Journal of Graph Algorithms and Applications
http://jgaa.info/ vol. 19, no. 2, pp. 111–153 (2015)
DOI: 10.7155/jgaa.00351

Ordering Metro Lines by Block Crossings

Martin Fink[1] *Sergey Pupyrev*[2,3] *Alexander Wolff*[4]

[1]Department of Computer Science
University of California, Santa Barbara, USA
[2]Department of Computer Science
University of Arizona, Tucson, USA
[3]Institute of Mathematics and Computer Science
Ural Federal University, Ekaterinburg, Russia
[4]Lehrstuhl für Informatik I
Universität Würzburg, Germany
http://www1.informatik.uni-wuerzburg.de/en/staff/wolff_alexander/

Abstract

A problem that arises in drawings of transportation networks is to minimize the number of crossings between different transportation lines. While this can be done efficiently under specific constraints, not all solutions are visually equivalent. We suggest merging single crossings into *block crossings*, that is, crossings of two neighboring groups of consecutive lines.

Unfortunately, minimizing the total number of block crossings is NP-hard even for very simple graphs. We give approximation algorithms for special classes of graphs and an asymptotically worst-case optimal algorithm for block crossings on general graphs. Furthermore, we show that the problem remains NP-hard on planar graphs even if both the maximum degree and the number of lines per edge are bounded by constants; on trees, this restricted version becomes tractable.

Submitted:	Reviewed:	Revised:	Accepted:	Final:
March 2014	August 2014	January 2015	January 2015	February 2015

Published:
February 2015

Article type:	Communicated by:
Regular Paper	H. Meijer

Some of the results in this paper have been presented at the *38th International Symposium on Mathematical Foundations of Computer Science* (MFCS 2013) [16].

E-mail addresses: fink@cs.ucsb.edu (Martin Fink) spupyrev@gmail.com (Sergey Pupyrev)

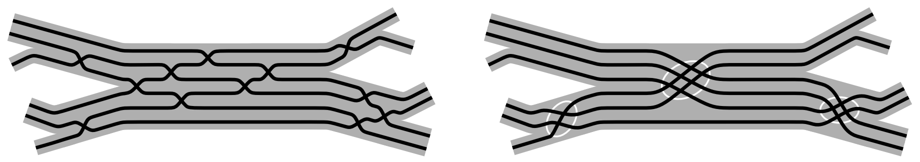

(a) 12 pairwise crossings. (b) 12 pairwise crossings grouped into 3 block crossings.

Figure 1: Optimal orderings of a metro network.

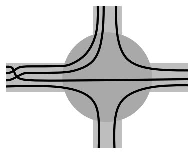
1 Introduction

A well-known visualization problem is creating drawings of transportation networks like metro maps. An important part of such networks are transportation *lines* that connect different points using streets or railway tracks of the underlying network. It is easy to model such networks by graphs. The vertices are stations or intersections and the edges represent railway tracks or road segments connecting the vertices. The lines then become *paths* in the graph. In many metro maps and other transportation networks some edges are used by several lines. Usually, to visualize such networks, lines that share an edge are drawn individually along the edge in distinct colors. Often, some lines must cross, and it is desirable to draw the lines with few crossings. The *metro-line crossing minimization* problem has been introduced [5] in 2006. The goal is to order the lines along each edge such that the number of crossings is minimized. So far, the focus has been on the number of crossings and not on their visualization, although two line orders with the same crossing number may look quite differently; see Figure 1.

Our aim is to improve the readability of metro maps by computing line orders that are aesthetically more pleasing. To this end, we merge *pairwise* crossings into crossings of blocks of lines minimizing the number of *block crossings* in the map. Informally, a block crossing is an intersection of two neighboring groups of consecutive lines sharing the same edge; see Figure 1b. We consider two variants of the problem. In the first variant, we want to find a line ordering with the minimum number of block crossings. In the second variant, we want to minimize both pairwise and block crossings.

1.1 Motivation

Although we present our results in terms of the problem of metro-map visualization, crossing minimization between paths on an embedded graph is used in various fields. In very-large-scale integrated (VLSI) chip layout, a wire diagram should have few wire crossings [18]. Another application is the visualization of biochemical pathways [26]. In graph drawing, the number of edge crossings is considered one of the most popular aesthetic criteria. Recently, a lot of research, both in graph drawing and information visualization, has been devoted to *edge bundling.* In this setting, some edges are drawn close together—like metro lines—


(a) No unavoidable vertex crossings.

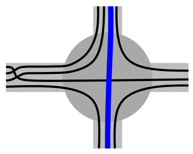

(b) An unavoidable vertex crossing in the center of the vertex.

Figure 2: Consistent line orders with and without unavoidable vertex crossings.

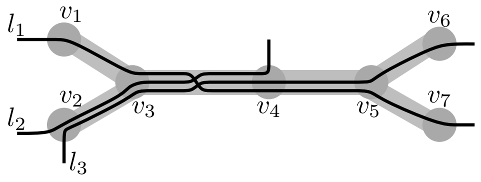


Figure 3: Lines ℓ_1 and ℓ_3 have an unavoidable crossing on edge (v_3, v_4) that could also be placed on (v_4, v_5). Avoidable crossings (such as ℓ_2 and ℓ_3) are forbidden in solutions with monotone block crossings.

which emphasizes the structure of the graph [10, 20, 25]. Block crossings can greatly improve the readability of bundled graph drawings.

1.2 Problem Definition

The input consists of an embedded graph $G = (V, E)$, and a set $\mathcal{L} = \{\ell_1, \ldots, \ell_{|\mathcal{L}|}\}$ of simple paths in G. We call G the *underlying network* and the paths *lines*. Note that the embedding of G does not necessarily have to be planar; we are only interested in the clockwise order of edges incident to a vertex. The vertices of G are *stations* and the endpoints v_0, v_k of a line $(v_0, \ldots, v_k) \in \mathcal{L}$ are *terminals*. For each edge $e = (u, v) \in E$, let $L_e = L_{uv}$ be the set of lines passing through e. At an interior point of an edge e, the order of lines is defined by a sequence $\pi = [\pi_1, \ldots, \pi_n]$ with $\{\pi_1, \ldots, \pi_n\} = L_e$. For $i \leq j < k$, a *block move* (i, j, k) on the sequence $\pi = [\pi_1, \ldots, \pi_n]$ of lines on e is the exchange of two consecutive blocks $\pi_i, \ldots, \pi_j$ and $\pi_{j+1}, \ldots, \pi_k$. Interpreting edge $e = (u, v)$ as being directed from u to v, we define a *line order*, $\pi^0(e), \ldots, \pi^{t(e)}(e)$, on e as follows. The initial sequence, $\pi^0(e)$, is the order of lines L_e at the beginning of e (that is, at vertex u), $\pi^{t(e)}(e)$ is the order at the end of e (that is, at vertex v), and, for $i = 1, \ldots, t(e)$, the sequence $\pi^i(e)$ is an ordering of L_e that is derived from $\pi^{i-1}(e)$ by a block move. A line order with these properties that consists of $t + 1$ sequences gives rise to t *block crossings*.

Following previous work [1, 23], we use the *edge crossings* model, that is, we do not hide crossings under station symbols if possible. Two lines sharing at least one common edge either do not cross or cross each other on an edge but never in a vertex; see Figure 2a. For pairs of lines sharing a vertex but no edges, crossings at the vertex are allowed and not counted as they exist in any solution. We call such crossings *unavoidable vertex crossings*; see Figure 2b. If the line orders on the edges incident to a vertex v produce only edge crossings and unavoidable vertex crossings, we call them *consistent* in v. Line orders for all edges are consistent if they are consistent in all vertices. Formally, we check consistency of line orders in a vertex v by considering each edge e incident to v.

At the end of e at vertex v the order of the lines in L_e is fixed. The other edges $e_1, \ldots, e_k$ incident to v together contain all lines of L_e that do not terminate in v. The combined order of these lines on edges $e_1, \ldots, e_k$ must be the same as their order on e; otherwise, lines of L_e would cross in v.

The *block crossing minimization* (BCM) problem is defined as follows.

Problem 1 (BCM) *Let $G = (V, E)$ be an embedded graph and let $\mathcal{L}$ be a set of lines on G. For each edge $e \in E$, find a line order $\pi^0(e), \ldots, \pi^{t(e)}(e)$ such that the line orders on all edges are consistent and the total number of block crossings, $\sum_{e \in E} t(e)$, is minimum.*

Note that crossings between edges of G are allowed. As the lines on two crossing edges cross in any case, there is no need to count such crossings.

In this paper, we restrict our attention to instances with two properties. First, the intersection of two lines, that is, the edges and vertices they have in common, forms a path (**path intersection property**). This includes the cases that the intersection is empty or a single vertex, but excludes pairs of lines that have two or more disjoint intersections. Second, any line terminates at vertices of degree one and no two lines terminate at the same vertex (**path terminal property**). The first property is introduced for simplicity of presentation; our results do hold for the general case (after straightforward reformulation), as every common subpath of two lines can be considered individually. The second property, as shown by Nöllenburg [23], is equivalent to restricting each line to be placed in a prescribed outermost position (left or right side) in its terminal. The restriction is introduced to improve readability of metro-map visualizations; it has been utilized in several earlier works [2, 4, 23]. We call the underlying network without the vertices of degree one and their incident edges the *skeleton*. Due to the path terminal property, crossings of lines occur only on edges of the skeleton.

If both properties hold, a pair of lines either has to cross, that is, a crossing is *unavoidable*, or it can be kept crossing-free, that is, a crossing is *avoidable*; see Figure 3. The orderings that are optimal with respect to pairwise crossings are exactly the orderings that contain just unavoidable crossings (Lemma 2 in the paper of Nöllenburg [23]); that is, any pair of lines crosses at most once, in an equivalent formulation. Intuitively, double crossings of lines can easily be eliminated by rerouting the two lines, thus decreasing the number of crossings. As this property is also desirable for block crossings, we use it to define the *monotone block crossing minimization* (MBCM) problem. Note that feasible solutions of MBCM must have the minimum number of pairwise crossings.

Problem 2 (MBCM) *Given an instance $(G = (V, E), \mathcal{L})$ of BCM, find a feasible solution that minimizes the number of block crossings subject to the constraint that any two lines cross at most once.*

There are instances for which BCM allows fewer block crossings than MBCM does; see Figure 4 in Section 2.

Table 1: Overview of our results for BCM and MBCM.

graph class (skeleton)	BCM		MBCM	
single edge	11/8-approx.	[12]	3-approx.	Sec. 2
path	3-approx.	Sec. 3.1	3-approx.	Sec. 3.2
tree	$\leq 2\lvert\mathcal{L}\rvert - 3$ cross.	Sec. 4	$\leq 2\lvert\mathcal{L}\rvert - 3$ cross.	Sec. 4
upward tree	6-approx.	Sec. 4.2	6-approx.	Sec. 4.2
general graph	$\mathcal{O}(\lvert\mathcal{L}\rvert\sqrt{\lvert E\rvert})$ cross.	Sec. 5	$\mathcal{O}(\lvert\mathcal{L}\rvert\sqrt{\lvert E\rvert})$ cross.	Sec. 5
	bounded degree & edge multiplicity			
tree	FPT	Sec. 6.1	FPT	Sec. 6.1
planar graph	NP-hard	Sec. 6.2	NP-hard	Sec. 6.2

1.3 Our Contribution

We introduce the new problems BCM and MBCM. To the best of our knowledge, ordering lines by block crossings is a new direction in graph drawing. So far BCM has been investigated only for the case that the skeleton, that is, the graph without terminals, is a single edge [3], while MBCM is a completely new problem. Table 1 summarizes our results.

We first analyze MBCM on a single edge (Section 2), exploiting, to some extent, the similarities to *sorting by transpositions* [3]. Then, we use the notion of *good pairs* of lines, that is, lines that should be neighbors, for developing an approximation algorithm for BCM on graphs whose skeleton is a path (Section 3); we properly define good pairs so that changes between adjacent edges are taken into account. Yet, good pairs cannot always be kept close; we introduce a good strategy for breaking pairs when needed.

Unfortunately, the approximation algorithm does not generalize to trees. We do, however, develop a worst-case optimal algorithm for trees (Section 4). It needs $2|\mathcal{L}| - 3$ block crossings and there are instances in which this number of block crossings is necessary in any solution. We then use our algorithm for obtaining approximate solutions for MBCM on the special class of *upward trees*.

As our main result, we present an algorithm for obtaining a solution for BCM on general graphs (Section 5). We show that the solutions constructed by our algorithm contain only monotone block moves and are, therefore, also feasible solutions for MBCM. We show that our algorithm always yields $\mathcal{O}(|\mathcal{L}|\sqrt{|E|})$ block crossings. While the algorithm itself is simple and easy to implement, proving the upper bound is non-trivial. Next, we show that the bound is tight; we use a result from projective geometry for constructing worst-case examples in which any feasible solution contains $\Omega(|\mathcal{L}|\sqrt{|E|})$ block crossings. Hence, our algorithm is asymptotically worst-case optimal.

Finally, we consider the restricted variant of the problems in which the maximum degree Δ as well as the maximum *edge multiplicity* c (the maximum number of lines per edge) are bounded (Section 6). For the case where the underlying network is a tree, we show that both BCM and MBCM are fixed-parameter tractable with respect to the combined parameter $\Delta + c$. On the other hand, we prove that both variants are NP-hard on general graphs even if both Δ and c are constant.

1.4 Related Work

Line crossing problems in transportation networks were first studied by Benkert et al. [5], who considered the *metro-line crossing minimization* problem (MLCM) on a single edge. Recently, Fink and Pupyrev showed that the general version of MLCM is NP-hard even on caterpillar graphs [15]; no efficient algorithms are known for the case of two or more edges. The problem has, however, been studied under additional restrictions. Bekos et al. [4] addressed the problem on paths and trees. They also proved hardness of a variant, called MLCM-P, in which all lines must be placed outermost in their terminals. Okamoto et al. [24] presented exact (exponential-time) and fixed-parameter tractable algorithms for MLCM-P on paths. Fink and Pupyrev presented a polynomial-time approximation algorithm for this variant and showed that it is fixed-parameter tractable with respect to the number of crossings on general graphs [15]. For general graphs with the path terminal property, Asquith et al. [2], Argyriou et al. [1], and Nöllenburg [23] devised polynomial-time algorithms. All these works are dedicated to pairwise crossings, the optimization criterion being the number of crossing pairs of lines.

A lot of recent research, both in graph drawing and information visualization, is devoted to *edge bundling* where some edges are drawn close together—like metro lines—thus emphasizing the structure of the graph [10, 17, 20]. Pupyrev et al. [25] studied MLCM in this context and suggested a linear-time algorithm for MLCM on instances with the path terminal property.

A closely related problem arises in VLSI design, where the goal is to minimize intersections between nets (physical wires) [18, 21]. Net patterns with fewer crossings are likely to have better electrical characteristics and require less wiring area as crossings consume space on the circuit board; hence, it is an important optimization criterion in circuit board design. Marek-Sadowska and Sarrafzadeh [21] considered not only minimizing the number of crossings, but also suggested distributing the crossings among circuit regions in order to simplify net routing.

As we will later see, BCM on a *single* edge is equivalent to the problem of sorting a permutation by block moves, which is well studied in computational biology for DNA sequences; it is known as *sorting by transpositions* [3,9]. The task is to find the shortest sequence of block moves transforming a given permutation into the identity permutation. BCM is, hence, a generalization of sorting by transpositions from a single edge to graphs. The complexity of sorting by transpositions was open for a long time; only recently it has been shown to be NP-hard [7]. The currently best known algorithm has an approximation ratio of 11/8 [12]. The proof of correctness of that algorithm is based on a computer analysis, which verifies more than 80, 000 configurations. To the best of our knowledge, no tight upper bound for the necessary number of steps in sorting by transpositions is known. There are several variants of sorting by transpositions; see the survey of Fertin et al. [14]. For instance, Vergara et al. [19] used *correcting short block moves* to sort a permutation. In our terminology, these are monotone moves such that the combined length of exchanged blocks does not exceed three. Hence, their problem is a restricted variant of MBCM on a single edge; its complexity is unknown. The general problem of sorting by (unrestricted) monotone block moves has not been considered, not even on a single edge.

2 Block Crossings on a Single Edge

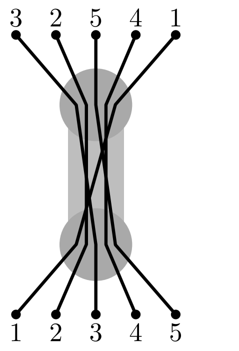
For getting a feeling for the problem, we restrict our attention to the simplest networks consisting of a single edge (the skeleton) with multiple lines passing through it, starting and ending in leaves; see Figure 4a. Subsequently, we will be able to reuse some of the ideas for a single edge for graphs whose skeleton is a longer path or even a tree.

On a single edge, BCM can be reformulated as follows. We choose a direction for the skeleton edge $e = (u, v)$, for example, from bottom (u) to top (v). Then, any line passing through e starts on the bottom side in a leaf attached to u and ends at the top side in a leaf attached to v. Suppose we have n lines $\ell_1, \dots, \ell_n$. The indices of the lines and the order of the edges incident to u and v yield an induced order τ (as a permutation of $\{1, \dots, n\}$) of the lines on the bottom side of e, that is, at u, and an induced order π of the lines at the top side of e; see Figure 4a.

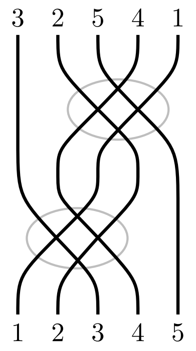
Given these two permutations π and τ, the problem now is to find a shortest sequence of block moves transforming π into τ. By relabeling the lines we can assume that τ is the identity permutation, and the goal is to sort π. This problem is *sorting by transpositions* [3], which is hence a special case of BCM. Sorting by transpositions is known to be NP-hard as Bulteau et al. [7] showed. Hence, BCM, as a generalization, is also NP-hard.

Theorem 1 *BCM is NP-hard even if the underlying network is a single edge with attached terminals.*

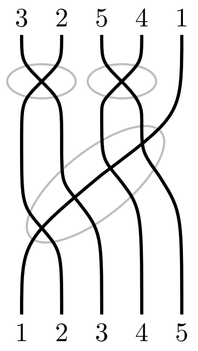
Sorting by transpositions is quite well investigated; Elias and Hartmann [12] presented an 11/8-approximation algorithm for the problem. Hence, we concentrate on the new problem of sorting with monotone block moves; this means that the relative order of any pair of elements changes at most once. The problems are not equivalent; see Figure 4 for an example where dropping monotonicity reduces the number of block crossings in optimum solutions. Hence, we do not know the complexity of MBCM on a single edge. The problem is probably NP-hard even on a single edge, but even for BCM (that is, sorting by transpositions) the NP-hardness proof is quite complicated. As we are mainly interested in more complex networks, we just give an approximation algorithm for MBCM on a single edge. Later, we show that on general planar graphs, MBCM is indeed NP-hard even if there are few lines per edge (see Section 6.2).



(a) Input instance with six pairwise crossings.



(b) Two block moves on the central edge.



(c) Three monotone block moves on the central edge.

Figure 4: Sorting permutation [3, 2, 5, 4, 1] by block moves and by monotone block moves; both types of block moves are enclosed by ellipses.

In the example of Figure 4, the optimum solution contains a pair of lines (lines 2 and 4) that cross twice. However, by replacing each of the two lines by a large number k of parallel lines, we get an examle where saving a single block crossing can lead to a solution in which almost every pair of lines crosses twice. Hence, using non-monotone solutions, even if they are optimal, can cost a lot in terms of pairwise crossings.

Next we present a simple 3-approximation algorithm for MBCM on instances whose skeleton consists of a single edge.

2.1 Terminology

We first introduce some terminology following previous work where possible. Let $\pi = [\pi_1, \ldots, \pi_n]$ be a permutation of n elements. For convenience, we assume that there are extra elements $\pi_0 = 0$ and $\pi_{n+1} = n + 1$ at the beginning of the permutation and at the end, respectively. A *block* in π is a sequence of consecutive elements $\pi_i, \ldots, \pi_j$ with $1 \leq i \leq j \leq n$. A *block move* (i, j, k) with $1 \leq i \leq j < k \leq n$ on π maps $\pi = [\ldots, \pi_{i-1}, \pi_i, \ldots, \pi_j, \pi_{j+1}, \ldots \pi_k, \pi_{k+1}, \ldots]$ to $[\ldots, \pi_{i-1}, \pi_{j+1}, \ldots, \pi_k, \pi_i \ldots, \pi_j, \pi_{k+1}, \ldots]$, that is, exchanges the blocks $\pi_i, \ldots, \pi_j$ and $\pi_{j+1}, \ldots, \pi_k$. As an example, let $\sigma = [4, 2, 3, 5, 1]$. The block move $(1, 3, 4)$ maps σ to the permutation $[5, 4, 2, 3, 1]$.

A block move (i, j, k) is *monotone* if $\pi_q > \pi_r$ for all $i \leq q \leq j < r \leq k$, that is, if every element in the first block $\pi_i, \ldots, \pi_j$ is greater than every element of the second block $\pi_{j+1}, \ldots, \pi_k$. For our example permutation σ, the block move $(1, 3, 4)$ is not monotone, but the block move $(1, 1, 3)$, which maps σ to $[2, 3, 4, 5, 1]$, is monotone. We denote the minimum number of block moves needed to sort π by $\mathrm{bc}(\pi)$, and the minimum number of monotone block moves needed for sorting π by $\mathrm{mbc}(\pi)$. In our example, one can check that $\mathrm{bc}(\sigma) = \mathrm{mbc}(\sigma) = 2$.

An ordered pair (π_i, π_{i+1}) (with $0 \leq i \leq n$) is a *good pair* if $\pi_{i+1} = \pi_i + 1$, and a *breakpoint* otherwise. In our example, permutation σ has only one good pair, $(2,3)$, and five breakpoints, namely $(4,2)$, $(3,5)$, $(5,1)$, $(0,4)$, $(1,6)$; the latter two because we assumed that there are extra elements 0 and $n+1$. Intuitively, sorting π is a process of creating good pairs (or destroying breakpoints) by block moves. The identity permutation $[1,\dots,n]$ is the only permutation that has only good pairs and no breakpoints.

A permutation is *simple* if it contains no good pairs. Any permutation can be uniquely simplified without affecting its distance to the identity permutation [9]. This is done by "glueing" good pairs together, that is, treating the two lines as one line and relabeling. The simplification of our example permutation σ is the permutation $[3,2,4,1]$, where element 2 represents the elements 2 and 3 of σ. A breakpoint (π_i,π_{i+1}) is a *descent* if $\pi_i > \pi_{i+1}$, and a *gap* otherwise. We use $\mathrm{bp}(\pi)$, $\mathrm{des}(\pi)$, and $\mathrm{gap}(\pi)$ to denote the number of breakpoints, descents, and gaps in π. In our example σ, $(4,2)$ is a descent, $(3,5)$ is a gap, and we have $\mathrm{bp}(\sigma)=5$, $\mathrm{des}(\sigma)=2$, and $\mathrm{gap}(\sigma)=3$. The *inverse* of π is the permutation π^{-1} in which each element and the index of its position are exchanged, that is, $\pi^{-1}_{\pi_i} = i$ for $1 \le i \le n$. In our example, $\sigma^{-1} = [5,2,3,1,4]$. A descent in π^{-1}, that is, a pair of elements (π_i,π_j) with $\pi_i = \pi_j + 1$ and $i < j$, is called an *inverse descent* in π. Analogously, an *inverse gap* is a pair of elements (π_i,π_j) with $\pi_i = \pi_j + 1$ and $i > j+1$. For example, σ has the inverse descent $(2,1)$ and the inverse gap $(5,4)$. Now, we give lower and upper bounds for MBCM, that is, on $\mathrm{mbc}(\pi)$.

As there are $n+1$ pairs (π_i,π_{i+1}) with $0 \le i \le n$ and any such pair is either a good pair or a breakpoint, we have $\mathrm{gp}(\pi) + \mathrm{bp}(\pi) = n+1$ for any permutation π. The number $\mathrm{bp}(\pi) = n+1-\mathrm{gp}(\pi)$ of breakpoints can, hence, be interpreted as the number of missing good pairs because the identity permutation $\mathrm{id} = [1,\dots,n]$ has $\mathrm{bp}(\mathrm{id}) = 0$ and $\mathrm{gp}(\mathrm{id}) = n+1$. The identity permutation is the only permutation with this property. Recall that a simple permutation τ does not have good pairs. Hence, $\mathrm{gp}(\tau) = 0$ and $\mathrm{bp}(\tau) = n+1$.

2.2 A Simple Approximation

A block move affects three pairs of adjacent elements. Therefore, the number of breakpoints can be reduced by at most three in any block move. This implies $\mathrm{mbc}(\pi) \ge \mathrm{bc}(\pi) \ge \lceil \mathrm{bp}(\pi)/3 \rceil$ for any permutation π as Bafna and Pevzner [3] pointed out. It is easy to see that $\mathrm{bp}(\pi) - 1$ moves suffice for sorting any permutation, which yields a simple 3-approximation for BCM.

We suggest the following algorithm for sorting a *simple* permutation π using only monotone block moves: In each step find the smallest i such that $\pi_i \ne i$ and move element i to position i, that is, exchange blocks $\pi_i,\dots,\pi_{k-1}$ and π_k, where $\pi_k = i$. Clearly, the step destroys at least one breakpoint, namely $(\pi_{i-1} = i-1, \pi_i)$. Furthermore, the move is monotone as element i is moved only over larger elements. Therefore, $\mathrm{mbc}(\pi) \le \mathrm{bp}(\pi)$ and the algorithm yields a 3-approximation. By first simplifying a permutation, applying the algorithm, and then undoing the simplification, we can also find a 3-approximation for permutations that are not simple.

Theorem 2 *There is a 3-approximation algorithm for MBCM on networks whose skeleton is a single edge. The algorithm finds a sequence of block moves in $\mathcal{O}(n\sqrt{\log n})$ time; reporting the corresponding sequence of permutations takes $\mathcal{O}(n^2)$ time.*

Proof: Note that the trivial implementation of the 3-approximation runs in $\mathcal{O}(n^2)$ time. If we need to output the permutations after each block move, $\Omega(n^2)$ time is also necessary because there can be a linear number of block moves (for example, for permutation $[n, n-1, \dots, 2, 1]$), and it takes linear time to output each permutation. If we are only interested in the sequence of block moves (i, j, k), the running time can be improved by proceeding as follows.

Recall that it suffices to consider a *simple* permutation π. In increasing order, we move elements $i = 1, \dots, n-1$. In every step, the monotone block move is described by $(i, k-1, k)$, where k is the current index of element i. Our crucial observation is that the current index of i increases by one whenever a block move is performed with an element $j < i$ located to the right of i in the input permutation. In other words, the total increase of the index of element i is the number of *inversions* in π for i: the number of elements in π that are smaller than i and located to the right of i.

It is well known that the number of inversions for each element in a permutation can be found in $\mathcal{O}(n \log n)$ total time using, for example, Mergesort. Chan and Pătraşcu [8] showed, however, how to solve the problem in $\mathcal{O}(n\sqrt{\log n})$ time. $\square$

Note that any future improvement of the time bound needed for counting inversions in a permutation automatically improves the time bound stated in the theorem.

2.3 Lower Bounds

The following observations yield better lower bounds than $\text{mbc}(\pi) \geq \lceil \text{bp}(\pi)/3 \rceil$.

Lemma 1 *In a monotone block move, the number of descents in a permutation decreases by at most one, and the number of gaps decreases by at most two.*

Proof: Consider a monotone move that transfers $[\dots a \mid b \dots c \mid d \dots e \mid f \dots]$ to $[\dots a \mid d \dots e \mid b \dots c \mid f \dots]$; it affects three adjacencies. Suppose a descent is destroyed between a and b, that is, $a > b$ and $a < d$. Then, $b < d$, which contradicts monotonicity. Similarly, no descent can be destroyed between e and f. The fact that c passed d in a monotone move yields $c > d$. Hence, no gap is destroyed between c and d. $\square$

A similar statement holds for inverse descents and gaps.

Lemma 2 *In a monotone block move, the number of inverse descents decreases by at most one, and the number of inverse gaps decreases by at most two.*

Proof: Consider a monotone exchange of blocks $\pi_i, \ldots, \pi_j$ and $\pi_{j+1}, \ldots, \pi_k$. Note that inverse descents can only be destroyed between elements π_q $(i \le q \le j)$ and π_r $(j+1 \le r \le k)$. Suppose that the move destroys two inverse descents such that the first block contains elements $x+1$ and $y+1$, and the second block contains x and y. Since the block move is monotone, $y+1 > x$ and $x+1 > y$, which means that $x = y$.

On the other hand, there cannot be inverse gaps between elements π_q $(i \le q \le j)$ and π_r $(j+1 \le r \le k)$. Therefore, there are only two possible inverse gaps between π_{i-1} and π_r $(j < r \le k)$, and between π_q $(i \le q \le j)$ and π_{k+1}. □

Combining the lemmas, we obtain the following result.

Theorem 3 *A lower bound on the number of monotone block moves needed to sort a permutation π is*

$$\mathrm{mbc}(\pi) \ge \left\lceil \max(\mathrm{bp}(\pi)/3, \mathrm{des}(\pi), \mathrm{gap}(\pi)/2, \mathrm{des}(\pi^{-1}), \mathrm{gap}(\pi^{-1})/2) \right\rceil .$$

2.4 An Upper Bound

To construct a better upper bound than $\mathrm{mbc}(\pi) \le \mathrm{bp}(\pi)$, we first consider a constrained sorting problem in which at least one of the moved blocks has unit size; that is, we allow only block moves of types (i, i, k) and $(i, k-1, k)$. Let $\mathrm{mbc}^1(\pi)$ be the minimum number of such block moves needed to sort π. We show how to compute $\mathrm{mbc}^1(\pi)$ exactly. An *increasing subsequence* of π is a sequence $\pi_{l_1}, \pi_{l_2}, \ldots$ such that $\pi_{l_1} < \pi_{l_2} < \ldots$ and $l_1 < l_2 < \ldots$. Let $\mathrm{lis}(\pi)$ be the size of the longest increasing subsequence of π.

Lemma 3 $\mathrm{mbc}^1(\pi) = n - \mathrm{lis}(\pi)$.

Proof: We first show $\mathrm{mbc}^1(\pi) \ge n - \mathrm{lis}(\pi)$. Consider a monotone unit move $\sigma = (i, i, k)$ in π ($\sigma = (i, k-1, k)$ is symmetric). Let $\tilde{\pi} = [\pi_1, \ldots, \pi_{i-1}, \pi_{i+1}, \ldots, \pi_n]$ be the permutation π without element π_i. Clearly, $\mathrm{lis}(\tilde{\pi}) \le \mathrm{lis}(\pi)$. If we apply σ, the resulting permutation $\sigma\pi = [\pi_1, \ldots, \pi_{i-1}, \pi_{i+1}, \ldots, \pi_k, \pi_i, \pi_{k+1}, \ldots, \pi_n]$ has one extra element compared to $\tilde{\pi}$, and, therefore, $\mathrm{lis}(\sigma\pi) \le \mathrm{lis}(\tilde{\pi}) + 1$. Hence, $\mathrm{lis}(\sigma\pi) \le \mathrm{lis}(\pi) + 1$, that is, a monotone unit move cannot increase the length of the longest increasing subsequence by more than one. The inequality follows since $\mathrm{lis}(\tau) = n$ for the identity permutation τ.

Next, we show $\mathrm{mbc}^1(\pi) \le n - \mathrm{lis}(\pi)$. Let $S = [\ldots s_1 \ldots s_2 \ldots s_{\mathrm{lis}} \ldots]$ be a fixed longest increasing subsequence in π. We show how to choose a move that increases the length of S. Let $\pi_i \notin S$ be the rightmost element (that is, i is maximum) lying between elements s_j and s_{j+1} of S so that $\pi_i > s_{j+1}$. We move π_i rightwards to its proper position p_i inside S. This is a monotone move, as π_i was chosen rightmost. If no such π_i exists, we symmetrically choose the leftmost π_i with $\pi_i < s_j$ and bring it into its proper position in S. In both cases S grows.

Together, the two inequalities yield the desired equality. □

Corollary 1 *Every permutation can be sorted by $n - \mathrm{lis}(\pi)$ monotone block moves.*

The Erdős-Szekeres theorem implies that if $n \geq (\text{lis} - 1) \cdot (\text{lds} - 1) + 1$, then the longest increasing subsequence has length $k \geq \text{lis}$, or there is a decreasing subsequence of length $k' \geq \text{lds}$. With the currently known bounds, however, this does not lead to a better approximation. There exist permutations in which both the longest increasing and the longest decreasing subsequences have length $\mathcal{O}(\sqrt{n})$; for example, $[7, 8, 9, 4, 5, 6, 1, 2, 3]$ is such a permutation for $n = 9$. The new upper bound is $\mathcal{O}(n - \sqrt{n})$, while the lower bound—using the number of breakpoints—is only $\mathcal{O}(\sqrt{n})$, which does not yield a constant approximation factor (the 3-approximation can, of course, still be applied).

Another possible direction for improving the approximation factor would be to relate the optimal solutions of BCM and MBCM, that is, to find $r = \sup_\pi\{\text{mbc}(\pi)/\text{bc}(\pi)\}$. On one hand, $\text{mbc}(\pi) \leq \text{bp}(\pi) \leq 3\,\text{bc}(\pi)$ for all permutations, that is, $r \leq 3$. On the other hand, $\text{bc}(\pi) = \lceil (n+1)/2 \rceil$ and $\text{mbc}(\pi) = n - 1$ for permutation $[n, n-1, \dots, 2, 1]$ (see [13]), that is, $r \geq 2$. If we had a constructive proof showing that $r < 24/11 \approx 2.18$, we could combine this with the 11/8-approximation algorithm for BCM [12]. This would yield an algorithm for MBCM with approximation factor $r \cdot 11/8 < 3$.

3 Block Crossings on a Path

We consider an embedded graph $G = (V, E)$ consisting of a path $P = (V_P, E_P)$—the skeleton—with attached terminals. For every vertex $v \in V_P$, the clockwise order of terminals adjacent to v is given, and we assume that the path is oriented from left to right. We say that a line *starts* at its leftmost vertex on P and *ends* at its rightmost vertex on P. As we consider only crossings of lines sharing an edge, we assume that the terminals connected to any path vertex v are in such an order that first lines end at v and then lines start at v; see Figure 5. We say that a line starts/ends above the path if its respective terminal is above the path and that it starts/ends below the path if the respective terminal is below the path. Similarly, we say that a line enters the path from the top or from the bottom depending on the position of its first terminal above or the below the path, and we say that the line leaves the path to the top or to the bottom depending on the position of its second terminal with respect to the path. Suppose that there are several lines that enter the path from above in vertex v. We say that the first line entering P in vertex v from above is the line whose terminal is leftmost since it is the first line that will join the lines on the path. Similarly, the last line ending at v above P is the line ending in the rightmost terminal above P at v. Symmetrically, we can define the first line starting in v below the path and the last line ending in v below the path. We first focus on developing an approximation algorithm for BCM. Then, we show how to modify the algorithm for monotone block crossings.

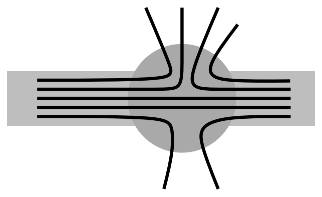


Figure 5: Lines starting and ending around a vertex of the path.

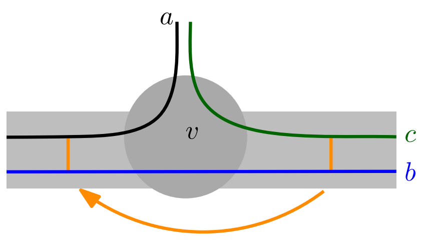


Figure 6: Inheritance of a good pair; (a, b) inherits from (c, b).

3.1 BCM on a Path

Recall that, as a generalization of *sorting by transpositions*, BCM on a path is NP-hard; see Theorem 1. We suggest a 3-approximation algorithm for BCM. Similar to the single edge case, the basic idea of the algorithm is to consider good pairs of lines. A *good pair* is, intuitively, an ordered pair of lines that will be adjacent—in this order—in any feasible solution when one of the lines ends. We argue that our algorithm creates at least one additional good pair per block crossing, while even the optimum creates at most three new good pairs per crossing. To describe our algorithm we first define good pairs.

Definition 1 (Good pair) *Let a and b be two lines. The ordered pair (a, b) is a* good pair *if one of the following two conditions holds.*

(i) Lines a and b end in the same vertex $v \in P$ and a and b are consecutive in clockwise order around v.

(ii) There are a line c and an interior vertex v of the path P such that c is the first line that enters P in v from above, a is the last line ending in v above P as shown in Figure 6, and (c, b) is a good pair.

(iii) Symmetrically to case (ii), there are a line c and an interior vertex v of the path P such that c is the first line that enters P in v from below, b is the last line ending in v below P, and (a, c) is a good pair.

Note that case (i) of the definition follows the definition of good pairs on a single edge; compare Section 2. In case (ii) we say that the good pair (a, b) is *inherited* from (c, b) and identify (a, b) with (c, b), which is possible as a and c do not share an edge. Analogously, in case (iii) the good pair (a, b) is inherited from (a, c) and we identify (a, b) with (a, c).

As a preprocessing step, we add two virtual lines, t_e and b_e, to each edge $e \in E_P$. Line t_e is the last line entering P before e from the top and the first line leaving P after e to the top. Symmetrically, b_e is the last line entering P before e from the bottom and the first line leaving P after e to the bottom. Although virtual lines are never moved, t_e participates in good pairs, which models the fact that the first line ending after an edge must be brought to the top. Symmetrically, b_e participates in good pairs modeling the fact that the first line ending after an edge must be brought to the bottom.

We now investigate some properties of good pairs.

Lemma 4 *Let $e \in E_p$ be an edge and let $\ell \in L_e$. Then ℓ is involved in at most one good pair (ℓ', ℓ) for some $\ell' \in L_e$ and in at most one good pair (ℓ, ℓ'') for some $\ell'' \in L_e$.*

Proof: Let $e = (u, v)$ be the rightmost edge with a line $\ell \in L_e$ that violates the desired property. Assume that the first part of the property is violated, that is, there are two different good pairs (ℓ'_1, ℓ) and (ℓ'_2, ℓ). If ℓ ends at vertex v, there clearly can be at most one of these good pairs because all good pairs have to be of case (i).

Now, suppose that ℓ also exists on edge $e' = (v, w)$ to the right of e on P. If both ℓ'_1 and ℓ'_2 existed on e', we would already have a counterexample on e'. Hence, at least one of the lines ends at v, that is, at least one of the good pairs results from inheritance at v. On the other hand, this can only be the case for one of the two pairs, suppose for (ℓ'_1, ℓ). Hence, there has to be another good pair (ℓ'_3, ℓ) on e', a contradiction to the choice of e. Symmetrically, we see that there cannot be two different good pairs (ℓ, ℓ''_1) and (ℓ, ℓ''_2). $\square$

Lemma 5 *If $e = (u, v) \in E_P$ is the last edge before a non-virtual line ℓ ends above the path, then there exists a line ℓ' on e that forms a good pair (ℓ', ℓ) with ℓ. Symmetrically, if e is the last edge before ℓ ends below the path, then there exists a line ℓ'' on e that forms a good pair (ℓ, ℓ'') with ℓ.*

Proof: We suppose that ℓ ends above the path; the other case is analogous. We consider the clockwise order of lines ending around v. If there is a non-virtual predecessor ℓ' of ℓ, then, by case (i) of the definition, (ℓ', ℓ) is a good pair. Otherwise, ℓ is the first line ending at v above the path. Then, virtual line t_e that we added is its predecessor, and (t_e, ℓ) is a good pair. $\square$

In what follows, we say that a solution or an algorithm *creates* a good pair (a, b) in a block crossing if the two lines a and b of the good pair are brought together in the right order by that block crossing; analogously, we speak of *breaking* a good pair if the two lines are neighbors in the right order before the block crossings and are no longer after the crossing.

Lemma 6 *There are only two possibilities for creating a good pair (a, b):*
(i) Lines a and b start at the same vertex consecutively in the right order.
(ii) A block crossing brings a and b together.
Similarly, there are only two possibilities for breaking a good pair:
(i) Lines a and b end at the same vertex.
(ii) A block crossing splitting a and b.

Proof: In the interior of the common subpath of a and b, the good pair (a, b) can only be created by block crossings because either a and b cross each other or lines between a and b cross a or b. Hence, (a, b) can only be created without a block crossing at the leftmost vertex of the common subpath, that is, when the last of the two lines, say a, starts at a vertex v. In this case a has to be the first line starting at v above P. This implies that, due to inheritance, there is a good

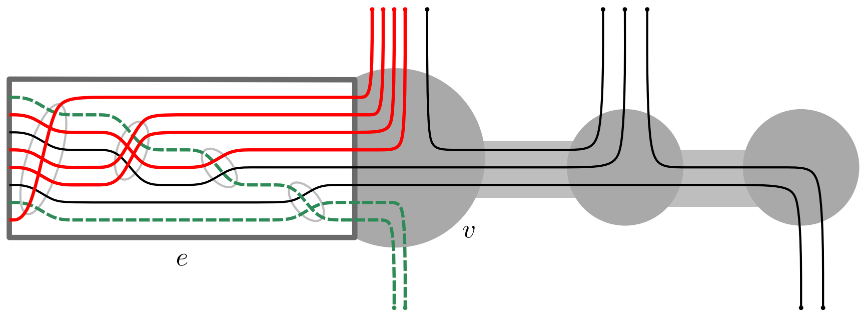


Figure 7: Ordering the lines on edge e in a step of the algorithm.

pair (c, b), where c is the last line ending at v above the path. It follows that the good pair (c, b), which is identical to (a, b), existed before v. Analogously, we get a contradiction if b is the first line starting at v below P. □

In case (i) of the lemma, we also say that (a, b) is an *initial good pair*.

It is easy to see that any solution, especially an optimal one, has to create all good pairs. As we identify good pairs resulting from inheritance with the original good pair (that is, resulting from case (i) of Definition 1), it suffices to consider good pairs resulting from two lines ending at the same vertex consecutively in clockwise order. As the lines must not cross in this vertex, they must be neighbors before this vertex is reached. We show that a crossing in which a good pair is broken cannot increase the number of good pairs at all.

Lemma 7 *In a block crossing, the number of good pairs increases by at most three. In a block crossing that breaks a good pair, the number of good pairs does not increase.*

Proof: We consider a block crossing on some edge that transforms the sequence

$$\pi = [\dots, a, b, \dots, c, d, \dots, e, f, \dots] \quad \text{into} \quad \pi' = [\dots, a, d, \dots, e, b, \dots, c, f, \dots],$$

that is, the blocks $b, \dots, c$ and $d, \dots, e$ are exchanged. The only new pairs of consecutive lines that π' contains compared to π are (a, d), (e, b), and (c, f). Even if these are all good pairs, the total number of good pairs increases only by three.

Now, suppose that the block crossing breaks a good pair. The only candidates are (a, b), (c, d), and (e, f). If (a, b) was a good pair, then the new pairs (a, d) and (e, b) cannot be good pairs because, on one edge, there can only be one good pair $(a, \cdot)$ and one good pair $(\cdot, b)$; see Lemma 4. Hence, only (c, f) can possibly be a new good pair. Since one good pair is destroyed and at most one good pair is created, the number of good pairs does not increase. The cases that the destroyed good pair is (c, d) or (e, f) are analogous. □

Using good pairs, we formulate our algorithm as follows; see Figure 7.

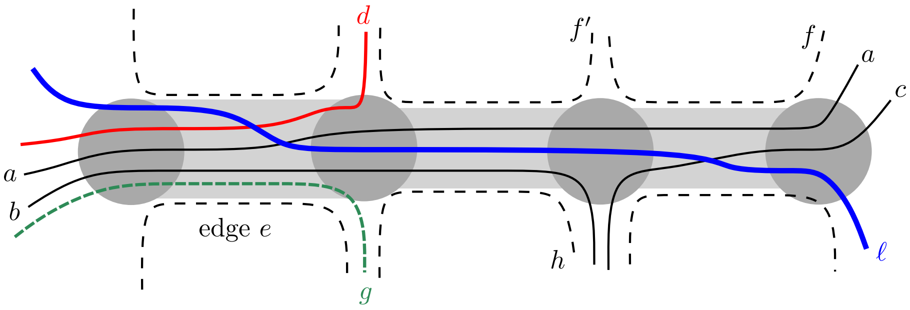


Figure 8: The (necessary) insertion of line ℓ forces breaking a good pair, that is, (a, b) $(\equiv (a, c))$, (d, a) $(\equiv (f, a) \equiv (f', a))$, or (b, g) $(\equiv (b, h))$, on edge e.

We traverse P from left to right. On an edge $e = (u, v) \in E_P$ of the path, there are *red lines* that end at v above the path, *green lines* that end at v below the path, and *black lines* that continue on the next edge. We bring the red lines in the right order to the top by moving them upwards. Doing so, we keep existing good pairs together. If a line is to be moved, we consider the lines below it consecutively. As long as the current line forms a good pair with the next line, we extend the block that will be moved. We stop at the first line that does not form a good pair with its successor. Then, we move the whole block of lines linked by good pairs in one block move to the top. Next, we bring the green lines in the right order to the bottom, again keeping existing good pairs together. There is an exception: sometimes one good pair on e cannot be kept together. If the moved block is a sequence of lines containing both red and green lines, and possibly some—but not all—black lines, then the block has to be broken; see block (d, a, b, g) in Figure 8.

Note that this can only happen in one move on an edge; there can only be one sequence containing both red and green lines because all red lines are part of a single sequence and all green lines are part of a single sequence due to case (i) of Definition 1. There are two cases when the sequence of good pairs has to be broken:

(i) A good pair in the sequence contains a black line and has been created by the algorithm previously. Then, we break the sequence at this good pair.
(ii) All pairs containing a black line are initial good pairs, that is, they have not been created by a crossing. Then, we break at the pair that ends last of these. When comparing the end of pairs we take inheritance into account, that is, a good pair ends only when the last of the pairs that are linked by inheritance ends.

After an edge has been processed, the lines ending above and below the path are on their respective sides in the right relative order. Hence, our algorithm produces a feasible solution. We show that the algorithm produces a 3-approximation for the number of block crossings. A key property is that our strategy for case (ii) is optimal.

Lemma 8 *Let* ALG *and* OPT *be the number of block crossings created by the algorithm and an optimal solution, respectively. Then,* $\mathrm{ALG} \leq 3\,\mathrm{OPT}$.

Proof: Block crossings that do not break a good pair always increase the number of good pairs. If we have a block crossing that breaks a good pair in a sequence as in case (i) then there has been a block crossing that created the good pair previously as a side effect: Each block crossing in our algorithm is caused by a good pair of two red or two green lines that end after the current edge. In such a crossing, there can be an additional good pair that is created unintentionally. Hence, we can say that the destroyed good pair did not exist previously and still have at least one new good pair per block crossing.

If we are in case (ii), that is, all good pairs in the sequence are initial good pairs (see Figure 8), then these good pairs also initially existed in the optimal solution. It is not possible to keep all these good pairs because the remaining black lines have to be somewhere between the block of red lines and the block of green lines. Hence, even the optimal solution has to break one of these good pairs, on this edge or previously.

Let $\overline{\mathrm{bc}}_{\mathrm{alg}}$ and $\overline{\mathrm{bc}}_{\mathrm{opt}}$ be the numbers of broken good pairs due to case (ii) in the algorithm and the optimal solution, respectively. In a crossing in which the algorithm breaks such a good pair the number of good pairs stays the same as one good pair is destroyed and another is created. On the other hand, in a crossing that breaks a good pair the number of good pairs can increase by at most two even in the optimal solution (actually, this number cannot increase at all; see Lemma 7). Let gp be the total number of good pairs in the instance according to the modified definition for monotone good pairs, and let $\mathrm{gp}_{\mathrm{init}}$ be the number of initial good pairs. Recall that, according to Definition 1, good pairs resulting from inheritance are not counted separately for gp as they are identified with another good pair. We get $\mathrm{gp} \geq \mathrm{ALG} - \overline{\mathrm{bc}}_{\mathrm{alg}} + \mathrm{gp}_{\mathrm{init}}$ and $\mathrm{gp} \leq 3 \cdot \mathrm{OPT} - \overline{\mathrm{bc}}_{\mathrm{opt}} + \mathrm{gp}_{\mathrm{init}}$. Hence, $\mathrm{ALG} \leq 3\,\mathrm{OPT} + (\overline{\mathrm{bc}}_{\mathrm{alg}} - \overline{\mathrm{bc}}_{\mathrm{opt}})$ combining both estimates.

To prove an approximation factor of 3, it remains to show that $\overline{\mathrm{bc}}_{\mathrm{alg}} \leq \overline{\mathrm{bc}}_{\mathrm{opt}}$. First, note that the edges where good pairs of case (ii) are destroyed are exactly the edges where such a sequence of initial good pairs exists; that is, the edges are independent of any algorithm or solution. We show that, among these edges, our strategy ensures that the smallest number of pairs is destroyed, and pairs that are destroyed once are reused as often as possible for breaking a sequence of initial good pairs.

To this end, let $e'_1, \dots, e'_{\overline{\mathrm{bc}}_{\mathrm{alg}}}$ be the sequence of edges, where the algorithm destroys a new good pair of type (ii), that is, an initial good pair that has never been destroyed before. We follow the sequence and argue that the optimal solution destroys a new pair for each of these edges. Otherwise, there is a pair e'_i, e'_j (with $i < j$) of edges in the sequence where the optimal solution uses the same good pair p on both edges. Let p' and p'' be the pairs used by the algorithm on e'_i and e'_j, respectively, for breaking a sequence of initial good pairs. As p' was preferred by the algorithm over p, we know that p' still exists on e'_j. As p' is in a sequence with p, the algorithm still uses p' on e'', a contradiction. $\square$

We can now conclude with the following theorem.

Theorem 4 *There is an $\mathcal{O}(|\mathcal{L}|(|\mathcal{L}|+n))$-time algorithm for finding a 3-approximation for BCM on instances where the underlying network is a path of length n with attached terminals.*

The runtime analysis is simple: In $\mathcal{O}(|\mathcal{L}|n)$ time, we can traverse the edges of the path from right to left and, on each edge, determine the good pairs of lines, resulting from lines ending together or from inheritance; for each good pair, we also store where the pair ends (taking inheritance into account), since we need this information in case (ii). In the following traversal from left to right, $\mathcal{O}(|\mathcal{L}|)$ time per edge is necessary for determining the order of lines on the left side of the edge, resulting from the combination of lines continuing from the edge to the left and lines joining the path. Additionally, there can be up to $|\mathcal{L}|$ block crossings in total over the whole path; each block crossing results from some line leaving the path. For each block crossing, we need linear time in the number of current lines for deciding which block move should be applied. Hence, $\mathcal{O}(|\mathcal{L}|^2)$ time in total suffices for all block crossings.

3.2 MBCM on a Path

The algorithm presented in the previous section does not guarantee monotonicity of the solution. It can, however, be turned into a 3-approximation algorithm for MBCM. To achieve this, we adjust the definition of inheritance of good pairs, as well as the step of destroying good pairs, and we sharpen the analysis.

We first modify our definition of inheritance of good pairs. We prevent inheritance in the situations in which keeping a pair of lines together at the end of an edge is not possible without either having an avoidable crossing in the following vertex or violating monotonicity. We concentrate on inheritance with lines ending above the path; the other case is symmetric.

Suppose we have a situation as shown in Figure 9 with a good pair (a_1, b). Line c must not cross b. On the other hand it has to be below a_2 near vertex v and separate a_2 and b there. Hence, bringing or keeping a_2 and b together is of no value, as they have to be separated in any solution. Therefore, we modify the definition of good pairs, so that pair (a_2, b) does not inherit from (a_1, b) in this situation; we say that line c is *inheritance-preventing* for (a_1, b).

Apart from the modified definition of good pairs, one part of our algorithm needs to be changed in order to ensure monotonicity of the solution. A block move including black lines could result in a forbidden crossing that violates monotonicity; see Figure 10. We focus on the case, where black lines are moved together with red lines to the top. This can only occur once per edge. The case that black lines are moved together with green lines to the bottom is symmetric. Let $b_0, b_1, \ldots, b_k$ be the sequence of good pairs from the bottommost red line $r = b_0$ on. If there is some line ℓ above the block $b_0, \ldots, b_k$ that must not be crossed by a line b_i of the block, then we have to break the sequence. We consider such a case and assume that i is minimal. Hence, we have to break one of the good pairs in $(r, b_1), (b_1, b_2), \ldots, (b_{i-1}, b_i)$. Similar to case (i) in the algorithm

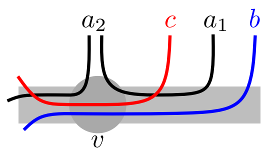


Figure 9: Line c prevents that (a_2, b) inherits from (a_1, b).

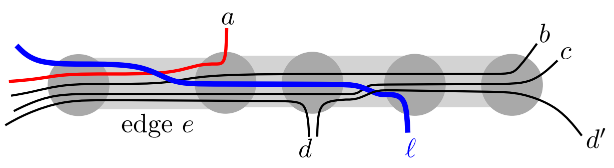


Figure 10: On edge e, there are the good pairs (a, b), (b, c), and (c, d) $(\equiv (c, d'))$. Before line a leaves the path, one of these good pairs must be destroyed—we chose (b, c)—otherwise line ℓ would have to cross line d twice, violating monotonicity.

for BCM, we break a pair of this sequence that is not initial. If all the pairs are initial (case (ii)), we choose the pair (b_{j-1}, b_j) with $j \leq i$ minimal such that the end vertex of b_j is below the path, and break the sequence there. Note that line ℓ must end below the path, otherwise it would prevent inheritance of at least one of the good pairs in the sequence. Hence, also b_i ends below the path, and b_j is well-defined.

It is easy to see that our modified algorithm still produces a feasible ordering. We now show that the solution is also monotone.

Lemma 9 *The modified algorithm produces an ordering with monotone block crossings.*

Proof: We show that any pair of lines that cross in a block crossing is in the wrong order before the crossing. Monotonicity of the whole solution then follows. We consider moves where blocks of lines are brought to the top; the other case is symmetric.

Suppose that a red line r is brought to the top. As all red lines that have to leave above r have been brought to the top before, r crosses only lines that leave below it, that is, lines that have to be crossed by r. If a black line ℓ is brought to the top, then it is moved together in a block that contains a sequence of good pairs from the bottommost red line r' to ℓ. Suppose that ℓ crosses a line c that must not be crossed by ℓ. Line c cannot be red because all red lines that are not in the block that is moved at the moment have been brought to the top before. It follows that r' has to cross c. Hence, we can find a good pair (a, b) in the sequence from r' to ℓ such that a has to cross c but b must not cross c. In this case, the algorithm will break at least one good pair between r' and b. It follows that c does not cross ℓ, a contradiction. □

Lemma 10 *Let* $\mathrm{ALG}_{\mathrm{mon}}$ *be the number of block crossings created by the algorithm for MBCM and let* $\mathrm{OPT}_{\mathrm{mon}}$ *be the number of block crossings of an optimal solution for MBCM. It holds that* $\mathrm{ALG}_{\mathrm{mon}} \leq 3\,\mathrm{OPT}_{\mathrm{mon}}$.

Proof: As for the non-monotone case, all block crossings that our algorithm introduces increase the number of good pairs, except when the algorithm breaks a sequence of initial good pairs in case (ii). Again, also the optimal solution has to have crossings where such sequences are broken. In such a crossing of case (ii), the two lines of the destroyed pair lose their partner. Hence, there is only one good pair after the crossing, and the total number of good pairs does not change at all; compare Lemma 7.

Hence, $\mathrm{gp} \geq \mathrm{ALG}_{\mathrm{mon}} - \overline{\mathrm{bc}}_{\mathrm{alg}} + \mathrm{gp}_{\mathrm{init}}$ and $\mathrm{gp} \leq 3\left(\mathrm{OPT}_{\mathrm{mon}} - \overline{\mathrm{bc}}_{\mathrm{opt}}\right) + \mathrm{gp}_{\mathrm{init}}$. Combining both estimates, we get $\mathrm{ALG}_{\mathrm{mon}} \leq 3\,\mathrm{OPT}_{\mathrm{mon}} + (\overline{\mathrm{bc}}_{\mathrm{alg}} - 3\overline{\mathrm{bc}}_{\mathrm{opt}})$. Let $\overline{\mathrm{bc}}_{\mathrm{alg,top}}$ be the number of splits for case (ii) where the block move brings lines to the top, and let $\overline{\mathrm{bc}}_{\mathrm{alg,bot}}$ be the number of such splits where the move brings lines to the bottom. Clearly, $\overline{\mathrm{bc}}_{\mathrm{alg}} = \overline{\mathrm{bc}}_{\mathrm{alg,top}} + \overline{\mathrm{bc}}_{\mathrm{alg,bot}}$. We get

$$\begin{aligned}\mathrm{ALG}_{\mathrm{mon}} &\leq 3 \cdot \mathrm{OPT}_{\mathrm{mon}} + (\overline{\mathrm{bc}}_{\mathrm{alg}} - 3 \cdot \overline{\mathrm{bc}}_{\mathrm{opt}})\\ &\leq 3 \cdot \mathrm{OPT}_{\mathrm{mon}} + (\overline{\mathrm{bc}}_{\mathrm{alg,top}} - \overline{\mathrm{bc}}_{\mathrm{opt}}) + (\overline{\mathrm{bc}}_{\mathrm{alg,bot}} - \overline{\mathrm{bc}}_{\mathrm{opt}}).\end{aligned}$$

To complete the proof, we show that $\overline{\mathrm{bc}}_{\mathrm{alg,top}} \leq \overline{\mathrm{bc}}_{\mathrm{opt}}$. Symmetry will then yield that $\overline{\mathrm{bc}}_{\mathrm{alg,bot}} \leq \overline{\mathrm{bc}}_{\mathrm{opt}}$.

Let $e'_1, \ldots, e'_{\overline{\mathrm{bc}}_{\mathrm{alg,top}}}$ be the sequence of edges where the algorithm uses a new good pair as a breakpoint for a sequence of type (ii) when lines leave to the top, that is, a good pair that has not been destroyed before. Again, we argue that even the optimal solution has to use a different breakpoint pair for each of these edges. Otherwise, there would be a pair e', e'' of edges in this sequence where the optimal solution uses the same good pair p on both edges. Let p' and p'' be the two good pairs used by the algorithm on e' and e'', respectively. Let $p' = (\ell', \ell'')$. We know that ℓ' leaves the path to the top and ℓ'' leaves to the bottom as described in case (ii). Because all lines in the orders on e' and e'' stay parallel, we know that lines above ℓ' leave to the top, and lines below ℓ'' leave to the bottom. In particular, p' still exists on e'', as p stays parallel and also still exists.

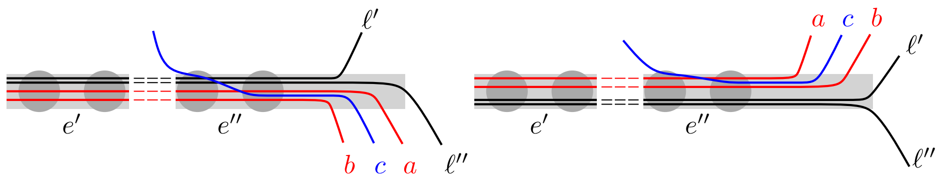
As in the description of the algorithm, let a and b be lines such that (a, b) is the topmost good pair in the sequence for which a line c exists on e'' that crosses a but not b. If (a, b) is below p' (see Figure 11a), then the algorithm would reuse p' instead of the new pair p'' since (a, b) is in a sequence below p; hence, also p' is in the sequence and above (a, b).

Now suppose that (a, b) is above p'; see Figure 11b. Pair (a, b) is created by inheritance because c ends between a and b. As both a and b end above the path, separated from the bottom side of the path by p', this inheritance takes place at a vertex. At this vertex, a is the last line to end above the path. But in this case, c prevents the inheritance of the good pair (a, b) because c crosses only a, a contradiction to (a, b) being a good pair. □

We can now conclude with the following theorem.

Theorem 5 *Given an instance of MBCM whose skeleton is a path of length n, a 3-approximation can be computed in $\mathcal{O}(|\mathcal{L}|(|\mathcal{L}| + n))$ time.*

Proof: Given Lemmas 9 and 10, it remains to analyze the running time. In order to ensure monotonicity of the solution, our algorithm precomputes for



(a) (a, b) is below (l', l''). (b) (a, b) is above (l', l'').

Figure 11: Different cases based on the pair (a, b).

each pair of lines whether the lines must or must not cross. This is done in $\mathcal{O}(|\mathcal{L}|^2 + n)$ time in total by following the path, keeping track of the lines in the order resulting from joining the path, and marking all crossings when a line leaves the path. We then use this information to check the inheritance of a pair of lines. For any other line that is currently active, we check whether it is inheritance-preventing. Overall, this needs to be done only $\mathcal{O}(n)$ times. Hence, the algorithm takes $\mathcal{O}(|\mathcal{L}|(|\mathcal{L}| + n))$ total time. $\square$

4 Block Crossings on Trees

In the following we focus on instances of BCM and MBCM where the underlying network is a tree. We first present an algorithm that yields a worst-case optimal bound on the number of block crossings. Then, we consider the special class of *upward trees* which have an additional constraint on the lines; for upward trees we develop a 6-approximation for BCM and MBCM.

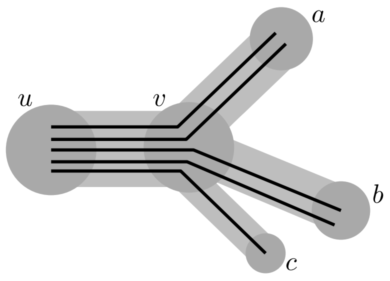
4.1 General Trees

Theorem 6 *Given an embedded tree $T = (V, E)$ of n vertices and a set $\mathcal{L}$ of lines on T, one can order the lines with at most* $2|\mathcal{L}| - 3$ monotone *block crossings in $\mathcal{O}(|\mathcal{L}|(|\mathcal{L}| + n))$ time.*

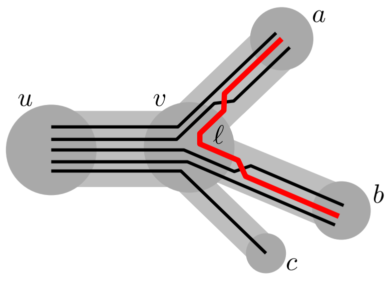
Proof: We root the given tree T at a leaf r and direct the edges away from the root. Our algorithm processes the edges of T in breadth-first order, starting from an edge incident to the root. In every step of the algorithm, we maintain the following invariant: The lines in L_e are in the *correct* order, that is, they do not need to cross on yet unprocessed edges of T. We maintain the invariant by crossing every pair of lines that needs to cross as soon as we see it.

The first step of the algorithm treats an edge $e = (r, w)$ incident to the root r. Since r is a terminal, there is only one line ℓ on edge e. We insert ℓ into the (currently empty) orders on the edges of ℓ, which clearly maintains the invariant. Next we proceed with the edges incident to w.

Consider a step of the algorithm processing an edge $e = (u, v)$. The lines in L_e are already in the correct order. We consider all unprocessed edges $(v, a), (v, b), \ldots$ incident to v in clockwise order and build the correct orders for



(a) Situation before inserting ℓ.



(b) Line ℓ is inserted on edges (v, a) and (v, b).

Figure 12: Inserting a new line ℓ (bold red) into the current order on edges (v, a) and (v, b).

them. The relative order of lines that also pass through (u, v) is kept unchanged on the new edges. For all lines passing through v that have not been treated before, we apply the following insertion procedure; see Figure 12.

Without loss of generality, a line ℓ passes through (v, a) and (v, b) so that a precedes b in the clockwise order around v. The line is inserted at the last position of the current orders $\pi^0(v, a), \ldots, \pi^{t(v,a)}(v, a)$ and at the first positions of the current orders $\pi^0(v, b), \ldots, \pi^{t(v,b)}(v, b)$. This guarantees that we do not get vertex crossings with the lines passing through (v, a) or (v, b). Then, as the lines $L_{va} \setminus \ell$ are in the correct order close to vertex a, there exists a correct position of ℓ in the order. We insert ℓ at the position using a single block crossing, thus increasing $t(v, a)$ by one. This crossing is the last one on edge (v, a) going from v to a. Similarly, ℓ is inserted into L_{vb}.

This way we insert all the new lines and construct the correct orders on $(v, a), (v, b), \ldots$ maintaining the invariant. We have to be careful with the insertion order of the lines that do not have to cross. As we know the right relative order for a pair of such lines, we can make sure that the one that has to be innermost at vertex v is inserted first. Similarly, by considering the clockwise order of edges around v, we know the right order of line insertions such that there are no avoidable vertex crossings. Once all new lines have been inserted, we proceed by processing the edges incident to v (except (u, v)).

We now prove the correctness of the algorithm. For each newly inserted line, we create at most two monotone block crossings. The first line that we insert into the empty orders cannot create a crossing, and the second line crosses the first line at most once. Hence, we need at most $2|\mathcal{L}| - 3$ monotone block crossings in total. Suppose that monotonicity is violated, that is, there is a pair of lines that crosses twice. Then, the crossings must have been introduced when inserting the second of those lines on two edges incident to some vertex. This, however, cannot happen as at any vertex the two lines are inserted in the right order by the above construction. Hence, the block crossings of the solution are monotone.

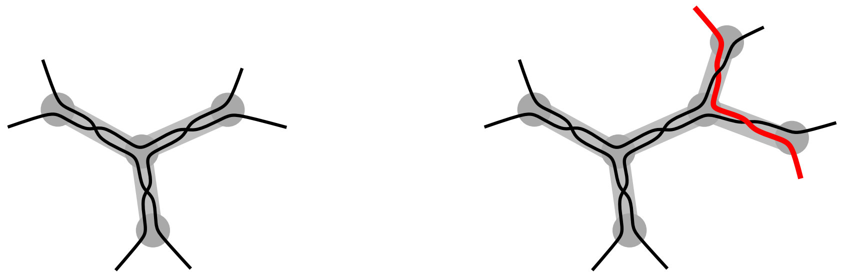

(a) Instance for $|\mathcal{L}| = 3$.
(b) The instance for $|\mathcal{L}| = 4$ is created by adding a line (bold red) to the instance for $|\mathcal{L}| = 3$.

Figure 13: Trees with $2|\mathcal{L}| - 3$ necessary crossings. By adding more lines using the same construction by which the instance for $|\mathcal{L}| = 4$ was created from the one for $|\mathcal{L}| = 3$, instances with an arbitrary number of lines can be created.

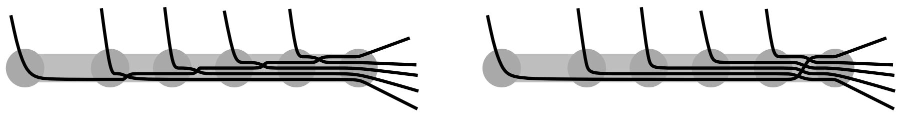

(a) Started at the leftmost edge, the algorithm, produces 4 crossings.
(b) In an optimum solution one block crossing suffices.

Figure 14: Worst case example for our algorithm for trees shown for five edges. It can easily be extended to an arbitrary number of edges (and crossings).

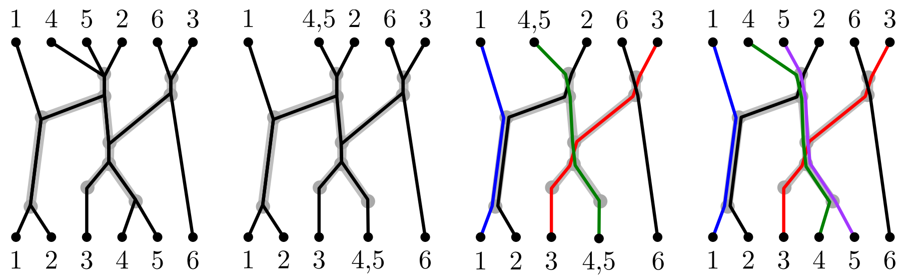
The algorithm can be implemented to run in $\mathcal{O}(|\mathcal{L}|(|\mathcal{L}| + n))$ time: we need $\mathcal{O}(|\mathcal{L}| \cdot n)$ time to traverse the tree and determine the insertion sequence of lines and $\mathcal{O}(|\mathcal{L}|^2)$ time to construct the correct orders. $\square$

Next we show that the upper bound that our algorithm yields is tight. Consider the instance shown in Figure 13. The new bold red line in Figure 13b is inserted so that it crosses two existing paths. The example can easily be extended to instances of arbitrary size in which $2|\mathcal{L}| - 3$ block crossings are necessary in any solution.

Unfortunately, there are also examples in which our algorithm creates $|\mathcal{L}| - 1$ crossings while a single block crossing suffices; see Figure 14 for $|\mathcal{L}| = 5$. The extension of the example to any number of lines is straightforward. This shows that the algorithm does not yield a constant-factor approximation.

4.2 Upward Trees

Here we introduce an additional constraint on the lines, which allows to approximate the minimum number of block crossings. Consider a tree T with a set of lines $\mathcal{L}$. The instance $(T, \mathcal{L})$ is an *upward* tree if T has a planar upward drawing—respecting the given embedding—in which all paths are monotone



(a) Input with pairwise crossings. (b) Simplification by merging 4 and 5. (c) Line ordering on simplified instance. (d) Simplification undone for solution.

Figure 15: Three steps of the algorithm for upward trees. The instance is drawn in the style of a permutation with lines numbered from 1 to 6.

in vertical direction, and all path sources are on the same height as well as all path sinks; see Figure 15a. Bekos et al. [4] already considered such trees (under the name "left-to-right trees") for the metro-line crossing minimization problem. Note that a graph whose skeleton is a path is not necessarily an upward tree, as the attached terminals obstruct a planar upward drawing.

Our algorithm consists of three steps. First, we perform a simplification step that removes some lines. Second, we use the algorithm for trees presented in the previous section on the simplified instance. Finally, we re-insert the removed lines into the constructed order without introducing new block crossings; see Figure 15 for an illustration of the steps of the algorithm. We first consider MBCM and start by analyzing the upward embedding.

Given an upward drawing of T, we read a permutation π produced by the terminals on the top similar to the case of a single edge; we assume that the terminals produce the identity permutation on the bottom. Similar to the single-edge case, the goal is to sort π by a shortest sequence of block moves. Edges of T restrict some block moves on π; for example, blocks $[1, 4]$ and $[5]$ in Figure 15a cannot be exchanged because there is no suitable edge with all these lines. However, we can use the lower bound for block crossings on a single edge; see Section 2: For sorting a *simple* permutation π, at least $\lceil \mathrm{bp}(\pi)/3 \rceil = \lceil (|\mathcal{L}| + 1)/3 \rceil$ block moves are necessary. We stress that simplicity of π is crucial here because the algorithm for trees may create up to $2|\mathcal{L}| - 3$ crossings. To get an approximation, we show how to simplify a tree.

Consider two non-intersecting paths a and b that are adjacent in both permutations and share a common edge. We prove that one of these paths can be removed without changing the optimal number of monotone block crossings. First, if any other line c crosses a then it also crosses b in any solution (i). This is implied by the monotonicity of the block crossings, by planarity, and by the y-monotonicity of the drawing. Second, if c crosses both a and b then all three paths share a common edge (ii); otherwise, there would be a cycle in the graph due to planarity. Hence, given any solution for the paths $\mathcal{L} \setminus \{b\}$, we can construct a solution for $\mathcal{L}$ by inserting b parallel to a without any new block crossing. To insert b, we must first move all block crossings involving a to the common subpath with b. This is possible due to observation (ii). Finally, we can place b parallel to a.

To get a 6-approximation for an upward tree, we first remove lines until the tree is simplified. Then we apply the insertion algorithm presented above, and finally re-insert the lines removed in the first step. The number of block crossings is at most $2|\mathcal{L}'|$, where $\mathcal{L}'$ is the set of lines of the simplified instance. As an optimal solution has at least $|\mathcal{L}'|/3$ block crossings for this simple instance, and re-inserting lines does not create new block crossings, we get the following result.

Theorem 7 *On embedded upward trees, there is an 6-approximation algorithm for MBCM. Given an upward tree with n vertices and a set $\mathcal{L}$ of lines on that tree, the algorithm runs in $\mathcal{O}(|\mathcal{L}|(|\mathcal{L}| + n))$ time.*

Proof: It remains to analyze the running time of the algorithm. Simplification of the tree can be implemented in linear time by considering the top and bottom permutations of the lines. Hence, the overall running time is $\mathcal{O}(|\mathcal{L}|(|\mathcal{L}| + n))$ by Theorem 6. □

If we consider BCM instead of MBCM, we face the problem that we do not know whether every solution for the simplified instance can be transformed into a solution for the input instance without additional crossings. However, we observe that our algorithms always finds monotone solutions for simplified instances and, hence, they can be transformed back. Furthermore, dropping lines can never increase the necessary number of block crossings. Hence, also for BCM we have the lower bound of $\lceil(|\mathcal{L}'| + 1)/3\rceil$ block crossings. Summing up, we get a 6-approximation for BCM by using the same algorithm.

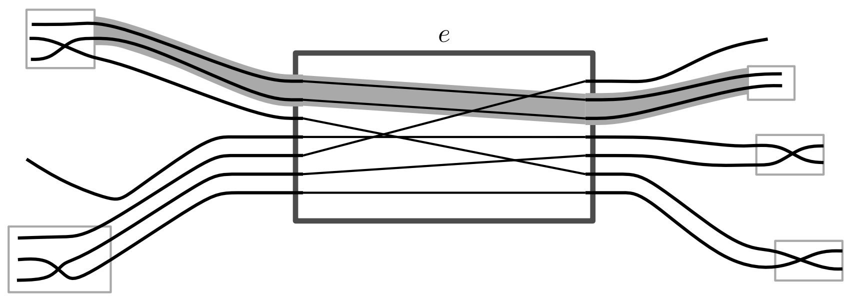
Corollary 2 *On embedded upward trees, there is an 6-approximation algorithm for BCM. Given an upward tree with n vertices and a set $\mathcal{L}$ of lines on that tree, the algorithm runs in $\mathcal{O}(|\mathcal{L}|(|\mathcal{L}| + n))$ time.*

5 Block Crossings on General Graphs

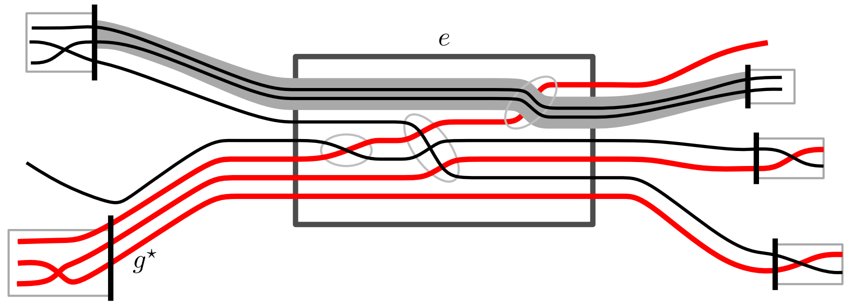
In this section, we consider general graphs. We suggest an algorithm that achieves an upper bound on the number of block crossings and show that it is asymptotically worst-case optimal. Our algorithm uses only monotone block moves, that is, each pair of lines crosses at most once. The algorithm works on any embedded graph; it just needs the circular order of incident edges around each vertex.

The idea of the algorithm is as follows. We process the edges in an arbitrary order. When we treat an edge, we sort the lines that traverse it. Among these lines, we create a crossing between a pair if and only if this is the first time that we consider this pair.

The crucial part of our algorithm is sorting the lines on an edge. We do this in four steps: First, we build line orders on both sides of the edge and identify



(a) Earlier edge cuts (at the gray boxes) define groups with respect to e. The two lines with gray shading are merged as they are in the same group on both sides.



(b) Sorting by insertion into the largest group $g^\star$ (bold red). The merged lines always stay together, in particular if their block crosses other lines.

Figure 16: Sorting the lines on an edge e in a step of our algorithm.

groups of lines. Then, we merge some lines if necessary. Next, we select the largest group of lines that stay parallel on the current edge. The sorting of lines then simply consists of inserting all other lines into the largest group. Finally, we undo the merging step. In the following paragraphs, we describe these four steps in detail.

An invariant. Our algorithm maintains the following invariant during the iteration over all edges. As soon as an edge has been treated, its line order is fixed. Each pair (ℓ, ℓ') of lines that necessarily has to cross in the instance crosses on the first common edge of ℓ and ℓ' that is treated by the algorithm; on any other edge the pair (ℓ, ℓ') is crossing-free. The relative order of ℓ and ℓ' on such an edge is the same as the one on the closer end of the edge containing the crossing. Each pair of lines that must not cross is crossing-free on all edges and their relative order on the processed edges is the one that is implied by a crossing-free solution.

Suppose we currently deal with edge e and want to sort L_e. Due to the path intersection property, the edge set used by the lines in L_e forms a tree on each side of e; see Figure 16.

Grouping lines. We cut these trees at the edges that have already been processed. Then, each line on e starts at a leaf on one side and ends at a leaf on the other side. Note that multiple lines can start or end at the same leaf representing an edge that has previously been treated by the algorithm.

From the tree structure and the orders on the edges processed previously, we get two orders of the lines, one on each side of e. Consider *groups of lines* that start or end at a common leaf of the tree (such as the group of red lines in Figure 16). All lines of a group have been seen on a common edge, and, hence, have been sorted. Therefore, lines of the same group form a consecutive subsequence on one side of e and have the same relative order on the other side.

Merging lines. Let g be a group of lines on the left side of e, and let g' be a group of lines on the right side of e. Let $\mathcal{L}'$ be the (possibly empty) set of lines starting in g on the left and ending in g' on the right. Suppose that $\mathcal{L}'$ consists of at least two lines. As the lines of g as well as the lines of g' stay parallel on e, $\mathcal{L}'$ must form a consecutive subsequence (in the same order) on both sides. Now, we *merge* $\mathcal{L}'$ into one representative, that is, we remove all lines of $\mathcal{L}'$ and replace them by a single line that is in the position of the lines of $\mathcal{L}'$ in the sequences on both sides of e. Once we find a solution, we replace the representative by the sequence. This does not introduce new block crossings as we will see. Consider a crossing that involves the representative of $\mathcal{L}'$, that is, the representative is part of one of the moved blocks. After replacing the representative, the sequence $\mathcal{L}'$ of parallel lines is completely contained in the same block. Furthermore, the lines of $\mathcal{L}'$ do not cross each other on edge e since they are part of the same group g. Hence, we do not need additional block crossings. We apply this merging step to all pairs of groups on the left and right end of e.

Sorting by insertion into the largest group. Now, we identify a group $g^\star$ with the largest number of lines after merging, and insert all remaining lines into $g^\star$ one by one. Clearly, each insertion requires at most one block crossing; in Figure 16 we need three block crossings to insert the lines into the largest (red) group $g^\star$. After computing the crossings, we undo the merging step and obtain a solution for edge e.

Maintaining the invariant. Note that by building the groups and the line orders on both sides of e, we ensure that the relative order of each pair of lines is consistent with the line orders of edges treated by the algorithm previously. If a pair of lines has to cross, but did not cross so far—because we did not treat a common edge before—, the line orders on both ends will be different, resulting in a crossing on edge e. However, if the edges did already cross on another edge, the line orders on both sides of e will be the same and, hence, the lines stay parallel on e.

Theorem 8 *The algorithm described above computes a feasible solution for MBCM on an instance $(G = (V, E), \mathcal{L})$ in $\mathcal{O}(|E|^2|\mathcal{L}|)$ time. The resulting number of block crossings is at most $|\mathcal{L}|\sqrt{|E'|}$, where $E' \subseteq E$ is the set of edges with at least two lines.*

Proof: First, it is easy to see that no avoidable crossings are created, due to the path intersection property. Additionally, we treat all edges with at least two lines, which ensures that all unavoidable crossings will be placed. Hence, we get a feasible solution using only monotone crossings.

Our algorithm sorts the lines on an edge in $\mathcal{O}(|\mathcal{L}||E|)$ time. We can build the tree structure and find orders and groups by following all lines until we find a terminal or an edge that has been processed before in $\mathcal{O}(|\mathcal{L}||E|)$ time. Merging lines and finding the largest group needs $\mathcal{O}(|\mathcal{L}|)$ time; sorting by insertion into this group and undoing the merging can be done in $\mathcal{O}(|\mathcal{L}|^2)$ time. Note that $|\mathcal{L}| \leq |E|$ due to the path terminal property.

Purely for the purpose of analyzing the total number of block crossings, we maintain an *information table* T with $|\mathcal{L}|^2$ entries. Initially, all entries are empty. After our algorithm has processed an edge e, we set $T[\ell, \ell'] = e$ for each pair (ℓ, ℓ') of lines that we see together for the first time. The main idea is that with b_e block crossings on edge e, we fill at least b_e^2 new entries of T. This ultimately yields the desired upper bound of $|\mathcal{L}|\sqrt{|E|}$ for the total number of block crossings.

More precisely, let the *information gain* $I(e)$ be the number of pairs of (not necessarily distinct) lines ℓ, ℓ' that we see together on a common edge e for the first time. Clearly, $\sum_{e \in E} I(e) \leq |\mathcal{L}|^2$. Suppose that $b_e^2 \leq I(e)$ for each edge e. Then, $\sum_{e \in E} b_e^2 \leq \sum_{e \in E} I(e) \leq |\mathcal{L}|^2$. Using the Cauchy-Schwarz inequality $|\langle x, y \rangle| \leq \sqrt{\langle x, x \rangle \cdot \langle y, y \rangle}$ with $x = (1)_{e \in E'}$ the all-ones vector and $y = (b_e)_{e \in E'}$ the vector of block crossing numbers, we see that the total number of block crossings is

$$\sum_{e \in E'} b_e = |\langle x, y \rangle| \leq \sqrt{\langle x, x \rangle \cdot \langle y, y \rangle} = \sqrt{|E'| \sum_{e \in E'} b_e^2} \leq \sqrt{|E'| \cdot |\mathcal{L}|^2} = |\mathcal{L}|\sqrt{|E'|}.$$

It remains to show that $b_e^2 \leq I(e)$ for every edge e. We analyze the lines after the merging step. Consider the groups on both sides of e; we number the groups on the left side $\mathfrak{L}_1, \ldots, \mathfrak{L}_n$ and the groups on the right side $\mathfrak{R}_1, \ldots, \mathfrak{R}_m$. For $1 \leq i \leq n$ let $l_i = |\mathfrak{L}_i|$, and for $1 \leq j \leq m$ let $r_j = |\mathfrak{R}_j|$. Without loss of generality, we can assume that $\mathfrak{L}_1$ is the largest of the $n + m$ groups. Hence, the algorithm inserts all remaining lines into $\mathfrak{L}_1$.

Then, $b_e \leq |L_e| - l_1$. Let s_{ij} be the number of lines that are in group $\mathfrak{L}_i$ on the left side and in group $\mathfrak{R}_j$ on the right side of e. Note that $s_{ij} \in \{0, 1\}$, otherwise we could still merge lines. Then $l_i = \sum_{j=1}^{m} s_{ij}$, $r_j = \sum_{i=1}^{n} s_{ij}$, $s := |L_e| = \sum_{i=1}^{n} \sum_{j=1}^{m} s_{ij}$, and $b_e \leq s - l_1$. In terms of this notation, the information gain is $I(e) = s^2 - \sum_{i=1}^{n} l_i^2 - \sum_{j=1}^{m} r_j^2 + \sum_{i=1}^{n} \sum_{j=1}^{m} s_{ij}^2$, which can be seen as follows. From the total number s^2 of pairs of lines on the edge, we have to subtract all pairs of lines that are in the same group on the left or on the right side of the edge; we must be careful not to subtract pairs that are in the same group on the left and on the right side twice. By applying the following Lemma 11 to the values s_{ij} (for $1 \leq i \leq n$ and $1 \leq j \leq m$), we get $b_e^2 \leq I(e)$.

To complete the proof, note that the unmerging step neither decreases $I(e)$ nor does it change b_e. □

Lemma 11 *For $1 \leq i \leq n$ and $1 \leq j \leq m$, let $s_{ij} \in \{0,1\}$. Let $l_i = \sum_{j=1}^m s_{ij}$ for $1 \leq i \leq n$ and let $r_j = \sum_{i=1}^n s_{ij}$ for $1 \leq j \leq m$ such that $l_1 \geq l_i$ for $1 \leq i \leq n$ and $l_1 \geq r_j$ for $1 \leq j \leq m$. Let $s = \sum_{i=1}^n \sum_{j=1}^m s_{ij}$, $b = s - l_1$, and $I = s^2 - \sum_{i=1}^n l_i^2 - \sum_{j=1}^m r_j^2 + \sum_{i=1}^n \sum_{j=1}^m s_{ij}^2$. Then, $b^2 \leq I$.*

Proof: It is easy to see that, for any $1 \leq i \leq n, 1 \leq j \leq m$, it holds that $s_{ij}(s_{ij} - s_{1j}) \geq 0$ as $s_{ij} \in \{0,1\}$. Using this property in the last line of the following sequence of (in-)equalities, we get

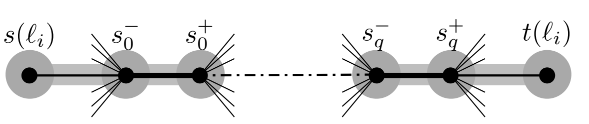
$$\begin{aligned}
I - b^2 &= \left(s^2 - \sum_{i=1}^n l_i^2 - \sum_{j=1}^m r_j^2 + \sum_{i=1}^n \sum_{j=1}^m s_{ij}^2\right) - \left(s^2 - 2sl_1 + l_1^2\right) \\
&= \sum_{i=1}^n \sum_{j=1}^m s_{ij}^2 + 2l_1(s - l_1) - \sum_{i=2}^n l_i^2 - \sum_{j=1}^m r_j \sum_{i=1}^n s_{ij} \\
&= \sum_{i=1}^n \sum_{j=1}^m s_{ij}^2 + 2l_1 \sum_{i=2}^n \sum_{j=1}^m s_{ij} - \sum_{i=2}^n l_i \sum_{j=1}^m s_{ij} - \sum_{i=1}^n \sum_{j=1}^m s_{ij} r_j \\
&= \sum_{i=2}^n \sum_{j=1}^m s_{ij}\left(s_{ij} + 2l_1 - l_i - r_j\right) - \sum_{j=1}^m s_{1j}\left(r_j - s_{1j}\right) \\
&= \sum_{i=2}^n \sum_{j=1}^m s_{ij}\left(s_{ij} + \underbrace{2l_1 - l_i - r_j}_{\geq 0}\right) - \sum_{j=1}^m s_{1j} \sum_{i=2}^n s_{ij} \\
&\geq \sum_{i=2}^n \sum_{j=1}^m \underbrace{s_{ij}\left(s_{ij} - s_{1j}\right)}_{\geq 0} \geq 0.
\end{aligned}$$

□

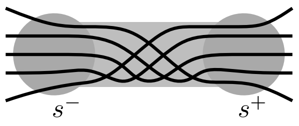
Next we show that the upper bound on the number of block crossings that our algorithm achieves is asymptotically tight. To this end, we use the existence of Steiner systems for building (nonplanar) worst-case examples of arbitrary size in which many block crossings are necessary.

Theorem 9 *For any prime power q, there exists a graph $G_q = (V_q, E_q)$ with $\Theta(q^2)$ vertices and a set $\mathcal{L}_q$ of lines so that $\Omega\big(|\mathcal{L}_q|\sqrt{|E'_q|}\big)$ block crossings are necessary in any solution, where $E'_q \subseteq E_q$ is the set of edges with at least two lines.*

Proof: Let q be a prime power. From the area of projective planes it is known that an $S(q^2+q+1, q+1, 2)$-Steiner system exists [27], that is, there is a set $\mathcal{S}$ of q^2+q+1 elements with subsets $S_1, S_2, \ldots, S_{q^2+q+1}$ of size $q+1$ each such that any pair of elements $s, t \in \mathcal{S}$ appears together in exactly one set S_i.



(a) Line ℓ_i is routed through the edges representing $s_0, \ldots, s_q$.



(b) The order of the lines is reverted between s^- and s^+.

Figure 17: Construction of the worst-case example.

We build graph $G_q = (V_q, E_q)$ by first adding vertices s^-, s^+ and an edge (s^-, s^+) for any $s \in \mathcal{S}$. These edges will be the only ones with multiple lines on them, that is, they form E'_q. Additionally, we add an edge (s^+, t^-) for each pair $s, t \in \mathcal{S}$. Next, we build a line ℓ_i for each set S_i as follows. We choose an arbitrary order $s_0, s_1, s_2, \ldots, s_q$ of the elements of S_i; then, we introduce extra terminals $s(\ell_i)$ and $t(\ell_i)$ in which the new line $\ell_i = \bigl(s(\ell_i), s_0^-, s_0^+, s_1^-, s_1^+, \ldots, s_q^-, s_q^+, t(\ell_i)\bigr)$ starts and ends, respectively; see Figure 17a.

As any pair of lines shares exactly one edge, the path intersection property holds. For each $s \in \mathcal{S}$, we order the edges around vertices s^- and s^+ in the embedding so that all $q + 1$ lines on the edge representing s have to cross. This is accomplished by using the reverse order of lines between s^- and s^+; see Figure 17b. Then at least $q/3$ block crossings are necessary on each edge (using the observation in Section 2), and, hence, $(q^2 + q + 1)q/3 = \Theta(q^3)$ block crossings in total. On the other hand, $|\mathcal{L}|\sqrt{|E'|} = (q^2 + q + 1)\sqrt{q^2 + q + 1} = \Theta(q^3)$. $\square$

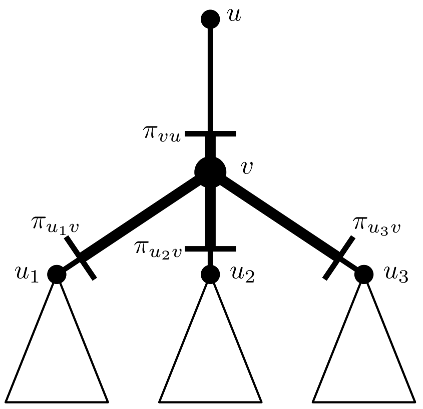
6 Instances with Bounded Maximum Degree and Edge Multiplicity

We now introduce two additional restrictions for (M)BCM. First, we consider instances in which the *maximum degree* Δ of a station is bounded by some constant. Second, we assume that on any edge e, there is at most a constant number c of lines, that is, $|L_e| \leq c$; we say that c is the maximum *edge multiplicity*. For metro maps both restrictions are realistic: In the popular octilinear drawing style, the maximum possible degree is 8. Furthermore, even in huge metro networks, edges that are served by more than 10 lines are unlikely to occur, as Nöllenburg [22] pointed out.

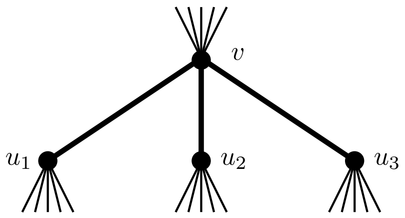
We first show that the restricted problem variants of both BCM and MBCM can be solved in polynomial time if the underlying network is a tree. On the other hand, we prove that the restricted variants are NP-hard on general graphs.

6.1 Restricted (M)BCM on Trees

We develop a dynamic program that solves (M)BCM on instances whose underlying network is a tree. First, we root tree $T = (V, E)$ at some arbitrary leaf r. Let $v \in V \setminus \{r\}$, and let u be the parent vertex of v. We say that a line *contributes* to



(a) Subtree $T[v]$; the remaining instance, which is bounded by orders π_{vu}, π_{u_1v}, π_{u_2v}, and π_{u_3v} is drawn bold.



(b) Remaining instance of constant size for the subtree; permutations are replaced by edges leaving in the right order.

Figure 18: Computation of $\mathrm{bc}[v, \pi_{vu}]$ for a subtree $T[v]$ in the dynamic program.

subtree $T[v]$ if at least one of its terminals is a vertex of $T[v]$; the line *leaves* the subtree if one of its terminals is in $T[v]$ and the other one is outside. Any line that leaves subtree $T[v]$ passes through edge $e = (u, v)$. If we fix the order π_{vu} of lines in L_e when leaving v on edge e, an optimum solution for $T[v]$ is independent of an optimum solution for the remaining graph; in other words, we can combine any optimum solution for $T[v]$ resulting in order π_{vu} with any optimum solution for the remaining graph resulting in the same order π_{vu}. Let $\mathrm{bc}[v, \pi_{vu}]$ be the number of block crossings in an optimum solution for $T[v]$ that results in order π_{vu} at vertex v on edge (u, v). If there is no feasible solution, that is, no solution without avoidable vertex crossings and—for MBCM—without double crossings, for the given order π_{vu}, then we let $\mathrm{bc}[v, \pi_{vu}] = \infty$.

If v is a leaf, then $|L_e| = 1$, there is only one possible order π_{vu}, and $\mathrm{bc}[v, \pi_{vu}] = 0$. Now, suppose that v has children $u_1, \dots, u_k$, with $k < \Delta$. For computing value $\mathrm{bc}[v, \pi_{vu}]$, we test all combinations of permutations π_{u_iv} for u_i with $i = 1, \dots, k$; see Figure 18a. Given such permutations, we can combine optimum solutions for the subtrees $T[u_1], \dots, T[u_k]$ resulting in orders $\pi_{u_1v}, \dots, \pi_{u_kv}$ with an optimum solution for the remaining instance, which consists of the edges $(u_1, v), \dots, (u_k, v)$ and is described by the orders $\pi_{u_1v}, \dots, \pi_{u_kv}$ and π_{vu}; see the bold region in Figure 18a and the transformed instance shown in Figure 18b. Let $f(v, \pi_{vu}, \pi_{u_1v}, \dots, \pi_{u_kv})$ be the number of block crossings in an optimum solution of this remaining instance; note that this value can be computed in constant time because the remaining instance has only constant size. Then,

$$\mathrm{bc}[v, \pi_{vu}] = \min_{\pi_{u_1v}, \dots, \pi_{u_kv}} \left(f(v, \pi_{vu}, \pi_{u_1v}, \dots, \pi_{u_kv}) + \sum_{i=1}^{k} \mathrm{bc}[u_i, \pi_{u_iv}] \right).$$

Note that $f(v, \pi_{vu}, \pi_{u_1v}, \dots, \pi_{u_kv}) = \infty$ if the permutations lead to an infeasible solution with avoidable vertex crossings or—for MBCM—double crossings. Table $\text{bc}[\cdot,\cdot]$ has at most $n \cdot c! = \mathcal{O}(n)$ entries, each of which can be computed in constant time. Hence, we get the following theorem.

Theorem 10 *BCM and MBCM can be solved optimally in $\mathcal{O}(n)$ time on trees of maximum degree Δ and maximum edge multiplicity c if both Δ and c are constants.*

Now, we want to analyze the runtime for computing an entry $\text{bc}[v, \pi_{vu}]$ more precisely. First, there are at most $(c!)^{\Delta-1}$ combinations for the orders $\pi_{u_1v}, \dots, \pi_{u_kv}$. Second, for computing $f(v, \pi_{vu}, \pi_{u_1v}, \dots, \pi_{u_kv})$, we can try all combinations for the orders on the edges around v. If such a combination leads to a feasible solution, we can solve each edge—as a permutation of constant size—individually; using breadth first search this is possible in $\mathcal{O}(\binom{c}{3} \cdot c!) = \mathcal{O}(c^3 c!)$ time since there are $c!$ permutations of length c and each permutation can be transformed into another permutation by $\mathcal{O}(\binom{c}{3})$ many block moves, each described by a triple of positions. Overall, evaluating $f(v, \pi_{vu}, \pi_{u_1v}, \dots, \pi_{u_kv})$ is then possible in $\mathcal{O}((c!)^{\Delta-1}\Delta c^3 c!) = \mathcal{O}((c!)^{\Delta}\Delta c^3)$ time. The total time for finding an optimum solution is, hence, $\mathcal{O}(n \cdot c! \cdot (c!)^{\Delta-1} \cdot (c!)^{\Delta}\Delta c^3) = \mathcal{O}(n(c!)^{2\Delta}\Delta c^3)$. As parameters c and Δ are well-separated from n, we can conclude as follows.

Corollary 3 *BCM and MBCM are fixed-parameter tractable on trees with respect to the parameter $c + \Delta$, where Δ is the maximum degree and c is the maximum edge multiplicity. The problems can be solved in $\mathcal{O}(n \cdot (c!)^{2\Delta}\Delta c^3)$ time.*

6.2 NP-Hardness of Restricted BCM and MBCM

We now show that restricted MBCM is NP-hard on general graphs. More specifically, it is NP-hard even if the graph is planar, the maximum degree is 3, and there is no edge with more than 11 lines.

Theorem 11 *MBCM is NP-hard on planar graphs even if the maximum degree is* 3 *and the maximum edge multiplicity is* 11.

Proof: We show hardness by reduction from PLANAR 3SAT, which is known to be NP-hard even if every variable occurs in exactly three different clauses [11]. Let (X, C) be an instance of PLANAR 3SAT; that is, X is a set of variables and C is a set of clauses consisting of literals, which are negated or unnegated variables, such that for any clause $c \in C$ (with $c \subseteq X \cup \{\neg x \mid x \in X\}$), it holds that $|c| \in \{2, 3\}$. Additionally, we can assume that $|\{c \in C \mid x \in c \text{ or } \neg x \in c\}| = 3$ for each $x \in X$. Graph $G_{XC} = (X \cup C, E_{XC})$ describing the occurrence of variables in clauses with the edge set $E_{XC} = \{\{x, \gamma\} \mid \text{ variable } x \text{ occurs in clause } \gamma\}$ is planar.

We now construct an instance $(G = (V, E), \mathcal{L})$ of MBCM modeling the 3SAT instance. To this end, we take a fixed planar embedding of G_{XC}. We replace each variable $x \in X$ in G_{XC} by a *variable gadget* V_x and each clause $\gamma \in C$ by a *clause gadget* C_γ. If $x \in \gamma$, then the edge $\{x, \gamma\}$ becomes an edge $\{v_x, v_\gamma\}$ where v_x and v_γ are vertices of the variable gadget and the clause gadget, respectively.

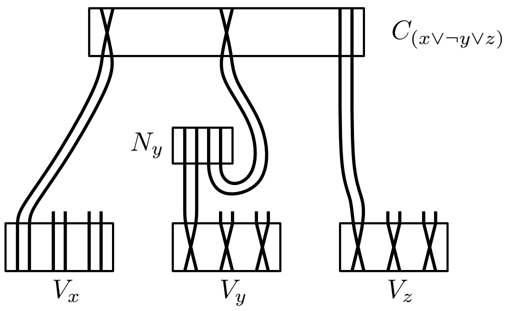


Figure 19: Connections of variable paths for a clause $\gamma = (x \vee \neg y \vee z)$ where x is `false` and y and z are `true`.

If $\neg x \in \gamma$, we replace edge $\{x, \gamma\}$ by a path (v_x, u, u', v_γ) where u and u' are vertices of a *negator gadget* N_x. In both cases, we call the edges of the connection between the gadgets the *variable path*. By placing the gadgets in the positions of the respective vertices of G_{XC} and routing the variable paths along the edges, we get a planar embedding of G.

On every edge outside of a gadget, exactly two lines represent a literal. We say that there the literal state is `true` if the lines cross in the one of the neighboring gadgets that is farther away from the clause gadget; it is `false` otherwise, that is, if the lines do not cross in the neighboring gadget farther away from the clause gadget. We will build the gadgets in such a way that crossings occur only within gadgets in crossing minimal solutions. Furthermore, we will connect the lines so that each pair of lines representing a literal has to cross, that is, it can be either in `true` or in `false` state. Figure 19 shows the connections for the variables of a clause. Note that, if we have the state `false` on an edge adjacent to a clause gadget, this means that the corresponding lines still have to cross within this clause gadget.

We first define the properties that we need for our gadgets. In the descriptions, we use global constants k_{var}, k_{neg}, k_{cls}, and k'_{cls} for numbers of crossings.

Variable Gadget: The variable gadget has three port edges e_1, e_2, and e_3 that are part of variable paths, and each of these edges has exactly two lines on it. These edges and lines are the only ones that leave the gadget. In a crossing-minimal solution in which the three pairs of lines either do or do not cross inside the gadget, there are exactly k_{var} crossings in the gadget. Any solution in which some, but not all, of these pairs cross inside the gadget has at least $k_{\text{var}} + 1$ crossings.

Negator Gadget: The negator gadget is basically a version of the variable gadget with only two ports. There are two port edges e_1 and e_2, each with a pair of lines. In crossing-minimal solutions in which both pairs either do or do not cross inside the gadget, there are exactly k_{neg} crossings. In the configurations in which exactly one of the pairs crosses inside the gadget, there are at least $k_{\text{neg}} + 1$ crossings.

Clause Gadget: The clause gadget has three (or two) port edges, each with a pair of lines. If at least one of the pairs does not cross inside the gadget, there are exactly k_{cls} crossings; if all pairs cross inside the gadget, at least $k_{\text{cls}} + 1$ crossings are necessary.

We also need a version of the clause gadget with only two port edges, both with a pair of lines. In this version, there are exactly k'_{cls} crossings if at least one of the pairs of lines does not cross inside the gadget; otherwise, there are at least $k'_{\text{cls}} + 1$ crossings.

Given such gadgets, we build the network that models the 3SAT instance. We are interested only in *canonical solutions*, that is, solutions in which (i) all crossings are inside gadgets and (ii) any variable gadget has exactly k_{var} crossings, any negator gadget has exactly k_{neg} crossings, and any clause gadget has exactly k_{cls} crossings (or k'_{cls} crossings if the clause has just two literals), resulting in a total number K of allowed crossings. It is easy to see that canonical solutions are exactly the solutions with at most K crossings. We claim that, if there is a canonical solution, the instance of 3SAT is satisfiable.

To see this, we analyze the variable gadget. As there are only k_{var} crossings in a canonical solution, the pairs of lines modeling the variable values either all cross, or all stay crossing-free. Hence, after leaving the gadget, the three pairs all have the same state, `true` if they crossed, and `false` otherwise. As there are no crossings outside of gadgets, this state can only change on the variable path if it contains a negator.

Suppose a variable path contains a negator gadget. In this case two lines ℓ_1 and ℓ_2, coming from a variable gadget, are connected by port edge e_1, and two lines ℓ_3 and ℓ_4, leaving towards a clause gadget, are connected by port edge e_2. As we consider a canonical solution, there are only two possibilities. If both pairs do not cross inside the negator, the pair $\{\ell_1, \ell_2\}$ has to cross in the variable gadget and, therefore, is in `true` state. Then, pair $\{\ell_3, \ell_4\}$ is in `false` state, as the lines do not cross in the negator gadget. On the other hand, if both pairs cross inside the negator, pair $\{\ell_1, \ell_2\}$ represents `false`, and $\{\ell_3, \ell_4\}$ represents `true`. Hence, the negator gadget works as desired.

Finally, we consider the clause gadgets. As there are only k_{cls} crossings (or k'_{cls} crossings in the version with only two literals), at least one of the variable pairs does not cross inside the gadget, which means that it is in `true` state. Hence, the clause is satisfied.

Now, suppose we are given a truth assignment that satisfies all clauses. We want to build a canonical solution for the block crossing problem. To this end, we fix, for each variable gadget, the order of the pairs of lines (crossing or non-crossing) corresponding to the truth value of the variable, which is the same for all port edges. Then, we take the appropriate solution with k_{var} block crossings for this gadget. Next, for each negator gadget, there is exactly one possible realization with k_{neg} block crossings given the state of the pair of lines on the ingoing port edge. Finally, for each clause gadget, there is at least one variable pair that did already cross, as the given truth assignment satisfies all variables. Hence, we can realize the clause gadget with only k_{cls} block crossings (or k'_{cls} block crossings in the version with two literals). Therefore, we can find a canonical solution.

We have now seen that, assuming that there are appropriate gadgets, deciding the satisfiability of a given instance of Planar 3SAT is equivalent to deciding whether the corresponding instance of MBCM has a canonical solution. To complete the proof, it remains to show how to build the gadgets with the desired properties. We do so in lemmas 12, 13, and 14. In the constructions, no edge contains more than 11 lines; the maximum degree of the underlying graph is 12. We can, however, easily modify the gadgets so that the maximum degree is 3 as follows. Each gadget basically contains a central edge with all lines of the gadget. On both sides of the central edge e of each gadget, we replace the vertex where the lines split by a tree-like structure in which the lines split into only two groups per step; we have indicated this modification for a negator gadget in Figure 21. Note that this modification does neither allow to save block crossings, nor does it make additional crossings necessary. This completes the proof. □

In the following three lemmas, we show how to build gadgets with the desired properties for the previous hardness proof.

Lemma 12 (Negator gadget) *There exists a negator gadget with* 10 *lines,* $k_{\mathrm{neg}} = 5$*, and the claimed properties for monotone block crossings.*

Proof: The negator gadget is illustrated in Figure 20a. It consists of an edge e with 10 lines, two port edges e_1 and e_2 with two lines each, and 16 edges, connected to leaves, with one line per edge. Assuming that the lines on e form the identity permutation on the lower end of the edge, we can read different permutations on the upper end, depending on the solution. However, the upper permutation always follows the *permutation template*

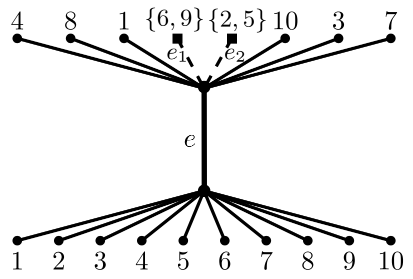
$$\pi_{\mathrm{neg}} = [4, 8, 1, a_1, a_2, b_1, b_2, 10, 3, 7],$$

where $\{a_1, a_2\} = \{6, 9\}$ and $\{b_1, b_2\} = \{2, 5\}$. Pairs $\{a_1, a_2\}$ and $\{b_1, b_2\}$ are on port edges e_1 and e_2, respectively, and they are connected to a variable or negator gadget.

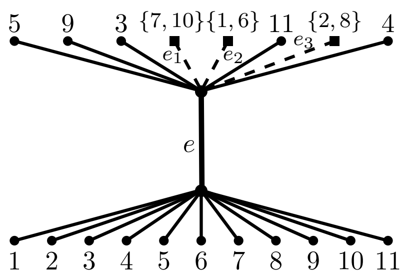
The important property of the permutations of type π_{neg} is that there are only two ways to arrange the lines in any solution of MBCM with the minimum number of block crossings. It is not hard to check that

- $\mathrm{mbc}(\pi) = 5$ if $\pi = [4, 8, 1, \underline{6, 9}, \underline{2, 5}, 10, 3, 7]$ or $\pi = [4, 8, 1, \underline{9, 6}, \underline{5, 2}, 10, 3, 7]$ and
- $\mathrm{mbc}(\pi) = 6$ in the remaining cases, that is, if $\pi = [4, 8, 1, \underline{6, 9}, \underline{5, 2}, 10, 3, 7]$ or $\pi = [4, 8, 1, \underline{9, 6}, \underline{2, 5}, 10, 3, 7]$.[1]

[1]These small instances of MBCM can be solved exactly by exhaustive search; see our implementation at `http://jgaa.info/accepted/2015/FinkPupyrevWolff2015.19.1/BlockCrossings.java`.



(a) Negator gadget.



(b) Main part of the variable gadget.

Figure 20: Gadgets for the NP-hardness proof. Lines starting/ending in leaves of the graph and passing through port edges (dashed)) are indicated by numbers (or sets of two numbers for port edges).

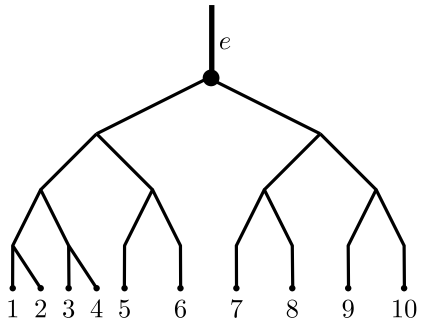


Figure 21: Lower part of a negator gadget modified for maximum degree 3.

Given a canonical solution, we can assume that the pairs of lines a_1, a_2 and b_1, b_2 do not cross on edges e_1 and e_2 since crossings on these edges can be moved to e without increasing the total number of block crossings in the solution. Hence, in a canonical solution, both pairs of lines $\{a_1, a_2\}$ and $\{b_1, b_2\}$ either cross on e or do not cross there. □

Lemma 13 (Variable gadget) *There exists a variable gadget with* 11 *lines per edge,* $k_{\text{var}} = 6$*, and the claimed properties for monotone block crossings.*

Proof: The basic part of the variable gadget is illustrated in Figure 20b. Its structure is similar to the negator gadget: The gadget consists of an edge e with 11 lines, three port edges e_1, e_2, and e_3 with two lines each, and 16 edges with one line per edge. Again, we can assume that all the crossings are located on e in a canonical solution. The lines on e form a permutation of the template

$$\pi_{\text{var}} = [5, 9, 3, a_1, a_2, b_1, b_2, 11, c_1, c_2, 4],$$

where $\{a_1, a_2\} = \{7, 10\}$, $\{b_1, b_2\} = \{1, 6\}$, and $\{c_1, c_2\} = \{2, 8\}$.

One can check that

- $\mathrm{mbc}(\pi) = 6$ if either $\pi = [5, 9, 3, \underline{7, 10}, \underline{1, 6}, 11, \underline{8, 2}, 4]$ or, by exchanging the lines of the marked pairs, $\pi = [5, 9, 3, \underline{10, 7}, \underline{6, 1}, 11, \underline{2, 8}, 4]$, and
- $\mathrm{mbc}(\pi) = 7$ in the remaining six cases that follow the template π_{var}.

In other words, in a canonical solution the pairs of lines $\{a_1, a_2\}$, $\{b_1, b_2\}$, and $\{c_1, c_2\}$ form either the state (`true`, `true`, `false`) or (`false`, `false`, `true`) in the gadget. We use an additional negator—as described in Lemma 12—connected to pair $\{c_1, c_2\}$ by the port edge e_3, so that, in a canonical solution, the variable gadget encodes either `true` or `false` for all variable pairs at the same time. □

Lemma 14 (Clause gadget) *There exists a clause gadget with six lines, $k_{\mathrm{cls}} = 2$, and the claimed properties for monotone block crossings. Furthermore, there exists a clause gadget for only two variables with four lines, $k'_{\mathrm{cls}} = 1$, and the claimed properties for monotone block crossings.*

Proof: The clause gadget is illustrated in Figure 22a. It consists of an edge e with 6 lines, three port edges e_1, e_2, and e_3 with two lines each, and 6 edges with one line per edge. The lines form a permutation of the template

$$\pi_{\mathrm{cls}} = [a_1, a_2, b_1, b_2, c_1, c_2],$$

where $\{a_1, a_2\} = \{1, 3\}$, $\{b_1, b_2\} = \{2, 5\}$, and $\{c_1, c_2\} = \{4, 6\}$.

One can check that

- $\mathrm{mbc}(\pi) = 3$ if $\pi = [\underline{3, 1}, \underline{5, 2}, \underline{6, 4}]$ and
- $\mathrm{mbc}(\pi) = 2$ in the remaining five cases of permutations following template π_{cls}.

Hence, in a crossing optimal solution, at least one of the pairs of lines, $\{a_1, a_2\}$, $\{b_1, b_2\}$, and $\{c_1, c_2\}$, must *not* cross inside the gadget, that is, the corresponding literal must be `true`; see Figure 22b for an example of such a configuration.

By dropping edge e_3 and the corresponding two lines 4 and 6 and renaming line 5 to 4, we get a variant for clause gadgets with two literals. Then, we have a permutation of the template $\pi'_{\mathrm{cls}} = [a_1, a_2, b_1, b_2]$ where $\{a_1, a_2\} = \{1, 3\}$ and $\{b_1, b_2\} = \{2, 4\}$. One can check that

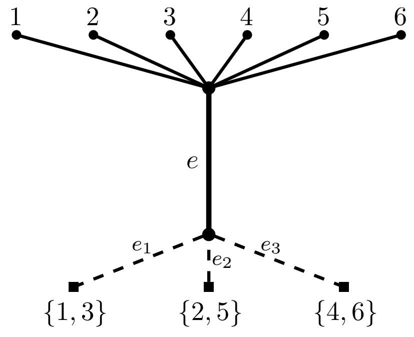
- $\mathrm{mbc}(\pi) = 2$ if $\pi = [\underline{3, 1}, \underline{4, 2}]$ and
- $\mathrm{mbc}(\pi) = 1$ in the remaining three cases following the template π'_{cls}.

Again, in a canonical solution, at least one of the literals corresponding to pairs $\{a_1, a_2\}$ and $\{b_1, b_2\}$ must be `true`. □

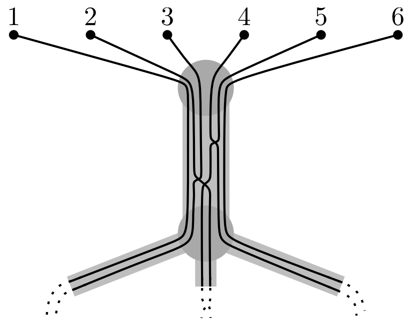
The general variant of BCM is NP-hard even for a single edge; see Theorem 1. For constant maximum degree and edge multiplicity, however, the problem is tractable on trees; see Theorem 10. Next we show that on general planar graphs BCM is NP-hard even for constant maximum degree and edge multiplicity. To this end, we modify the negator and variable gadgets; the clause gadget does not need to be changed because the properties of the permutations we used there still hold if we allow non-monotone block moves.

Negator gadget. The structure for the negator gadget stays the same as described in Lemma 12. We just replace the used permutation template by

$$\pi_{\mathrm{neg}} = [3, a_1, a_2, 4, 7, b_1, b_2],$$



(a) Clause gadget.



(b) Sketch of a solution for the clause gadget for $c = (l_1 \vee l_2 \vee l_3)$, where l_1 and l_3 are `true` and l_2 is `false`.

Figure 22: The clause gadget for the NP-hardness proof. Lines starting/ending in leaves of the graph and passing through port edges (dashed) are indicated by numbers (or sets of two numbers for port edges).

where the lines $\{a_1, a_2\} = \{1, 6\}$ leave the gadget on port edge e_1 and the lines $\{b_1, b_2\} = \{2, 5\}$ leave the gadget on e_2.

One can check that

- $\mathrm{bc}(\pi) = 3$ if $\pi = [3, \underline{1, 6}, 4, 7, \underline{2, 5}]$ or $\pi = [3, \underline{6, 1}, 4, 7, \underline{5, 2}]$ and
- $\mathrm{bc}(\pi) = 4$ in the remaining two cases for template π_{neg} (note that we now use non-monotone block crossings).

Hence, both pairs of lines $\{a_1, a_2\}$ and $\{b_1, b_2\}$ either cross in the gadget or do not cross there in a canonical solution.

Variable gadget. Also the structure of the variable gadget stays the same as described in Lemma 13. We just replace the used permutation template by

$$\pi_{\text{var}} = [6, a_1, a_2, b_1, b_2, c_1, c_2],$$

where lines a_1 and a_2 leave the gadget on port edge e_1, b_1 and b_2 leave the gadget on e_2, and c_1 and c_2 leave it on e_3; furthermore, $\{a_1, a_2\} = \{1, 4\}$, $\{b_1, b_2\} = \{3, 7\}$, and $\{c_1, c_2\} = \{2, 5\}$.

One can check that

- $\mathrm{bc}(\pi) = 3$ if $\pi = [6, \underline{1, 4}, \underline{3, 7}, \underline{5, 2}]$ or $\pi = [6, \underline{4, 1}, \underline{7, 3}, \underline{2, 5}]$ and
- $\mathrm{bc}(\pi) = 4$ in the remaining six cases for template π_{var}.

Hence, in a canonical solution the pairs of lines a_1, a_2, b_1, b_2, and c_1, c_2 form either the state (`true`, `true`, `false`) or (`false`, `false`, `true`) in the gadget. Again, we use an additional negator connected to pair c_1, c_2 by port edge e_3 for ensuring that the variable gadget encodes either `true` or `false` for all variable pairs at the same time.

Using the new gadgets, we immediately get the reduction for BCM. We note that we can ensure maximum degree 3 by the same construction that we used for MBCM. Note that both negator and variable gadget for BCM use fewer lines compared to MBCM; the maximum number of lines on an edge is 7.

Theorem 12 *BCM is NP-hard on planar graphs even if the maximum degree is* 3 *and the maximum edge multiplicity is* 7*.*

We point out that the hardness results for bounded degree and edge multiplicity imply that, in contrast to the case of trees, BCM and MBCM are not fixed-parameter tractable with respect to these parameters on general graphs. The problems could, however, be fixed-parameter tractable with respect to different parameters such as the number of crossings.

7 Conclusion and Open Problems

We have introduced the new variants BCM and MBCM of the metro-line crossing minimization problem in which one wants to order the lines taking more advanced crossings into account. We have presented approximation algorithms for single edges, paths, and upward trees. Then we have developed an algorithm that bounds the number of block crossings on general graphs and have showed that our bound is asymptotically tight. Finally, we have investigated the problems under bounded maximum degree and edge multiplicity, both of which are valid assumptions for practical purposes. Under these restrictions, we have solved BCM and MBCM optimally on trees by giving a fixed-parameter tractable algorithm. Additionally, we have proven that BCM and MBCM are NP-hard on general graphs even if maximum degree and edge multiplicity are small.

Open Problems. As our results are the first for block crossing minimization, there are still many interesting open problems. First, the complexity status of MBCM on a single edge would be interesting to know, mainly from a theoretical point of view. The hardness proof for BCM is quite complicated and does not easily extend to MBCM. An improvement of the current approximation factor 3 for MBCM on an edge is also interesting. Second, a challenging task is to develop approximation algorithms for BCM and MBCM on general graphs. Note that the graphs for the worst-case instances for the problems are not planar. It is interesting to decide whether our bound on the number of necessary block crossings is asymptotically tight for planar instances. Another important question is whether there exists a fixed-parameter tractable algorithm for BCM and MBCM on paths, trees, and general graphs with respect to the allowed number of block crossings.

Recently, Bereg et al. [6] investigated the problem of drawing permutations with few bends; they represented each element of the permutation as a line, similar to a metro line. Also for the visual complexity of a metro line an important criterion is the number of its bends. Hence, an interesting question is how to visualize metro lines using the minimum total number of bends.

Acknowledgments

We are grateful to Sergey Bereg, Alexander E. Holroyd, and Lev Nachmanson for the initial discussion of the block crossing minimization problem, and for pointing out a connection with sorting by transpositions. We thank Jan-Henrik Haunert and Joachim Spoerhase for fruitful discussions and suggestions. We also like to thank the two anonymous reviewers of this paper for many helpful comments and suggestions that helped improve the presentation of the final version of the paper.

References

[1] E. N. Argyriou, M. A. Bekos, M. Kaufmann, and A. Symvonis. On metro-line crossing minimization. *J. Graph Algorithms Appl.*, 14(1):75–96, 2010. doi:10.7155/jgaa.00199.

[2] M. Asquith, J. Gudmundsson, and D. Merrick. An ILP for the metro-line crossing problem. In J. Harland and P. Manyem, editors, *Proc. 14th Comput.: Australian Theory Symp. (CATS'08)*, volume 77 of *CRPIT*, pages 49–56. ACS, 2008. URL: http://crpit.com/abstracts/CRPITV77Asquith.html.

[3] V. Bafna and P. A. Pevzner. Sorting by transpositions. *SIAM J. Discr. Math.*, 11(2):224–240, 1998. doi:10.1137/S089548019528280X.

[4] M. A. Bekos, M. Kaufmann, K. Potika, and A. Symvonis. Line crossing minimization on metro maps. In S.-H. Hong, T. Nishizeki, and W. Quan, editors, *Proc. 15th Int. Symp. Graph Drawing (GD'07)*, volume 4875 of *LNCS*, pages 231–242. Springer, 2008. doi:10.1007/978-3-540-77537-9_24.

[5] M. Benkert, M. Nöllenburg, T. Uno, and A. Wolff. Minimizing intra-edge crossings in wiring diagrams and public transportation maps. In M. Kaufmann and D. Wagner, editors, *Proc. 14th Int. Symp. Graph Drawing (GD'06)*, volume 4372 of *LNCS*, pages 270–281. Springer, 2007. doi:10.1007/978-3-540-70904-6_27.

[6] S. Bereg, A. E. Holroyd, L. Nachmanson, and S. Pupyrev. Drawing permutations with few corners. In S. Wismath and A. Wolff, editors, *Proc. 21st Int. Symp. Graph Drawing (GD'13)*, volume 8242 of *LNCS*, pages 484–495. Springer, 2013. doi:10.1007/978-3-319-03841-4_42.

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

[8] T. M. Chan and M. Pătraşcu. Counting inversions, offline orthogonal range counting, and related problems. In *Proc. 21st ACM-SIAM Symp. Discrete Algorithms (SODA'10)*, pages 161–173, 2010. doi:10.1137/1.9781611973075.15.

[9] D. A. Christie and R. W. Irving. Sorting strings by reversals and by transpositions. *SIAM J. Discr. Math.*, 14(2):193–206, 2001. doi:10.1137/S0895480197331995.

[10] W. Cui, H. Zhou, H. Qu, P. C. Wong, and X. Li. Geometry-based edge clustering for graph visualization. *IEEE Trans. Vis. Comput. Graph.*, 14(6):1277–1284, 2008. doi:10.1109/TVCG.2008.135.

[11] E. Dahlhaus, D. S. Johnson, C. H. Papadimitriou, P. D. Seymour, and M. Yannakakis. The complexity of multiterminal cuts. *SIAM J. Comput.*, 23(4):864–894, 1994. doi:10.1137/S0097539792225297.

[12] I. Elias and T. Hartman. A 1.375-approximation algorithm for sorting by transpositions. *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, 3(4):369–379, 2006. `doi:10.1109/TCBB.2006.44`.

[13] 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`.

[14] G. Fertin, A. Labarre, I. Rusu, E. Tannier, and S. Vialette. *Combinatorics of Genome Rearrangements*. MIT Press, 2009.

[15] M. Fink and S. Pupyrev. Metro-line crossing minimization: Hardness, approximations, and tractable cases. In S. Wismath and A. Wolff, editors, *Proc. 21st Int. Symp. Graph Drawing (GD'13)*, volume 8242 of *LNCS*, pages 328–339. Springer, 2013. `doi:10.1007/978-3-319-03841-4_29`.

[16] M. Fink and S. Pupyrev. Ordering metro lines by block crossings. In K. Chatterjee and J. Sgall, editors, *Proc. 38th Int. Symp. Mathematical Foundations of Computer Science (MFCS'13)*, volume 8087 of *LNCS*, pages 397–408. Springer, 2013. `doi:10.1007/978-3-642-40313-2_36`.

[17] E. Gansner, Y. Hu, S. North, and C. Scheidegger. Multilevel agglomerative edge bundling for visualizing large graphs. In G. D. Battista, J.-D. Fekete, and H. Qu, editors, *Proc. Pacific Visualization Symposium (PacificVis'11)*, pages 187–194. IEEE, 2011. `doi:10.1109/PACIFICVIS.2011.5742389`.

[18] P. Groeneveld. Wire ordering for detailed routing. *IEEE Des. Test*, 6(6):6–17, 1989. `doi:10.1109/54.41670`.

[19] L. S. Heath and J. P. C. Vergara. Sorting by bounded block-moves. *Discrete Appl. Math.*, 88(1–3):181–206, 1998. `doi:10.1016/S0166-218X(98)00072-9`.

[20] D. Holten. Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. *IEEE Trans. Vis. Comput. Graph.*, 12(5):741–748, 2006. `doi:10.1109/TVCG.2006.147`.

[21] M. Marek-Sadowska and M. Sarrafzadeh. The crossing distribution problem. *IEEE Trans. CAD Integrated Circuits Syst.*, 14(4):423–433, 1995. `doi:10.1109/43.372368`.

[22] M. Nöllenburg. *Network Visualization: Algorithms, Applications, and Complexity*. PhD thesis, Fakultät für Informatik, Universität Karlsruhe (TH), 2009. URL: `http://nbn-resolving.org/urn:nbn:de:swb:90-114562`.

[23] M. Nöllenburg. An improved algorithm for the metro-line crossing minimization problem. In D. Eppstein and E. R. Gansner, editors, *Proc. 17th Int. Symp. Graph Drawing (GD'09)*, volume 5849 of *LNCS*, pages 381–392. Springer, 2010. `doi:10.1007/978-3-642-11805-0_36`.

[24] Y. Okamoto, Y. Tatsu, and Y. Uno. Exact and fixed-parameter algorithms for metro-line crossing minimization problems. ArXiv e-print abs/1306.3538, 2013. URL: `http://arxiv.org/abs/1306.3538`.

[25] S. Pupyrev, L. Nachmanson, S. Bereg, and A. E. Holroyd. Edge routing with ordered bundles. In M. van Kreveld and B. Speckmann, editors, *Proc. 19th Int. Symp. Graph Drawing (GD'11)*, volume 7034 of *LNCS*, pages 136–147. Springer, 2012. `doi:10.1007/978-3-642-25878-7_14`.

[26] F. Schreiber. High quality visualization of biochemical pathways in BioPath. *In Silico Biol.*, 2(2):59–73, 2002. URL: `http://iospress.metapress.com/content/1MDPFN33UQ5729JK`.

[27] O. Veblen and W. H. Bussey. Finite projective geometries. *Trans. AMS*, 7(2):241–259, 1906. URL: `http://www.jstor.org/stable/1986438`, `doi:10.2307/1986438`.