b) M_1 in case 1.2.2.1.



(c) C_3 in case 1.2.2.2.1.1.

Figure 19: Edges e and e' overlapping in cell c from the same direction (II).

1.2.2. Otherwise, there is an element of $e \cap e'$ in c_r.

1.2.2.1. If $\tilde{e}$ contains an element not in $e \cup e'$, then we have found pattern M_1; see Fig. 19(b).

1.2.2.2. If $\tilde{e}$ contains no such vertex then, due to the insertion criterion, we know that there must be at least one previously inserted hyperedge $\bar{e}$ overlapping $\tilde{e}$.

1.2.2.2.1. Assume that $\bar{e}$ is overlapping $\tilde{e}$ from the left.

1.2.2.2.1.1. If there is a vertex of $e \cap e'$ in $\tilde{e} \setminus \bar{e}$, we have found pattern C_3; see Fig. 19(c).

1.2.2.2.1.2. Otherwise, there must be a vertex in $\tilde{e} \setminus \bar{e}$ that is contained in only one of e and e', say, in e, and we find pattern F_1; see Fig. 20(a).

1.2.2.2.2. Now, assume that $\bar{e}$ is overlapping with $\tilde{e}$ coming from the right.

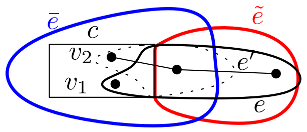
1.2.2.2.2.1. If $\tilde{e} \cap \bar{e}$ contains a vertex of only one of the edges, say e, we argue as follows. If there is a vertex not in e (and not in e'), we have found pattern M_2; see Fig. 20(b); if $\tilde{e} \cap \bar{e}$ contains a vertex of $e \cap e'$, we find pattern M_1 instead.

1.2.2.2.2.2. Otherwise,—that is, if $\tilde{e} \cap \bar{e} \subseteq e \cap e'$—we can continue to explore more edges since $\tilde{e} \cup \bar{e}$ must overlap with at least one more edge. As long as there is a hyperedge overlapping with the hyperedges $\tilde{e} \cup \bar{e} \cup \ldots$ to the right, we choose the one ending rightmost, thus forming a path of hyperedges that is extending to the right.

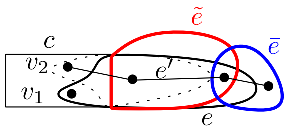
1.2.2.2.2.2.2.1. If this process eventually finds a vertex that is neither in e nor in e', we find pattern M_k; see Fig. 20(c).

1.2.2.2.2.2.2.2. If we do not reach a vertex not in e or e' with the path because there are no more edges overlapping from the right, we know that there must be an edge $\bar{e}'$ overlapping the whole path from the left; otherwise, the edges of the path would not have been inserted before; see Lemma 8.

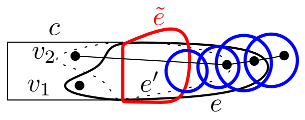
1.2.2.2.2.2.2.2.1. Now, if there is a vertex of $e \cap e'$ not contained in $\bar{e}'$, we have found pattern C_3; see Fig. 21(a).



(a) F_1 in case 1.2.2.2.1.2.

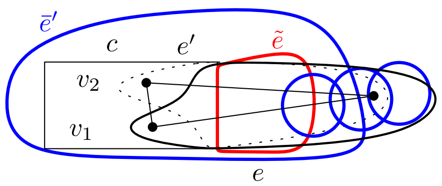


(b) M_2 in case 1.2.2.2.2.1.

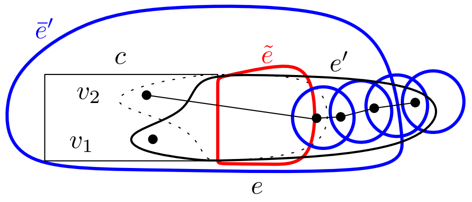


(c) M_k in case 1.2.2.2.2.2.1.

Figure 20: Edges e and e' overlapping in cell c from the same direction (III).



(a) C_3 in case 1.2.2.2.2.2.2.1.



(b) F_k in case 1.2.2.2.2.2.2.2.

Figure 21: Edges e and e' overlapping in cell c from the same direction (IV).

1.2.2.2.2.2.2.2. Otherwise, the part of the path outside of $\bar{e}'$ contains a vertex that is only in one of the hyperedges, say, in e. Then, the pattern that we find is F_k with $\bar{e}'$ and e as the big hyperedges; see Fig. 21(b).

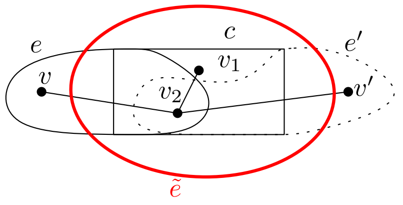
2. Now, we consider the second case in which we get a contradiction to the interval property after inserting e': again, let e' split cell c into $c_1 = c \setminus e'$ and $c_2 = c \cap e'$. Then, e contains vertices from cell c_l of c, at least one vertex v_2 of c_2, but there is also a vertex $v_1 \in c_1 \setminus e$, i.e., e does not completely contain c_1. We know that there must be at least one edge containing cell c.

   2.1. First, assume that such an edge $\tilde{e}$ as well as vertices $v \in e \setminus \tilde{e}$ and $v' \in e' \setminus \tilde{e}$ exist. Then, we find pattern M_1; see Fig. 22(a).

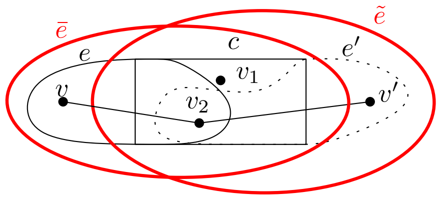
   2.2. If no such $\tilde{e}$ with vertices $v \in e \setminus \tilde{e}$ and $v' \in e' \setminus \tilde{e}$ exists, we know that any edge containing c must fully contain at least one of e and e' as a subset. On the other hand, we know that there must be at least one edge overlapping with e and one edge overlapping with e' (and by now, these two edges must be different; otherwise, we would be in the previous case).

      2.2.1. Assume that there are edges $\tilde{e}$ overlapping with e and fully containing e' and $\bar{e}$ overlapping with e' and fully containing e. Then we find pattern F_1; see Fig. 22(b).

      2.2.2. Now, assume that there is only a hyperedge $\tilde{e}$ overlapping with e and fully containing e'; among these edges let $\tilde{e}$ be the one starting rightmost and (among the ones starting rightmost) the shortest one, that is, the one with the smallest extension to the left. We know

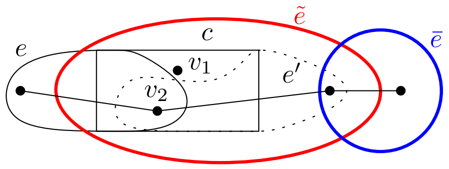


(a) M_1 in case 2.1.

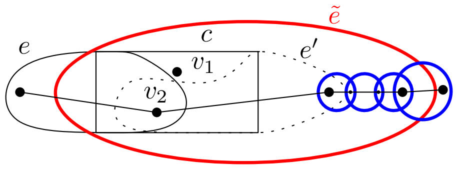


(b) F_1 in case 2.2.1.

Figure 22: Edges e and e' overlapping in cell c from different directions (I).



(a) M_2 in case 2.2.2.1.



(b) M_k in case 2.2.2.2.1.

Figure 23: Edges e and e' overlapping in cell c from different directions (II).

that there must be at least one edge overlapping with e', but no such edge can go to the left (and contain c), otherwise we would be in one of the previous cases. Let $\bar{e}$ be the hyperedge overlapping with e' and starting rightmost.

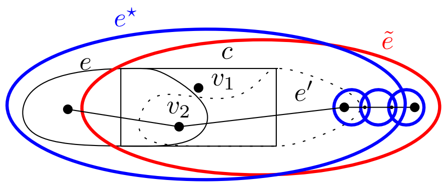
2.2.2.1. If $\bar{e}$ contains a vertex not contained in $\tilde{e}$, then we have found pattern M_2; see Fig. 23(a).

2.2.2.2. Otherwise, we continue searching the rightmost hyperedge that overlaps $\bar{e}$. This yields a path of hyperedges reaching to the right—similar to case 1.2.2.2.2.2..

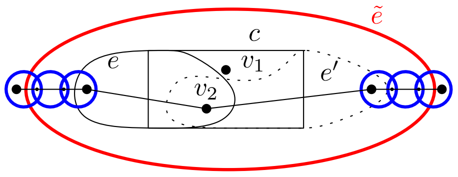
2.2.2.2.1. If eventually we reach a vertex not contained in $\tilde{e}$, then we have found pattern M_k; see Fig. 23(b).

2.2.2.2.2. On the other hand, if the path ends before reaching out of $\tilde{e}$, by considering the union of the path hyperedges ending with $\bar{e}$, we know that there must be a hyperedge $e^\star$ overlapping this union from the left. Due to the choice of $\tilde{e}$, $e^\star$ may or may not overlap with e, but it must contain an element of e that is not contained in $\tilde{e}$; otherwise, this would violate our choice of $\tilde{e}$ as starting rightmost. Therefore, we find pattern F_k; see Fig. 24(a).

2.2.3. In the remaining case, each edge containing cell c must fully contain both e and e'. Let $\tilde{e}$ be the edge containing c that is shortest and ends rightmost. Both for e and e' we know that there is at least one edge previously inserted that overlaps with them. Similarly to the argument before, we can start with the rightmost overlapping for e' and the leftmost for e and build paths of overlapping edges into

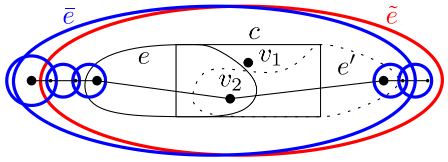


(a) F_k in case 2.2.2.2.2.

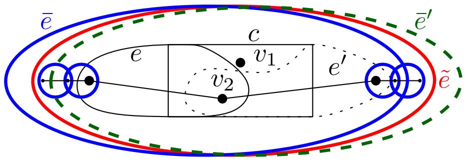


(b) M_k in case 2.2.3.1.

Figure 24: Edges e and e' overlapping in cell c from different directions (III).



(a) F_k in case 2.2.3.2.



(b) F_k in case 2.2.3.3.

Figure 25: Edges e and e' overlapping in cell c from different directions (IV).

these directions until we reach a vertex outside of $\tilde{e}$, or we find no further hyperedge to extend the path.

2.2.3.1. If both paths leave $\tilde{e}$, we find pattern M_k; see Fig. 24(b).

2.2.3.2. Now, assume that only the one for e reaches out of $\tilde{e}$, but the one for e' doesn't (the other case is symmetric). Since the hyperedges of the path for e' have been inserted, there must still be a hyperedge overlapping with them. The only remaining possibility is then that this hyperedge $\bar{e}$ extends to the left, contains c, and fully contains both e and e'. Due to the choice of $\tilde{e}$ as being the shortest hyperedge containing c, $\bar{e}$ must also contain at least one cell left of $\tilde{e}$. Hence, we find pattern F_k; see Fig. 25(a).

2.2.3.3. The remaining case is that neither path reaches out of $\tilde{e}$. Then, apart from $\bar{e}$, with the symmetric argument (again using that $\tilde{e}$ was chosen shortest) we find a hyperedge $\bar{e}'$ that overlaps with the path for e, fully contains e and e', and reaches out of $\tilde{e}$ to the right. By using $\bar{e}'$ in place of $\tilde{e}$, we again find pattern F_k; see Fig. 25(b). This completes the proof. □

The lemma shows that we can keep refining the cell structure by inserting edges that contain vertices of at least two different cells. We end up with a cyclic order of cells so that each edge of the connected component that we did not insert lies completely within a single cell. Several edges can lie within the same cell, sharing vertices, and forming a small hypergraph that imposes further restrictions on the relative order of vertices within the cell. However, the cell contains fewer than d vertices, so this small hypergraph cannot contain any long cycle. Since we removed all other patterns, it must be an interval hypergraph.

With this cell structure, we can show that the following strategy to make the connected component an interval hypergraph is optimal (see the following Lemmas 10, 11 and 12): for each pair of adjacent cells determine the number of edges containing both cells, select the pair minimizing that number, and remove all edges containing both. The cell order then yields an order of the connected component's vertices that supports all remaining edges. Since this last step of the algorithm is done optimally, the approximation ratio of $d+1$ is not affected: we never remove more than $d+1$ edges for at least one edge that the optimal solution removes.

Lemma 10 *If for each pair of two adjacent cells there is at least one hyperedge containing the vertices of both cells, we can find a cycle.*

Proof: We start at an arbitrary cell c. There must be a hyperedge e_1 containing both c and the next cell in clockwise order. We iteratively form a path by considering the rightmost cell explored so far and finding a hyperedge that contains that cell as well as the cell right of it. Since the number of cells is finite, we eventually reach the first cell. By dropping edges fully contained in other edges found, if necessary, we have a complete cycle. □

Lemma 11 *In any interval hypergraph that is obtained from the connected component, there is at least one pair of neighboring cells so that all edges containing both cells have been removed.*

Proof: The lemma is a direct corollary from the previous Lemma 10: if there were no such pair of adjacent cells, the condition of that lemma would hold, and there would be a cycle. □

Lemma 12 *Given a cyclic cell-order, let c and c' be a neighboring pair of cells in clockwise order. Removing all edges that contain both c and c' results in an interval hypergraph.*

Proof: We number the cells $c' = c_1, c_2, \ldots, c_k = c$ in clockwise order. Next, we place the vertices on a straight line so that vertices of each cell form an interval on the line and the cells appear as $c_1, \ldots, c_k$ from top to bottom. Since the edges falling completely within a cell form an interval hypergraph, we put the vertices within a cell into an order that supports this interval hypergraph; recall that this is an interval hypergraph of constant size. Hence, each edge falling within a cell is supported.

Now consider an edge e that spans several cells. If the interval that e spans consists of cells $c_i, \ldots, c_j$ with $1 \leq i < j \leq k$, it is supported by our order of vertices. On the other hand, if the cyclic interval of e is of the type $c_j \ldots, c_k, c_1, \ldots, c_i$ with $1 \leq i < j \leq k$, then e also contains the cells $c = c_k$ and $c' = c_1$ and, therefore, has been removed. □

Lemma 11 shows that any solution must have a pair of adjacent cells for which all hyperedges containing both must be removed. On the other hand, by Lemma 12, a single pair of adjacent cells with this property suffices. Hence, the strategy of using the pair with the minimum number of common hyperedges is optimal.

Runtime. By trying all ordered subsets of up to $d + 2$ vertices and then checking whether we find hyperedges connecting the vertices such that a given pattern is formed, we can search for any pattern in $O(mn^{d+2})$ time. This clearly leads to a polynomial time algorithm for bounded rank; however, with a more careful implementation, we can get rid of the exponential dependency on d, resulting in an implementation that runs in $O(m^2)$ time for m hyperedges, as we will now see.

The first phase of our algorithm consists mainly of searching for given patterns. The theory of fixed-parameter hardness tells us that searching for a pattern of size k is hard to achieve in time $n^{o(k)}$ since our problem generalizes the clique problem parameterized by size [3]. However, the structure of our problem allows us to do the search in $O(m^2)$ time as follows. First, we check for cycles by considering any edge e, choosing any pair v_1, v_2 of its up to d vertices (we have to try every pair), removing all edges containing both vertices, and then trying to find a shortest path from v_1 to v_2 using breadth-first search. If there is such a path of length $k \leq d$, we have found C_{k+1}, and we remove all its edges. Since any edge has to be considered only once—it is then either removed or cannot be part of a short cycle—this part takes $O(m^2)$ time.

For destroying the remaining types of patterns, we make use of the fact that each of them contains a specific *large edge* that contains all but one (O_2 and F_k), two (M_k), or three (O_1) vertices of the pattern. Hence, we test for every edge e whether it can play the role of the large edge in the corresponding pattern. Since e has at most d vertices (a constant), we can test any combination of its vertices as vertices of the large edge as shown in Fig. 13. Since there are only up to three more vertices not in e required, we could try all combinations for these and end up with an $O(m^2n^3)$-time algorithm. However, we can get rid of the factor n^3 as follows. Suppose that there are vertices $v_1, v_2 \in e$ and hyperedges e_1, e_2 so that $v_1 \in e_1, v_2 \in e_2$ but $v_1 \notin e_2$ and $v_2 \notin e_1$. If there is a vertex $v \in (e_1 \cap e_2) \setminus e$ in the intersection of e_1 and e_2 outside of e, then vertices v, v_1, and v_2 with hyperedges e, e_1, and e_2 form pattern C_3. However, we have already removed short cycles: a contradiction.

Now, consider the search for O_1. If for each of the three involved vertices in the larger hyperedge e we find a hyperedge containing vertices not in e, then we must have found O_1, otherwise the above argument yields C_3; see Algorithm 1. For the other patterns, we must additionally check whether there is at least one hyperedge realizing exactly each of the necessary pairwise adjacencies within e. For M_k, $k \leq d - 1$, this suffices to check for an occurrence; see Algorithm 2. For O_2 we must also check whether there is a hyperedge containing the two nonadjacent vertices of e and an element not in e; see Algorithm 3.

For F_k, $k \leq d - 2$, we need a vertex in the intersection of a hyperedge e'

```
foreach hyperedge $e$ do
    foreach triplet $V' = \{v_1, v_2, v_3\} \subseteq e$ do
        if $\exists e_1\colon V' \cap e_1 = \{v_1\}, e_1 \setminus e \neq \emptyset$ and
            $\exists e_2\colon V' \cap e_2 = \{v_2\}, e_2 \setminus e \neq \emptyset$ and
            $\exists e_3\colon V' \cap e_3 = \{v_3\}, e_3 \setminus e \neq \emptyset$ then
            return true
return false
```

Algorithm 1: Searching for O_1.

```
foreach hyperedge $e$ do
    foreach $V' = \{v_1, \dots, v_k, \tilde{v}\} \subseteq e$ do
        if $\exists e_0\colon V' \cap e_0 = \{v_1\}, e_0 \setminus e \neq \emptyset$ and
            $\exists e_k\colon V' \cap e_k = \{v_k\}, e_k \setminus e \neq \emptyset$ then
            if $\exists e_1\colon e_1 \cap V' = \{v_1, v_2\}$ and $\dots$ and
                $\exists e_{k-1}\colon e_{k-1} \cap V' = \{v_{k-1}, v_k\}$ then
                return true
return false
```

Algorithm 2: Searching for M_k with $k \leq d-1$.

```
foreach hyperedge $e$ do
    foreach $V' = \{v_0, v_1, v_2, v_3\} \subseteq e$ do
        if $\exists \tilde{e}\colon \tilde{e} \cap V' = \{v_1, v_2\}$ and $\tilde{e} \setminus e \neq \emptyset$ then
            if $\exists e_1\colon e_1 \cap V' = \{v_0, v_1\}$ and $\exists e_2\colon e_2 \cap V' = \{v_2, v_3\}$ then
                return true
return false
```

Algorithm 3: Searching for O_2.

```
foreach hyperedge $e$ do
    foreach $V' = \{v_0, \dots, v_k, \tilde{v}\} \subseteq e$ do
        if $\exists e_0\colon e_0 \cap V' = \{v_0, v_1\}$ and $\dots$ and
            $\exists e_{k-1}\colon e_{k-1} \cap V' = \{v_{k-1}, v_k\}$ then
            foreach hyperedge $\tilde{e}$ with $\tilde{e} \cap V' = \{v_1, \dots, v_k, \tilde{v}\}$ do
                mark all vertices of $\tilde{e} \setminus e$ as red
            foreach hyperedge $e'$ with $e' \cap V' = \{v_k\}$ do
                if $e'$ contains a red vertex then
                    return true
return false
```

Algorithm 4: Searching for F_k, $k \leq d-2$.

connecting the rightmost path-vertex to something outside of e with the second hyperedge $\tilde{e}$ containing $k+2$ vertices. This can be checked in $O(m)$ time by searching all feasible hyperedges and marking vertices outside of e if they lie in one such edge. Note that no hyperedge realizing one of the pairwise adjacencies of F_k can contain such a vertex of $e' \cap \tilde{e} \setminus e$ since our above argument yields C_3 in that case; see Algorithm 4.

Summing up, we can test in $O(m)$ time whether a given edge is the large edge—the edge of highest cardinality—of any pattern. After considering an edge it is either removed or we know that it is not contained as the large edge in any pattern, so we can make $\mathcal{H}$ $\mathcal{F}$-free in $O(m^2)$ time.

Then, we determine the connected components in linear time, find a cycle for each of them and initialize the cell order, in $O(n+m)$ time in total. For all components, the stepwise refinement can be done in $O(m^2)$ time in total. Counting the numbers of hyperedges between adjacent cells, determining the optimum splitting point, as well as finding the final order, can all be done in linear time (since the size of edges is constant). Therefore, the overall runtime is quadratic in the number of hyperedges.

Theorem 8 *We can find a $(d+1)$-approximation for* Interval Hypergraph Edge Deletion *on hypergraphs with m hyperedges of rank d in $O(m^2)$ time.*

8 Conclusion and Open Problems

We have considered the storyline visualization problem in connection with block crossings whose minimization helps to make such visualizations more readable. We have shown that minimizing the number of block crossings is NP-hard even for the case that all meetings involve only pairs of characters. Then, we have developed exact algorithms for the general case and a greedy heuristic for meetings of pairs. Our main result is a constant-factor approximation for minimizing block crossings for the case that each meeting is of bounded size. As a subproblem, we have considered the Interval Hypergraph Edge Deletion Problem, for which, too, we presented a constant-factor approximation on constant-rank hypergraphs.

While our paper yields insight into various aspects of SBCM, several interesting problems remain open.

- Does the greedy algorithm yield an approximation for 2-SBCM? Can it be reasonably generalized to more than two characters per meeting? Can we find an optimal starting permutation in polynomial time?

- Our experiments strongly suggest some relation between n, k, and the optimum in random instances, but we have not investigated this.

- Can we obtain approximation algorithms with better approximation ratios for any variant of the problem if we consider the start permutation part of the input and fixed?

- Can similar approximation results be obtained for simple crossings rather than block crossings? Since our analysis and algorithms heavily depend on the extended powers of block crossings, it seems hard to adjust our approach.

Acknowledgments

The work of M. Fink and S. Suri was partially supported by the National Science Foundation grant CCF-1525817. The work of F. Lipp was partially supported by Cusanuswerk. The work of T. C. van Dijk was partially supported by DFG grant Di 2161/2-1.

References

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

[2] L. Bulteau, G. Fertin, and I. Rusu. Sorting by transpositions is difficult. *SIAM J. Discrete Math.*, 26(3):1148–1180, 2012. doi:10.1137/110851390.

[3] J. Chen, X. Huang, I. A. Kanj, and G. Xia. Strong computational lower bounds via parameterized complexity. *J. Comp. System Sciences*, 72(8):1346–1367, 2006. doi:10.1016/j.jcss.2006.04.007.

[4] H. Eriksson, K. Eriksson, J. Karlander, L. Svensson, and J. Wästlund. Sorting a bridge hand. *Discrete Math.*, 241(1):289–300, 2001. doi:10.1016/S0012-365X(01)00150-9.

[5] M. Fink, S. Pupyrev, and A. Wolff. Ordering metro lines by block crossings. *J. Graph Algorithms Appl.*, 19(1):111–153, 2015. doi:10.7155/jgaa.00351.

[6] P. W. Goldberg, M. C. Golumbic, H. Kaplan, and R. Shamir. Four strikes against physical mapping of DNA. *J. Comput. Biol.*, 2(1):139–152, 1995. doi:10.1089/cmb.1995.2.139.

[7] M. Gronemann, M. Jünger, F. Liers, and F. Mambelli. Crossing minimization in storyline visualization. In M. Nöllenburg and Y. Hu, editors, *Proc. 24th Int. Symp. Graph Drawing Network Vis. (GD'16)*, volume 9801 of *LNCS*, pages 367–381. Springer, 2016. URL: https://arxiv.org/abs/1608.08027, doi:10.1007/978-3-319-50106-2_29.

[8] N. W. Kim, S. K. Card, and J. Heer. Tracing genealogical data with TimeNets. In G. Santucci, editor, *Proc. Int. Conf. Adv. Vis. Interfaces (AVI'10)*, pages 241–248, 2010. doi:10.1145/1842993.1843035.

[9] R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artif. Intell.*, 27(1):97–109, 1985. doi:10.1016/0004-3702(85)90084-0.

[10] I. Kostitsyna, M. Nöllenburg, V. Polishchuk, A. Schulz, and D. Strash. On minimizing crossings in storyline visualizations. In E. D. Giacomo and A. Lubiw, editors, *Proc. 23rd Int. Symp. Graph Drawing Network Vis. (GD'15)*, volume 9411 of *LNCS*, pages 192–198. Springer, 2015. URL: https://arxiv.org/abs/1509.00442, doi:10.1007/978-3-319-27261-0_16.

[11] C. J. Minard. The Russian campaign 1812–1813. Diagram available at https://commons.wikimedia.org/wiki/File:Minard.png, 1869. Accessed 2017/04/03.

[12] J. I. Moore Jr. Interval hypergraphs and D-interval hypergraphs. *Discrete Math.*, 17(2):173–179, 1977. doi:10.1016/0012-365X(77)90148-0.

[13] C. W. Muelder, T. Crnovrsanin, A. Sallaberry, and K. Ma. Egocentric storylines for visual analysis of large dynamic graphs. In *Proc. IEEE Int. Conf. Big Data*, pages 56–62, 2013. doi:10.1109/BigData.2013.6691715.

[14] R. Munroe. Movie narrative charts. Diagram available at https://xkcd.com/657/, 2009. Accessed 2017/04/03.

[15] Y. Tanahashi and K. Ma. Design considerations for optimizing storyline visualizations. *IEEE Trans. Vis. Comput. Graph.*, 18(12):2679–2688, 2012. doi:10.1109/TVCG.2012.212.

[16] W. T. Trotter and J. I. Moore. Characterization problems for graphs, partially ordered sets, lattices, and families of sets. *Discrete Math.*, 16(4):361–381, 1976. doi:10.1016/S0012-365X(76)80011-8.

[17] T. C. van Dijk, M. Fink, N. Fischer, F. Lipp, P. Markfelder, A. Ravsky, S. Suri, and A. Wolff. Block crossings in storyline visualizations. In M. Nöllenburg and Y. Hu, editors, *Proc. 24th Int. Symp. Graph Drawing Network Vis. (GD'16)*, volume 9801 of *LNCS*, pages 382–398. Springer, 2016. URL: https://arxiv.org/abs/1609.00321, doi:10.1007/978-3-319-50106-2_30.

[18] T. C. van Dijk, F. Lipp, P. Markfelder, and A. Wolff. Computing storyline visualizations with few block crossings. In F. Frati and K.-L. Ma, editors, *Proc. 25th Int. Symp. Graph Drawing Network Vis. (GD'17)*, LNCS. Springer, 2017. To appear.

[19] M. Yannakakis. Node- and edge-deletion NP-complete problems. In R. J. Lipton, W. Burkhard, W. Savitch, E. P. Friedman, and A. Aho, editors, *Proc. 10th Annu. ACM Symp. Theory Comput. (STOC'78)*, pages 253–264, 1978. doi:10.1145/800133.804355.