Grid Sifting: Leveling and Crossing Reduction

CHRISTIAN BACHMAIER, WOLFGANG BRUNNER, and ANDREAS GLEIßNER,
University of Passau

Directed graphs are commonly drawn by the Sugiyama algorithm where first vertices are placed on distinct hierarchical levels, and second the vertices on the same level are permuted to reduce the overall number of crossings. Separating these two phases simplifies the algorithms but diminishes the quality of the result.

We introduce a combined leveling and crossing reduction algorithm based on sifting, which prioritizes few crossings over few levels. It avoids type 2 conflicts, which are crossings of edges whose endpoints are dummy vertices. This helps straightening long edges spanning many levels. The obtained running time is roughly quadratic in the size of the input graph and independent of dummy vertices.

Categories and Subject Descriptors: F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Routing and Layout*; G.2.2 [**Discrete Mathematics**]: Graph Theory—*Graph algorithms*

General Terms: Algorithms, Design, Experimentation, Performance, Theory

Additional Key Words and Phrases: Crossing reduction, global approach, level graphs, no dummy vertices, Sugiyama framework

ACM Reference Format:
Bachmaier, C., Brunner, W., Gleißner, A. 2011. Grid Sifting: Leveling and Crossing Reduction. ACM J. Exp. Algor. 17, 1, Article 1.7 (September 2012), 23 pages.
DOI = 10.1145/2133803.2345682 http://doi.acm.org/10.1145/2133803.2345682

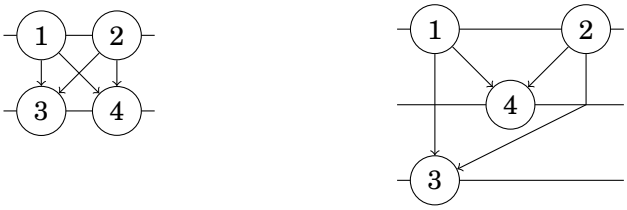
1. INTRODUCTION

The Sugiyama framework [Sugiyama et al. 1981] is the standard drawing algorithm for directed graphs. It displays them in a hierarchical manner and operates in four phases: cycle removal (reverse appropriate edges to eliminate cycles), leveling (assign vertices to levels which define the y-coordinates and introduce dummy vertices on long edges), crossing reduction (permute the vertices on the levels), and coordinate assignment (assign x-coordinates to the vertices according to some aesthetic criteria). Typical applications are schedules, UML diagrams, and flow charts.

There are many different leveling and crossing reduction algorithms. Traditional leveling methods minimize the number of levels by the longest path method [Kaufmann and Wagner 2001], by the Coffman/Graham algorithm with a predefined maximum number of vertices per level (*width*) [Coffman and Graham 1972], or by Gansner et al.'s [1993] ILP minimizing the sum of the edge lengths. The common solution for *k-level crossing minimization* is a reduction to the still $\mathcal{NP}$-hard [Eades and Wormald 1994] *one-sided 2-level crossing minimization problem*, which is repeatedly solved

This work is supported by the Deutsche Forschungsgemeinschaft (DFG) under grant Br835/15-1 and grant Br835/15-2.
Author's address: Faculty of Informatics and Mathematics, Innstr. 33, University of Passau, 94030 Passau, Germany.

DOI 10.1145/2133803.2345682 http://doi.acm.org/10.1145/2133803.2345682



(a) Two levels. (b) Three levels.

Fig. 1. The $K_{2,2}$ as typical pattern on two levels of an unavoidable crossing.

by some up and down sweeps [Sugiyama et al. 1981; Kaufmann and Wagner 2001]. Bastert and Matuszewski [Kaufmann and Wagner 2001] claim that the results of this level-by-level sweep are far from optimum. "One can expect better results by considering all levels simultaneously, but k-level crossing minimization is a very hard problem" [Kaufmann and Wagner 2001, p. 102].

An important feature of crossing reduction algorithms is the avoidance of *type 2 conflicts*, which are crossings of two edges between dummy vertices. Among others, the standard fourth phase algorithm by Brandes and Köpf [2002] assumes the absence of type 2 conflicts. Then, it aligns long edges vertically and so achieves a crucial aesthetic criterion [Kaufmann and Wagner 2001] regarding pleasing hierarchical drawings.

Sifting was at first used for vertex minimization in ordered binary decision diagrams [Rudell 1993] and was later on adapted to the one-sided 2-level crossing reduction [Matuszewski et al. 1999]. The idea is to keep track of the number of crossings while in a *sifting step* a vertex v on level i is moved along a fixed order of all the vertices V_i on level i. Finally, v is placed at its locally optimal position. We call a single swap a *sifting swap*, and we call the execution of a sifting step for every vertex in V_i a *sifting round*.

Matuszewski et al. [1999] have extended sifting toward a global view. They sort all vertices by their degree and sift them first in increasing and then in decreasing order. This is no traditional level-by-level approach, although only crossings between two pairs of neighboring levels are considered. Jünger et al. [1997] presented an exact ILP approach for the $\mathcal{NP}$-hard k-level crossing minimization that can be used in practice for small graphs. Moreover, metaheuristics have been suggested in literature, for example, genetic algorithms [Utech et al. 1998; Kuntz et al. 2006], tabu search [Laguna et al. 1997], or windows optimization [Eschbach et al. 2002]. We have presented a global sifting algorithm [Bachmaier et al. 2011], which aligns long edges to blocks and tries to find an apt position for each block as a whole.

Although these algorithms try to achieve a global view on the overall number of crossings, they do not change the leveling. "Since level assignment and crossing reduction are realized as independent steps, the resulting drawing might have many unnecessary crossings caused by an unfortunate level assignment" [Chimani et al. 2011, p. 128]. A minimal and typical pattern for level graphs showing this phenomenon is the complete bipartite graph $K_{2,2}$. With two vertices placed on one level and two on another level, there is one unavoidable crossing independent of the order of the vertices on their respective levels (see Figure 1). If, however, the leveling is free, the crossing can be resolved by placing one vertex on a third level. This basic example motivates the idea of yielding less crossings by a different leveling. In general, more levels lead to fewer crossings.

Recently, Chimani et al. [2010; 2011] presented a planarization approach to reduce the number of crossings. They compute a planar upward embedding of the graph by

adding dummy vertices where two edges would cross. This embedding is leveled respecting the given orders of the adjacency lists. Then, the number of at most $\mathcal{O}(|V|)$ levels is reduced by compacting the drawing. The algorithm needs $\mathcal{O}(|E|^5)$ time and still $\mathcal{O}(|E|^3) = \mathcal{O}(|V|^3)$ time for planar input graphs.

In this article we propose a new leveling and crossing reduction technique that combines both phases in a natural way. We extend global sifting [Bachmaier et al. 2011] to *grid sifting* such that the blocks are not only sifted "horizontally" on their levels but also "vertically" on all suitable levels and prioritize the crossing reduction over the leveling. The algorithm yields better results than traditional heuristics. It avoids type 2 conflicts and runs in roughly quadratic time in the size of the input graph.

2. PRELIMINARIES

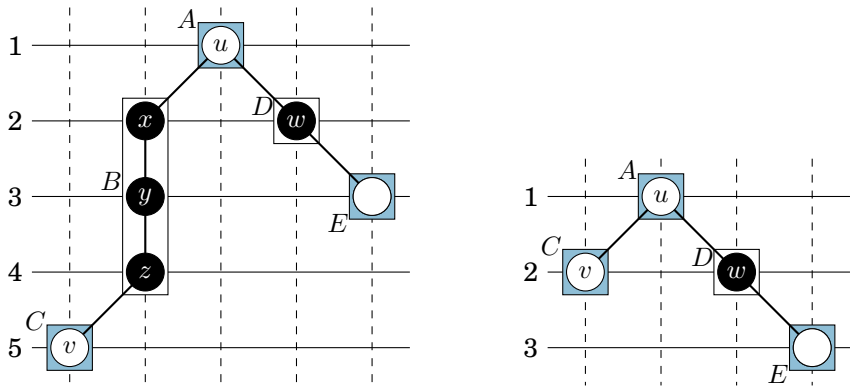
We suppose that a directed graph without self-loops has passed through the cycle removal phase and has been assigned an initial leveling. The outcome is a *k-level graph* $G = (V, E, \phi)$, where $\phi\colon V \to \{1, 2, \ldots, k\}$ is a surjective level assignment of the vertices with $\phi(u) < \phi(v)$ for each edge $(u, v) \in E$. For an edge $e = (u, v) \in E$ we define $\operatorname{span}(e) := \phi(v) - \phi(u)$. An edge e is ***short*** if $\operatorname{span}(e) = 1$ and *long* otherwise. A graph is *proper* if all edges are short.

The traditional approach is to make each level graph proper by adding $\operatorname{span}(e) - 1$ dummy vertices for each edge e, which splits e in $\operatorname{span}(e)$ many short edges. Let $G' = (V', E', \phi')$ denote the proper version of G. As in Brandes and Köpf [2002], we call short edges *segments* of e. The first and the last segments are the *outer segments* and the others the *inner segments*. Inner segments connect two dummy vertices. For a *(level) embedded* proper level graph, the vertices on each level are ordered from left to right. In an embedded level graph, two segments are *conflicting* if they cross or share a vertex. Conflicts are of *type 0*, *1*, or *2* if they are induced by 0, 1, or 2 inner segments, respectively.

Similarly to Brandes and Köpf [2002], a *block* either represents a vertex of V (i. e., a *vertex block*), or an edge of E (i. e., an *edge block*). Note that a block is an entity of its own and is not treated as a set. The sets of vertex blocks, edge blocks, and all blocks are called $\mathcal{B}_V$, $\mathcal{B}_E$, and $\mathcal{B} = \mathcal{B}_V \dot\cup \mathcal{B}_E$. More formally, $\mathcal{B}_V := \{ B_v \mid v \in V \}$ and $\mathcal{B}_E := \{ B_e \mid e \in E \}$. Hence, the difference to Brandes and Köpf [2002] is that our edge blocks do not "cover" the endvertices of the respective edge; these build discrete vertex blocks. For each $x \in V \cup E$, denote by $\operatorname{block}(x)$ the block representing x. The set $\mathcal{F} := \{(\operatorname{block}(u), \operatorname{block}(e)) \in \mathcal{B}_V \times \mathcal{B}_E : e = (u, \cdot)\} \cup \{(\operatorname{block}(e), \operatorname{block}(v)) \in \mathcal{B}_E \times \mathcal{B}_V : e = (\cdot, v)\}$ models a relation between blocks induced by the incidence relation in G. $\mathcal{G} := (\mathcal{B}, \mathcal{F})$ forms the directed *block graph* of G. For an example, see Figure 2. For a long edge $e \in E$, its edge block $\operatorname{block}(e)$ corresponds to the maximum connected subgraph of dummy vertices in G', that is, the inner segments of e. Such blocks are called *active*. Likewise, blocks corresponding to short edges, which do not need any dummy vertices in G', are called *inactive*.[1] Edges of the block graph that are incident to an active edge block correspond to outer segments of G'. For a dummy vertex v' in G', let $\operatorname{block}(v')$ be the edge block corresponding to the subgraph containing v'.

Note that G' and the notion of dummy vertices are only used to simplify definitions and are never directly represented in the algorithm. As the grid sifting algorithm modifies the level assignment ϕ of vertex blocks, edge spans as well as the number of dummy vertices of G' change over time. Hence, edge blocks switch between the active and inactive state.

[1]Note that block graphs are also known as (bipartite) *incidence* or *Levi graphs*. However, we omit inactive blocks and visualize the incidence relation between vertex blocks as simple (short) edges.



(a) The edge (u, v) spans multiple levels such that its edge block B represents a path of three dummy vertices.

(b) The edge (u, v) is short, which results in an inactive edge block B.

Fig. 2. Example for the proper version G' of a graph G and its block graph $\mathcal{G} = (\{A, B, C, D, E\}, \{(A, B), (B, C), (A, D), (D, E)\})$ with two different levelings. The white circles are vertices of G, while the black are dummy vertices of G'. Blocks are shown as rectangles.

Due to technical reasons explained later, most vertex blocks lie on even levels. If an odd level number is not assigned to any vertex block, the level is nonexistent. In G', there is no need to create dummy vertices for that level number. Let $\text{next}^+(l)$ ($\text{next}^-(l)$, respectively) be the next existent level after (before) level l. A block is defined to *use* a level if a (dummy) vertex of the corresponding subgraph in G' is assigned to this level. Let $\text{levels}(B)$ be the set of all levels used by block B. We define $\overline{\phi}(B) := \min \text{levels}(B)$ ($\underline{\phi}(B) := \max \text{levels}(B)$, respectively) as the topmost (bottommost) level used by B. For each vertex block $B = \text{block}(v)$, $\overline{\phi}(B) = \underline{\phi}(B) = \phi(B) := \phi(v)$ holds. An active edge block $B = \text{block}((u, v))$ uses the levels $\overline{\phi}(B) = \text{next}^+(\phi(u)), \text{next}^+(\text{next}^+(\phi(u))), \ldots, \text{next}^-(\phi(v)) = \underline{\phi}(B)$. For inactive edge blocks B, the set $\text{levels}(B)$ is empty.

We introduce l-neighbors of (especially edge) blocks to address parts of these blocks as if the dummy vertices were existent. If (u, v) is any segment in G' and $l = \phi(u)$, we call $\text{block}(v)$ an *l-neighbor* of $\text{block}(u)$ in *direction* "+". Likewise we call $\text{block}(u)$ an *$l+1$-neighbor* of $\text{block}(v)$ in *direction* "–". Note that if additionally $\text{block}(u), \text{block}(v) \in \mathcal{B}_V$, then, ignoring the inactive edge block in between, u is a $\phi(v)$-neighbor of v and, accordingly, v is a $\phi(u)$-neighbor of u. For each block B, direction $d \in \{+, -\}$, and level $l \in \text{levels}(B)$, we define $N^d(B, l)$ to be the set of all l-neighbors of B in direction d. For an edge block B, $N^d(B, l)$ contains exactly one element. Note that if $\overline{\phi}(B) < l < \underline{\phi}(B)$, then $N^d(B, l) = \{B\}$, so that an edge block can be its own l-neighbor. For the perspective using dummy vertices, let v be the dummy vertex of an edge block B on level l. Then, the l-neighbor of B in direction d is the block of the neighbor of v in direction d, which is either B or an adjacent vertex block. For an example, see Figure 2. In Figure 2(a), the edge (u, v) is long such that the proper version of the graph contains three dummy vertices x, y, and z. They are represented by the edge block B. The 1-neighbors of A in "+"-direction, $N^+(A, 1)$, are B and D, since they contain x and w, the immediate successors of u. y is the dummy vertex of B on level 3. It has an edge to the dummy

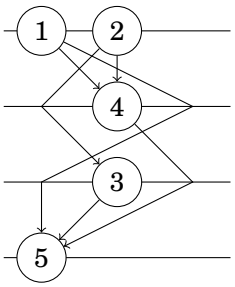


Fig. 3. Example of a 4-level graph.

vertex z, which also belongs to B. Thus, the set of 3-neighbors of B in "+"-direction consists of B itself, that is, $N^+(B,3) = \{B\}$. In Figure 2(b), the edge (u,v) is short. Thus, the edge block representing (u,v) disappears and is inactive. The proper graph reflects this by u and v connected by an edge. Consequently, $N^+(A,1) = \{C,D\}$. For each vertex block B, we define $N^d(B) := N^d(B,\phi(B))$, as B uses only one level. Let $\mathcal{B}$ be an arbitrarily ordered list of all blocks and let $\pi\colon\mathcal{B} \to \{0,\ldots,|\mathcal{B}|-1\}$ assign each block its current *position* in this order. Note that the drawing of a short edge (u,v) is independent of $\pi(\mathrm{block}((u,v)))$, since $\mathrm{block}((u,v))$ is inactive. In Figure 2(a), $\mathcal{B} = \{A,B,C,D,E\}$ with $\pi(C) < \pi(B) < \pi(A) < \pi(D) < \pi(E) = 4$. The same ordering fits to Figure 2(b). We call the pair $\mathcal{E} = (\pi,\phi)$ a *level embedding* of the graph.

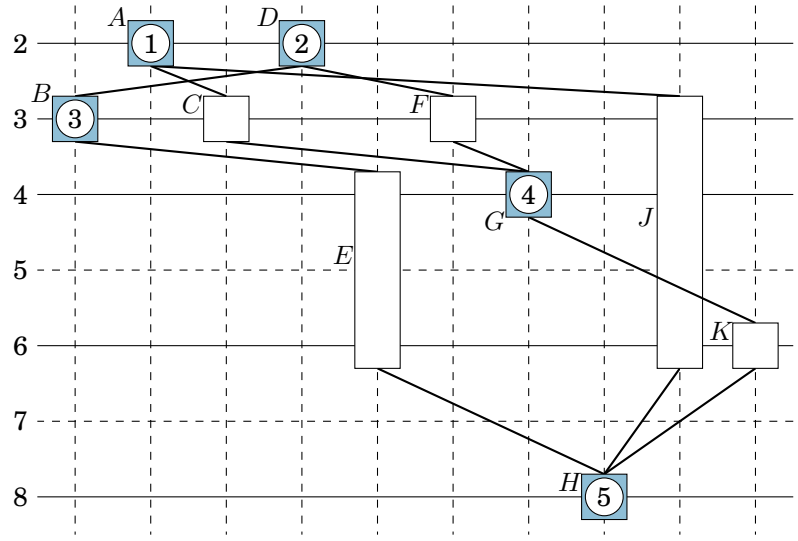
See Figure 3 for another example of a graph G with five vertices and six edges. In Figure 4(a) each of its vertices is represented by a vertex block, and five active edge blocks are shown. In contrast to Figure 2, we now do not use any dummy vertices in the block graph $\mathcal{G}$ of G. The edge $(2,3)$ is short in Figure 4(a), and hence its edge block is inactive and not visualized. Levels 5 and 7 are nonexistent (dashed). Thus, $\mathrm{next}^+(4) = 6$ and $\mathrm{next}^-(8) = 6$. Let J be the edge block of the edge $(1,5)$. Then, $\mathrm{levels}(J) = \{3,4,6\}, \overline{\phi}(J) = 3, \underline{\phi}(J) = 6, N^+(J,3) = \{J\}, N^+(J,4) = \{J\}$, and $N^+(J,6) = \{\mathrm{block}(5)\} = \{H\}$. Furthermore, $N^+(D) = \{B,F\}$ in Figure 4(a) (ignoring the currently inactive edge block $L = \mathrm{block}(2,3)$) and $N^+(D) = \{L,G\}$ in Figure 4(c) (as level 3 is nonexistent, $F = \mathrm{block}(2,4)$ is inactive).

3. GRID SIFTING

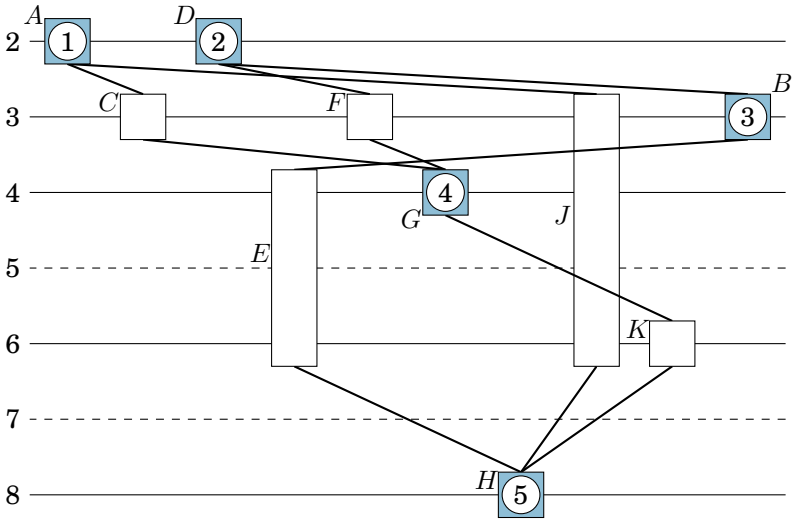
We extend our approach of global sifting [Bachmaier et al. 2011] and try to find optimal positions for each (vertex) block on all levels. We use a two-dimensional grid on which we place each vertex block (see Figure 4(a)). The edge blocks span the levels between their incident vertex blocks and represent the dummy vertices of each edge. Each block has a unique x-coordinate on the grid, which is given by the order of $\mathcal{B}$. As an initialization, we use an arbitrary leveling and a random permutation of $\mathcal{B}$.

If a given ordering should only be improved in a postprocessing step, a straightforward initialization strategy is to topologically sort the blocks according to the orderings on the levels from left to right in $\mathcal{O}(|V|+|E|)$. Our experiments showed that a good initial ordering of the blocks leads to better results. However, these can also be achieved by one or two additional sifting rounds.

Finding an optimal position for a vertex block basically consists of two nested loops. The outer loop, called a *vertical step*, iterates over all levels the vertex can use without reversing an incident edge. In the inner loop, called a *horizontal step*, all positions of the vertex block on the current level are tested. In the end, the vertex block is placed on the level and position causing the minimal number of crossings. See Algorithm 1 for the entry to the detailed description. If there are several positions causing the same



(a) The block graph of Figure 3 after normalization, that is, doubling the levels with B on its top- and leftmost admissible position.



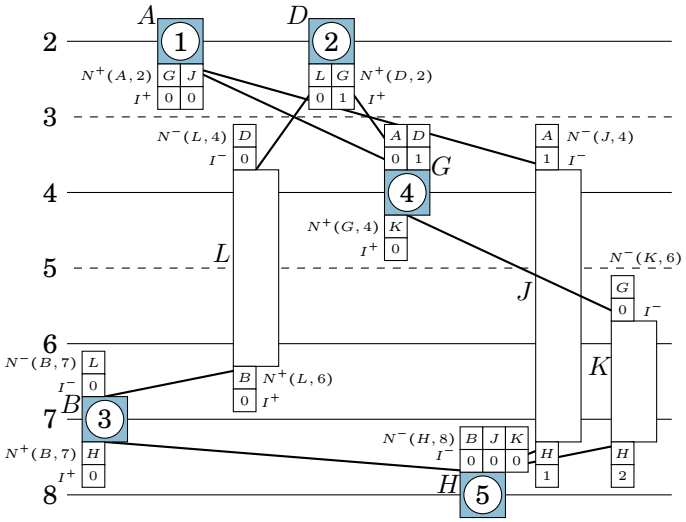
(b) B on its top- and rightmost position.

Fig. 4. Extremal positions of vertex block B during its vertical step.

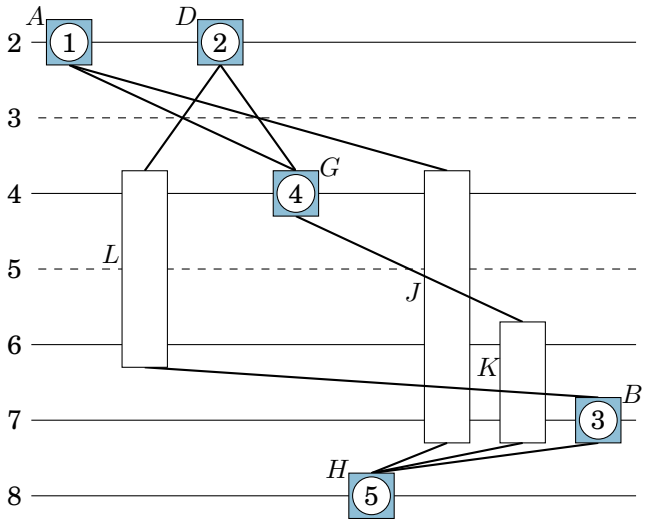
number of crossings, we place the vertex block at the level nearest to the middle level and use the x-coordinate minimizing the horizontal edge lengths, which is omitted in pseudocode. Edge blocks are sifted only horizontally, since their levels are determined by the levels of their adjacent vertex blocks.

3.1. Vertical Sifting Step

In a vertical sifting step, we test a vertex block $B \in \mathcal{B}$ at each possible level. To improve the result, we also try to place each vertex block between existing levels. Therefore, we



(c) B on its bottom- and leftmost position.



(d) B on its bottom- and rightmost position.

Fig. 4. Extremal positions of vertex block B during its vertical step (cont.).

normalize (i. e., relabel) the existing levels to $2, 4, 6, \ldots$ (line 1 in Algorithm 2) and then test the vertex block for all level numbers.

To avoid reversing edges, the uppermost possible level $l_{\min}$ for a vertex block is one level below its bottommost preceding vertex block (lines 2–5) or 1 if there is no such block. Likewise, the lowermost possible level $l_{\max}$ is determined (lines 6–9). See Figure 4, where block B is tested with $l_{\min} = 3$, $l_{\max} = 7$ and old level 6. In practice, we confine the number of tested levels above and below the old level of B (i. e., the *level radius*) by a constant to improve the running time.

The algorithm counts only the relative differences between the tested levels, not the total number of crossing. Thus, it first places B at the uppermost possible level

ALGORITHM 1: GRID-SIFTING

Input: Block graph $\mathcal{G} = (\mathcal{B} = \mathcal{B}_V \cup \mathcal{B}_E, \mathcal{F})$ with initial level assignment $\phi : \mathcal{B}_V \to \mathbb{Z}$, number ρ of sifting rounds
Output: Block graph $\mathcal{G}$ with embedding $\mathcal{E}$ given by blocks ordered by values $\pi(B)$ for each $B \in \mathcal{B}$ and updated level assignment ϕ

```
1  initialize π : B → {1, ..., |B|} with random permutation
2  E ← (φ, π)
3  for 1 ≤ i ≤ ρ do
4      foreach B ∈ B_V do
5          E ← VERTICAL-STEP(G, E, B)
6  return (G, E)
```

using VERTICAL-JUMP (line 10). All crossing numbers χ are relative to the number of crossings produced by placing B at the leftmost position on this level. Then, step-by-step, B is swapped downwards to lower levels with VERTICAL-SWAP (Algorithm 3) and the locally calculated relative differences in crossing count Δ are summed up in χ (lines 14–19).

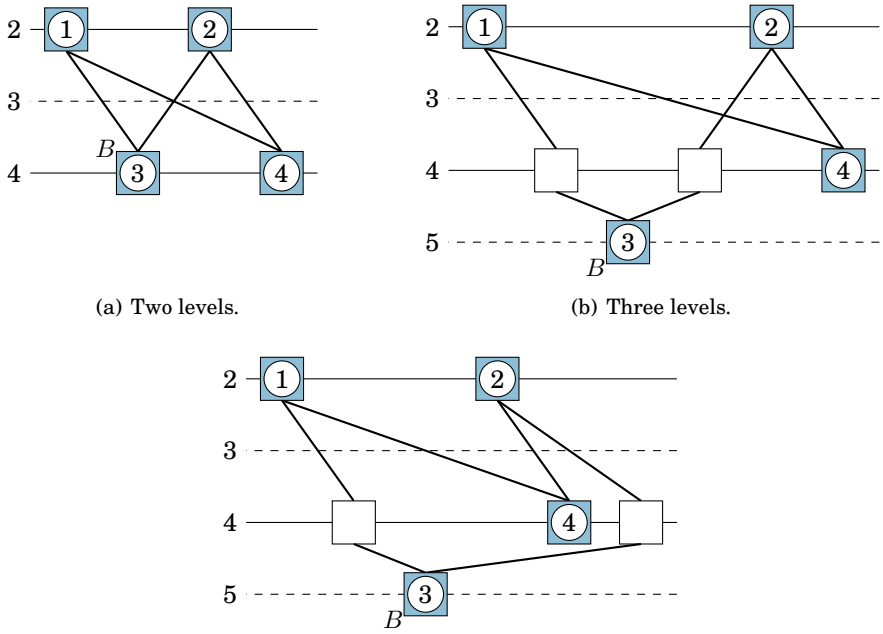
ALGORITHM 2: VERTICAL-STEP($\mathcal{G}, \mathcal{E}, B$)

Input: Block graph $\mathcal{G} = (\mathcal{B}, \mathcal{F})$, embedding $\mathcal{E} = (\phi, \pi)$ with k used levels, vertex block B to sift
Output: Updated embedding $\mathcal{E} = (\phi, \pi)$

```
1  normalize level numbers φ to 2, 4, 6, ..., 2k
2  if N⁻(B) = ∅ then
3      l_min ← 1
4  else
5      l_min ← max_{A∈N⁻(B)} φ(N⁻(A, φ̄(A))[0]) + 1
6  if N⁺(B) = ∅ then
7      l_max ← 2k + 1
8  else
9      l_max ← min_{C∈N⁺(B)} φ(N⁺(C, φ̲(C))[0]) − 1
10 (E, Δ) ← VERTICAL-JUMP(G, E, B, l_min)
11 E_best ← E
12 χ_best ← Δ
13 χ ← Δ
14 for l ← l_min + 1 to l_max do
15     (E, Δ) ← VERTICAL-SWAP(G, E, B, l)
16     χ ← χ + Δ
17     if χ < χ_best then
18         E_best ← E
19         χ_best ← χ
20 return E_best
```

3.2. Vertical Jump and Vertical Swap

In the vertical swap, we move a vertex block B from level $l - 1$ to level l (line 6 in Algorithm 3). Then, we horizontally sift each of the active edge blocks incident to B (lines 12–14) to resolve the crossings. See Figure 5 for an example of $K_{2,2}$s. Afterward,



(a) Two levels.

(b) Three levels.

(c) Edge blocks sifted.

Fig. 5. Resolving the crossing of a $K_{2,2}$ during the vertical swap of block B.

we sift B itself horizontally (line 15) and return the best embedding $\mathcal{E}$ and the sum of crossing changes Δ.

As $\phi(B)$ is modified, previously inactive edge blocks may become active. $\pi(A)$ of such an edge block A has previously been meaningless and may be outdated as A did not correspond to any dummy vertex. To keep track of the number of crossings, we put each edge block becoming active at the horizontal barycenter of its adjacent vertex blocks. The injectivity of π is repaired by bucket-sort. In Figures 4(a) and 4(b) the edge block $\mathrm{block}((2,3))$ is inactive and $\mathrm{block}((3,5))$ is active. In Figures 4(c) and 4(d) it is the other way round.

Moving B to (from) an odd level number means effectively creating (deleting) that level. Surprisingly, this changes the number of crossings even in subgraphs not connected to B, as shown in Figure 6. Hence, a local update is not possible and we use the efficient bilayer crossing counting algorithm by Barth et al. [2004] to count the crossings between these levels. In Figure 6, the vertex block B is moved to level $l = 5$. Hence, to update the number of crossings, we subtract the crossings between levels 2 and 4 as well as levels 4 and 6 before the swap (line 3) and add the crossings between levels 2 and 4, levels 4 and 5, as well as levels 5 and 6 (line 9). Note that some of the numbers of crossings after one vertical swap can be reused in the next one.

In contrast, the vertical jump does not count these changes, as all crossings are relative to the current level $l_{\min}$. Hence, only the lines 1, 6, 7, and 12–17 of Algorithm 3 are executed. One problem created by the vertical jump is that the edge blocks of B might have completely new positions when B arrives at its old level. This impedes optimizing the position of B near its old position. We avoid this by starting the vertical step at the current level twice, swapping upward and downward. For simplicity, We omit this detail in pseudocode.

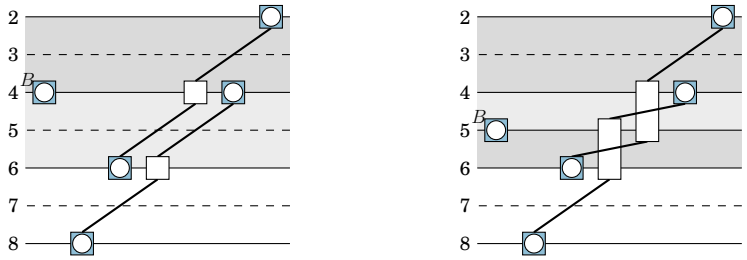


Fig. 6. Modifying $\phi(B)$ changes the crossing count.

ALGORITHM 3: VERTICAL-SWAP($\mathcal{G}, \mathcal{E}, B, l$)

Input: Block graph $\mathcal{G} = (\mathcal{B}, \mathcal{F})$, current embedding $\mathcal{E} = (\phi, \pi)$, vertex block B to sift on level l
Output: Updated embedding $\mathcal{E} = (\phi, \pi)$, change in crossing number Δ

$\Delta \leftarrow 0$
if l is odd **then**
  $\Delta \leftarrow \Delta - \texttt{CROSSINGS}(l-3, l-1) - \texttt{CROSSINGS}(l-1, l+1)$
else
  $\Delta \leftarrow \Delta - \texttt{CROSSINGS}(l-2, l-1) - \texttt{CROSSINGS}(l-1, l) - \texttt{CROSSINGS}(l, l+2)$
$\phi(B) \leftarrow l$
update π by placing each edge block that becomes active at the barycenter of its adjacent vertex blocks and bucket-sorting all blocks
if l is odd **then**
  $\Delta \leftarrow \Delta + \texttt{CROSSINGS}(l-3, l-1) + \texttt{CROSSINGS}(l-1, l) + \texttt{CROSSINGS}(l, l+1)$
else
  $\Delta \leftarrow \Delta + \texttt{CROSSINGS}(l-2, l) + \texttt{CROSSINGS}(l, l+2)$
foreach active edge block $A \in N^-(B) \cup N^+(B)$ **do**
  $(\mathcal{E}, \delta) \leftarrow \texttt{HORIZONTAL-STEP}(\mathcal{G}, \mathcal{E}, A)$
  $\Delta \leftarrow \Delta + \delta$
$(\mathcal{E}, \delta) \leftarrow \texttt{HORIZONTAL-STEP}(\mathcal{G}, \mathcal{E}, B)$
$\Delta \leftarrow \Delta + \delta$
return $(\mathcal{E}, \Delta)$

3.3. Initialization of a Horizontal Sifting Step

As the level assignment ϕ does not change during a horizontal sifting step, the algorithm basically equals the sifting step introduced in global sifting [Bachmaier et al. 2011].

To improve the performance of one horizontal sifting step [Baur and Brandes 2004], it is necessary to keep the adjacency lists $N^-(B, \overline{\phi}(B))$ and $N^+(B, \underline{\phi}(B))$ of each block $B \in \mathcal{B}$ sorted according to ascending positions of the neighboring blocks in $\mathcal{B}$. We store them as arrays for random access. Remember that we do not need to store the l-neighbors for $\overline{\phi}(A) < l < \underline{\phi}(A)$, as $N^{\cdot}(A, l) = \{A\}$.

Additionally, we maintain two index arrays $I^-(B)$ and $I^+(B)$ of lengths $|I^-(B)| := \left|N^-(B, \overline{\phi}(B))\right|$ and $|I^+(B)| := \left|N^+(B, \underline{\phi}(B))\right|$, respectively. $I^-(B)$ stores the indices where B is stored in each adjacent block A's adjacency $N^-(A, \underline{\phi}(A))$. More precisely, let $A = N^+(B, \overline{\phi}(B))[i]$ be a $\overline{\phi}(B)$-neighbor of B in direction "–". Then, $I^-(B)[i]$ holds the index at which B is stored in $N^+(A, \underline{\phi}(A))$. Symmetrically, $I^+(B)$ stores the indices at which

B is stored in the adjacency $N^-(A, \overline{\phi}(A))$ of each adjacent block A. See Figure 4(c) for an example. The creation of the four arrays for each block (line 7 of Algorithm 4) can be done in $\mathcal{O}(|E|)$ time by traversing $\mathcal{B}$ once. See Algorithm 5 for details.

3.4. Sorting and Updating the Adjacencies

With Algorithm 5, we build up the sorted adjacency of each block before each horizontal step. We traverse each block B in the current order of $\mathcal{B}$ and add B to the next free position j of the cleared adjacency array $N^+(A, \underline{\phi}(A))$ ($N^-(C, \overline{\phi}(C))$) of each incoming $\overline{\phi}(B)$-neighbor A (outgoing $\underline{\phi}(B)$-neighbor C). Both values for $I^+(A)$ and $I^-(B)$ ($I^+(B)$ and $I^-(C)$) and their positions are only known after the second traversal of a segment e. Thus, we cache the first array position j as an attribute p of e. Let $v \in V'$ be a dummy vertex of an edge block B with $\phi(v) = l$. We explicitly do not update any outgoing l-neighbor adjacency of v if $\overline{\phi}(B) \leq l < \underline{\phi}(B)$ and no incoming adjacency of v if $\overline{\phi}(B) < l \leq \underline{\phi}(B)$. For performance reasons, these vertices and arrays are only implicit and would only contain one element $N^{\cdot}(B, l)[0] = B$. Thus, associated values "$I^{\cdot}(B, l)$" are not needed in Algorithm 7. For B, only two arrays, $I^-(B)$ and $I^+(B)$, remain, representing the position of B in its incident vertex blocks.

3.5. Horizontal Sifting Step

In a horizontal sifting step (Algorithm 4), all positions i in $\mathcal{B}$ are tested for a (vertex or edge) block $B \in \mathcal{B}$ (lines 8–15) and the best embedding $\mathcal{E}_{\text{best}}$ is returned. Similar to the vertical step, we only compute the change in the number of crossings when swapping A iteratively with its right neighbor (line 11). To be able to return the change in the number of crossings (line 16), we store the relative number of crossings that B causes at its old position p (lines 9, 10).

ALGORITHM 4: HORIZONTAL-STEP($\mathcal{G}, \mathcal{E}, B$)

Input: Block graph $\mathcal{G} = (\mathcal{B}, \mathcal{F})$, embedding $\mathcal{E} = (\phi, \pi)$, block B to sift
Output: Updated ordering π, change in crossing number Δ

```
1  p ← π(B)
2  E_best ← E
3  Δ_best ← ∞
4  Δ ← 0
5  Δ_old ← 0
6  place B at first position of B
7  SORT-ADJACENCIES (G, π)
8  for i ← 1 to |B| − 1 do
9      if i = p then
10         Δ_old ← Δ
11     (E, δ) ← HORIZONTAL-SWAP (G, E, B, π^{-1}(i + 1))
12     Δ ← Δ + δ
13     if Δ < Δ_best then
14         E_best ← E
15         Δ_best ← Δ
16 return (E_best, Δ_best − Δ_old)
```

ALGORITHM 5: SORT-ADJACENCIES($\mathcal{G}, \pi$)

Input: Block graph $\mathcal{G} = (\mathcal{B}, \mathcal{F})$, ordering π
Output: Ordered sets $N^-(B, \overline{\phi}(B))$, $N^+(B, \underline{\phi}(B))$, $I^{\cdot}(B)$ for each block $B \in \mathcal{B}$

```
1  for i ← 0 to |B| − 1 do
2      π(B[i]) ← i
3      clear arrays N⁻(B[i], φ̄(B[i])), N⁺(B[i], φ̲(B[i])) and I·(B[i])
4  foreach vertex block or active edge block B ∈ B do
5      foreach e = (A, B) ∈ F do
6          if A is an inactive edge block then
7              A ← A′ with unique (A′, A) ∈ F
8          add B to the next free position j of N⁺(A, φ̲(A))
9          if π(B) < π(A) then
10             p[e] ← j                          // first traversal of e
11         else
12             I⁺(A)[j] ← p[e]
13             I⁻(B)[p[e]] ← j                   // second traversal of e
14     foreach e = (B, C) ∈ F do
15         if C is an inactive edge block then
16             C ← C′ with unique (C, C′) ∈ F
17         add B to the next free position j of N⁻(C, φ̄(C))
18         if π(B) < π(C) then
19             p[e] ← j                          // first traversal of e
20         else
21             I⁺(B)[p[e]] ← j
22             I⁻(C)[j] ← p[e]                   // second traversal of e
```

3.6. Horizontal Sifting Swap

The horizontal sifting swap is the actual computation of the change in the number of crossings when a block A is swapped with its right neighbor B. Lemma 3.1 states which segments are involved and Proposition 3.2 states how the number of crossings changes on such a level. Both can also be found in Bachmaier et al. [2011].

Algorithm 6 shows the details of a horizontal sifting swap. First, the levels at which (significant) swaps occur and the direction of the segments changing their crossings are found (lines 7–11). For each entry (l, d) of the set $\mathcal{L}$, the l-neighbors of A and B in direction d are retrieved. Using the notion of G', only the consecutive (dummy) vertices in A and B on level l are swapped. The permutation of the neighboring level $\mathrm{next}^d(l)$ remains unchanged. Thus, the computation of the change in the number of crossings among segments between l and $\mathrm{next}^d(l)$ can be done as in Baur and Brandes [2004], which we adapt to our notation (lines 16–31): The l-neighbors are traversed from left to right. If an l-neighbor of A is found (line 21), the corresponding segment will cross all remaining $s - j$ incident/inner segments of B after the swap. If an l-neighbor of B is found (line 24), the segment has crossed all remaining $r - i$ incident/inner segments of A before the swap. Common neighbors present both cases at the same time (line 30). An update of the adjacency after a swap (line 14) is only necessary if A and B have common l-neighbors. Algorithm 7 shows how this can be done in overall $\mathcal{O}(\deg(A)+\deg(B))$ time similarly to the function uswap.

ALGORITHM 6: HORIZONTAL-SWAP($\mathcal{G}, \mathcal{E}, A, B$)

Input: Block graph $\mathcal{G} = (\mathcal{B}, \mathcal{F})$, embedding $\mathcal{E} = (\phi, \pi)$, consecutive blocks A, B
Output: Updated embedding $\mathcal{E}$, change in crossing count

```
1  begin
2      π' ← π
3      π'(A) ← π(B)
4      π'(B) ← π(A)
5      E ← (φ, π')
6      if B ∈ B_E ∧ B is not active then return (E, 0)
7      L ← ∅; Δ ← 0                          // L is duplicate free
8      if φ̄(A) ∈ levels(B) then L ← L ∪ {(φ̄(A), −)}
9      if φ(A) ∈ levels(B) then L ← L ∪ {(φ(A), +)}
10     if φ̄(B) ∈ levels(A) then L ← L ∪ {(φ̄(B), −)}
11     if φ(B) ∈ levels(A) then L ← L ∪ {(φ(B), +)}
12     foreach (l, d) ∈ L do
13         Δ ← Δ + uswap(A, B, l, d)
14         UPDATE-ADJACENCIES(A, B, l, d)
15     return (E, Δ)

16 function uswap(A, B, l, d): integer
17     let X_0 ≺ ··· ≺ X_{r−1} ∈ N^d(A, l) be the l-neighbors of A in direction d
18     let Y_0 ≺ ··· ≺ Y_{s−1} ∈ N^d(B, l) be the l-neighbors of B in direction d
19     c ← 0; i ← 0; j ← 0
20     while i < r and j < s do
21         if π(X_i) < π(Y_j) then
22             c ← c + (s − j)
23             i ← i + 1
24         else if π(X_i) > π(Y_j) then
25             c ← c − (r − i)
26             j ← j + 1
27         else
28             c ← c + (s − j) − (r − i)
29             i ← i + 1
30             j ← j + 1
31     return c
```

LEMMA 3.1. *Let $\mathcal{B}$ be the block list in the current order. Let $B \in \mathcal{B}$ be the successor of $A \in \mathcal{B}$. If swapping A and B changes the crossings between any two segments, then one of them is an incident outer segment of A or B. The other segment is an incident outer segment of the same kind (incoming or outgoing) of the other block or an inner segment of the other block.*

PROOF. Note that only segments between the same levels can cross. As type 2 conflicts are absent, at least one of the segments of a crossing is an outer segment. Let $(u, v), (w, z) \in E'$ be two segments between the same levels with $u \neq w$ and $v \neq z$. If the two segments cross after swapping A and B but did not cross before (or vice versa), either u and w or v and z were swapped. Therefore, one of the segments is adjacent to A or is a part of A and the other is adjacent to B or is a part of B. If v and z were swapped and thus u and w were not, $\phi(v) = \phi(z)$ is the upper level of A or B, and thus one of the crossing segments is an incoming outer segment of A or B. The other segment is either an incoming outer segment or an inner segment of the other block. Note that it

cannot be an outgoing outer segment of this block because then neither u and w nor v and z would have been swapped. The other case of swapping u and w instead of v and z is symmetric. □

PROPOSITION 3.2. *Let $\mathcal{B}$ be the block list in the current order. Let $B \in \mathcal{B}$ be the successor of $A \in \mathcal{B}$. Let i and j be the two levels framing the incoming outer segments of A, the other three cases are symmetric. If there is a segment (u, v) between i and j, which is either an incoming outer segment of B or an inner segment of B, then the incoming segments of A starting at a block left of $\mathrm{block}(u)$ cross (u, v) after the swap of A and B only, the segments starting at $\mathrm{block}(u)$ never cross (u, v), and the segments starting right of $\mathrm{block}(u)$ cross (u, v) before the swap only. There are no other changes of crossings due to Lemma 3.1.*

After a horizontal swap of two blocks A and B, we adjust the adjacency arrays of common neighbors with Algorithm 7.

ALGORITHM 7: UPDATE-ADJACENCIES(A, B, l, d)

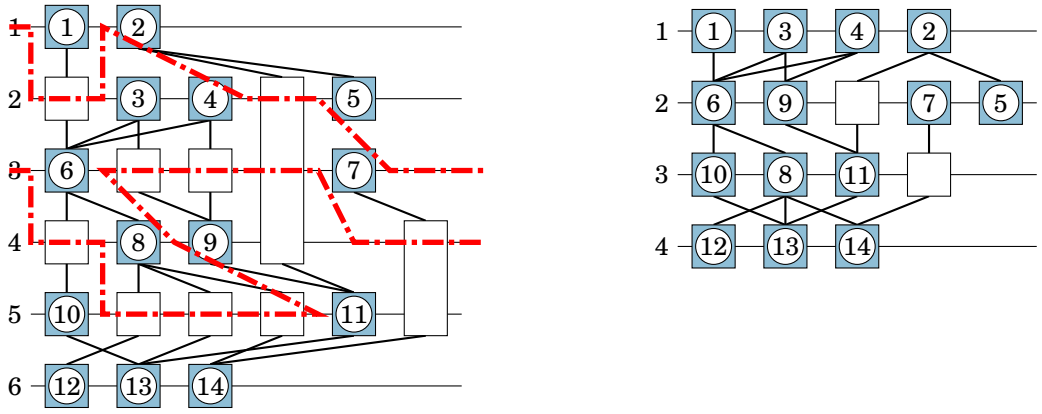
Input: Consecutive blocks $A, B \in \mathcal{B}$, level l, direction d $N^d(A, l), I^d(A), N^d(B, l), I^d(B)$
Output: Updated adjacencies of A and B and all common neighbors

```
1  let X_0 ≺ ··· ≺ X_{r-1} ∈ N^d(A, l) be the l-neighbors of A in direction d
2  let Y_0 ≺ ··· ≺ Y_{s-1} ∈ N^d(B, l) be the l-neighbors of B in direction d
3  i ← 0; j ← 0
4  while i < r and j < s do
5      if π(X_i) < π(Y_j) then
6          i ← i + 1
7      else if π(X_i) > π(Y_j) then
8          j ← j + 1
9      else
10         Z ← X_i                                  // = Y_j
11         swap entries at pos. I^d(A)[i] and I^d(B)[j] in N^{-d}(Z, l) and in I^{-d}(Z)
12         I^d(A)[i] ← I^d(A)[i] + 1
13         I^d(B)[j] ← I^d(B)[j] − 1
14         i ← i + 1
15         j ← j + 1
```

3.7. Time Complexity

THEOREM 3.3. *One round of grid sifting (Algorithm 1) has a time complexity of $\mathcal{O}(|E|^2 + |E| \cdot |V| \cdot \log |V|)$ for a nonnecessarily proper level graph $G = (V, E, \phi)$.*

PROOF. A horizontal sifting step of a block B needs $\mathcal{O}(|E| \cdot \deg(B))$ time [Bachmaier et al. 2011]. Hence, (horizontally) sifting an edge block takes $\mathcal{O}(|E|)$ time, as its degree is 2. A vertical swap of vertex block B consists of counting the crossings between five level pairs ($\mathcal{O}(|E| \cdot \log |E|)$, [Barth et al. 2004]), (de-)activating edge blocks ($\mathcal{O}(|E|)$), a horizontal sifting step for each incident edge block ($\mathcal{O}(|E| \cdot \deg(B))$ in total), and the horizontal sifting step of B itself ($\mathcal{O}(|E| \cdot \deg(B))$). We fix the level radius (i. e., the number of tested levels in each direction) to a constant. Thus, we obtain the time complexity $\mathcal{O}(|E| \cdot \deg(B) + |E| \cdot \log |E|)$ for a vertical step. A sifting round consists of a vertical step of each vertex block and has time complexity $\mathcal{O}(\sum_{B \in \mathcal{B}_V} (|E| \cdot \deg(B) + |E| \cdot \log |E|)) = \mathcal{O}(|E|^2 + |E| \cdot |V| \cdot \log |V|)$, since $\mathcal{O}(\log |E|) = \mathcal{O}(\log |V|)$. □



(a) Cuts from left to right. (b) Compacted.

Fig. 7. Vertical compaction.

The time complexity is $\mathcal{O}(|E|^2)$ for dense graphs with $|E| \geq |V| \log |V|$. Our experiments show that the counting of the crossings can be neglected in practice. The time complexity increases to $\mathcal{O}(|E|^2 \cdot |V| + |E| \cdot |V|^2 \cdot \log |V|)$ in total using an unfixed level radius.

4. VERTICAL COMPACTION

Similarly to Chimani et al. [2011], we apply a postprocessing step to reduce the number of levels without changing the crossing number. In the level embedding, we search for distinct (nonnecessarily monotonic) cuts from the left to the right which consist solely of dummy vertices (at most one of each edge block) and do not cross outer segments. We delete these cuts (i. e., its vertices) and lift the subgraph below each cut by one level. See Figure 7 for an example. Even our naïve implementation needs less than 5% of the overall running time in our benchmarks.

5. EXPERIMENTAL RESULTS

All of the following benchmarks ran on Intel Xeon 2.0 GHz cluster nodes each with 16GB main memory. The upward planarization layout (UPL) algorithm was a binary C++ executable (provided by the authors of [Chimani et al. 2011]) and the other algorithms were implemented in JAVA within Gravisto [Bachmaier et al. 2004]. All tests in sum needed a (sequential) computation time of roughly 219 days.

5.1. Synthetic Benchmarks

For each density $d = \frac{|E|}{|V|} \in \{1.5, 2.5, \ldots, 10.5\}$ and vertex number $|V| \in \{20, 40, \ldots, 400\}$, we generated 4 virtually random graphs with $|V|$ vertices and an edge set chosen from all $d \cdot |V|$-element subsets of the edge set of the complete graph [Rodionov and Choo 2004]. More specifically, we identified the vertices with numbers from 1 to $|V|$ and introduced the set E' containing all directed edges from lower numbered to higher numbered vertices. Thus, (V, E') is a complete directed acyclic graph. Then, we partitioned the vertices into two sets $V_1 \dot{\cup} V_2$, initially with $V_1 = \{1\}$ and $V_2 = V - V_1$. We then iteratively chose a random edge $e = (u, v) \in E'$ with $u \in V_1$ and $v \in V_2$ and set $E \leftarrow E \cup \{e\}$, $E' \leftarrow E' - \{e\}$, $V_1 \leftarrow V_1 \cup \{v\}$, and $V_2 \leftarrow V_2 - \{v\}$ until $V_2 = \varnothing$. Thus, (V, E) is connected.

Then, we iteratively chose a random edge e from the remaining edges in E' and set $E' \leftarrow E' - \{e\}$ and $E \leftarrow E \cup \{e\}$. In the latter step, whenever $|E| = d \cdot |V|$, we stopped and obtained the benchmark graph (V, E).[2]

We used the state of the art Coffman/Graham algorithm with level width 1, yielding a virtually random injective initial leveling $\phi : V \to \{1, \ldots, |V|\}$, in order to give the algorithms some freedom to place vertices. As a side effect, ϕ also imposes an ascending topological ordering on the vertices. We compared the algorithms iterative one-sided 2-level barycenter (BC) [Kaufmann and Wagner 2001], global sifting (GlS) [Bachmaier et al. 2011], upward planarization layout (UPL) [Chimani et al. 2011], and our grid sifting algorithm with the level radii 3 (GrS3), 10 (GrS10) and unconfined (GrS*). GrS21 uses radius 21 but odd (i. e., new) levels only. Hence, GrS21 holds the invariant of having only one vertex per level. Then, the new levels adjacent to the current level of each vertex can be omitted as well, and GrS21 tests the same number of levels as GrS10.

We executed eight rounds for each grid sifting variant, since then there is no further significant improvement in the number of crossings. We applied UPL 20 times on each input to choose the best result. Clearly, BC and GlS are the fastest algorithms by far, but they cannot change the leveling. To ensure that their results do not suffer from saving running time, we execute (actually unreasonable) 400 sweeps and sifting rounds, respectively.

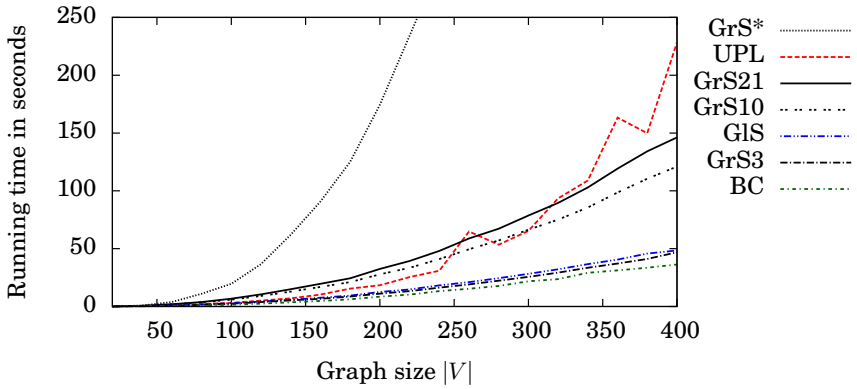
As UPL is slower than our grid sifting algorithms (Figure 10), we could not test UPL for graphs with a density greater than 4.5 and more than 200 vertices. The running time (Figure 8) of the grid sifting variants with restricted level radius are rather comparable to GlS, whereas BC is always fastest.

Note the crossings of the lines GrS10, GrS21, and GlS in Figure 8(c), which, at first glance, contradict the same asymptotic runtime of $\mathcal{O}(|E|^2)$. Compared to GrS10, GrS21 uses the double level radius, that is, it tries to place vertices at levels that are up to double the distance of their original level, but it tests only odd levels. Consequently, GrS10 and GrS21 perform the same number of horizontal steps in large graphs. Then, GrS10 is slightly slower because of the different relative amount of runtime required by counting the bilayer crossings. However, as in small graphs the number of actually tested levels is effectively restricted by the total number of levels rather than the level radius, GrS21 is faster than GrS10 for graphs with less than 80 vertices. A similar argument applies to GlS, which corresponds to grid sifting with level radius 0, that is, it omits vertical swaps. GrS21 is faster than GlS for small graphs by performing less rounds, but it is slower once the number of actually tested levels converges to 21 as the graph size increases. As there are no graphs with 20 vertices and a density of 10.5, the benchmarks in Figure 8(c) start at 40.

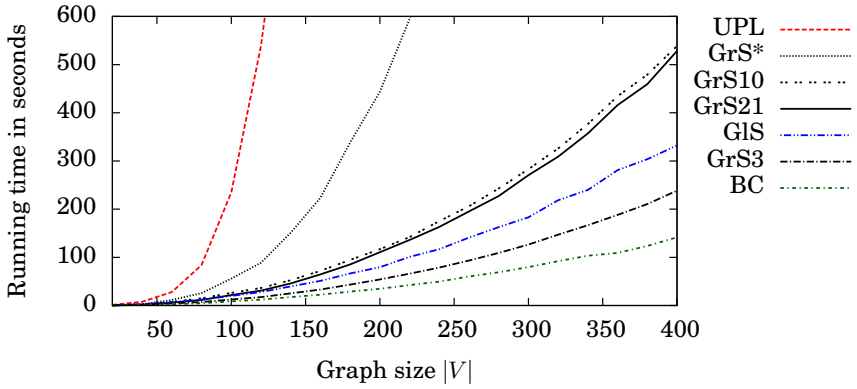
Figure 9 shows the obtained quality of the algorithms in terms of crossings at different densities. As expected, the higher the level radius is chosen, the less crossings remain. Compared to that, an increased number of rounds for GlS does not pay out. BC is qualitatively inferior.

Figures 10–12 compare the performance of the algorithms with the density of the graphs. Clearly, the values shown in Figure 10 are dominated by the running time for the large instances. For a similar benchmark using graphs with a fixed number of vertices, see Figure 13. As a further consequence of the absence graphs with 20 vertices

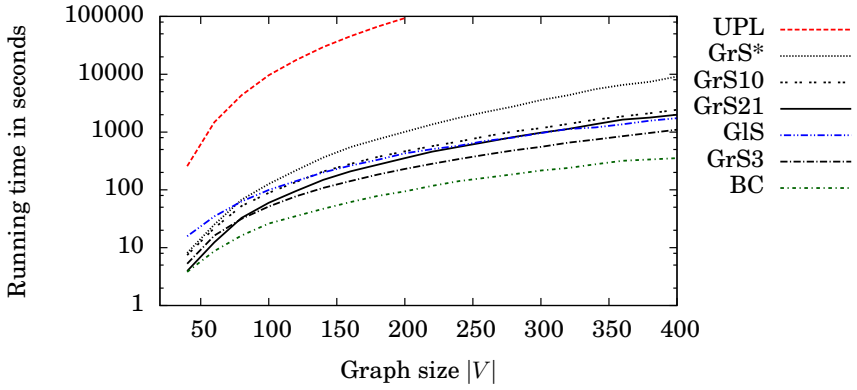
[2]The generators used are included within the source code of Gravisto [Bachmaier et al. 2004]. The primary Java classes are `org.graffiti.plugins.algorithms.sugiyama.gridsifting.benchmark.RandomPA` and `RandomPA2`, which contain further instructions for their usage. The employed graphs generated with a random (but for reproducibility hard-coded) seed can be obtained from the Graph Archive [Bachmaier et al. 2012] under keyword “Grid Sifting”.



(a) Density 1.5.

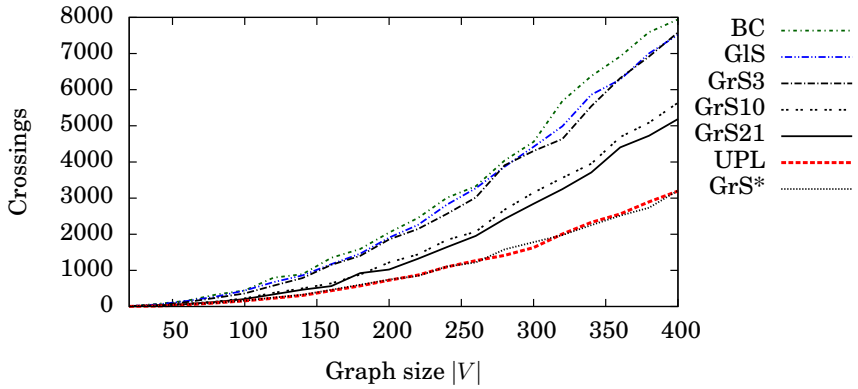


(b) Density 4.5.

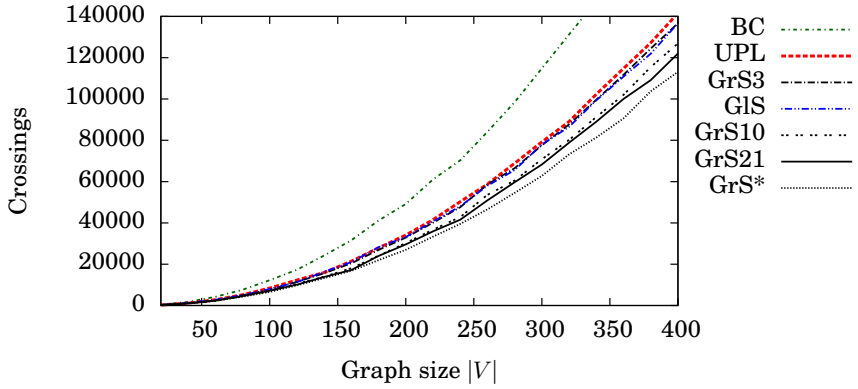


(c) Density 10.5.

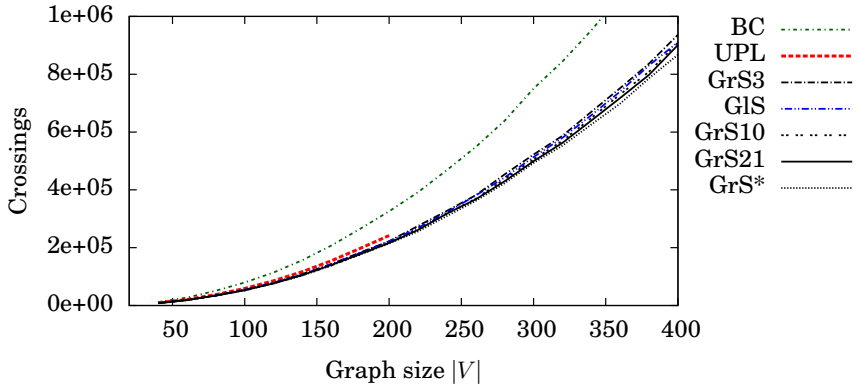
Fig. 8. Random graphs: number of vertices vs. running times.



(a) Density 1.5.



(b) Density 4.5.



(c) Density 10.5.

Fig. 9. Random graphs: number of vertices vs. number of crossings.

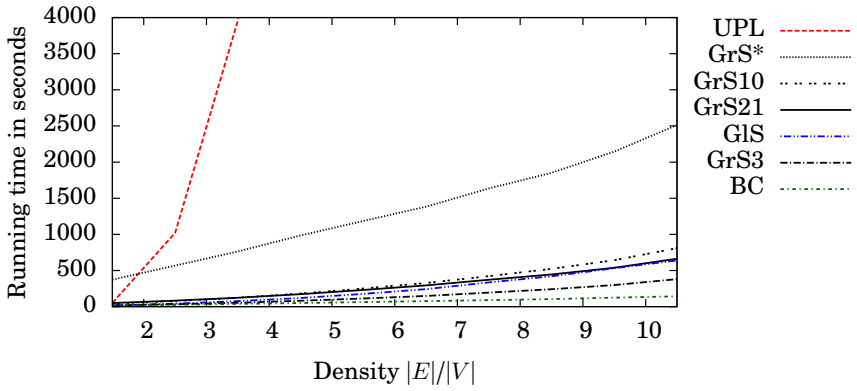


Fig. 10. Random graphs: density vs. running times.

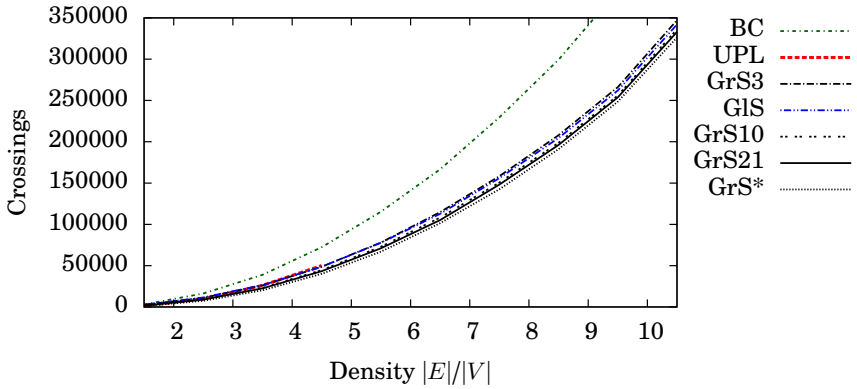


Fig. 11. Random graphs: density vs. number of crossings.

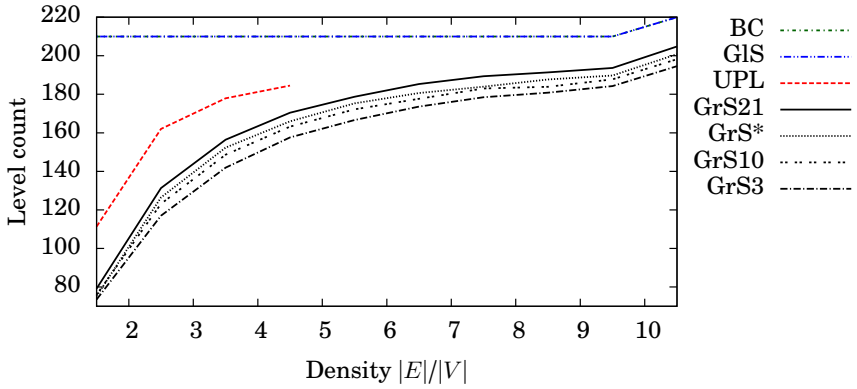


Fig. 12. Random graphs: density vs. number of levels.

and a density of 10.5, the average vertex number of the tested graphs is higher for that density. This causes the bend in the lines in Figure 12.

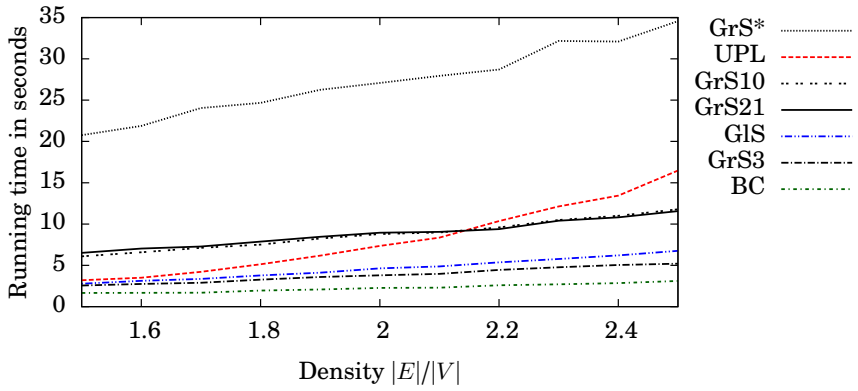


Fig. 13. Random graphs with 100 vertices: density vs. running times.

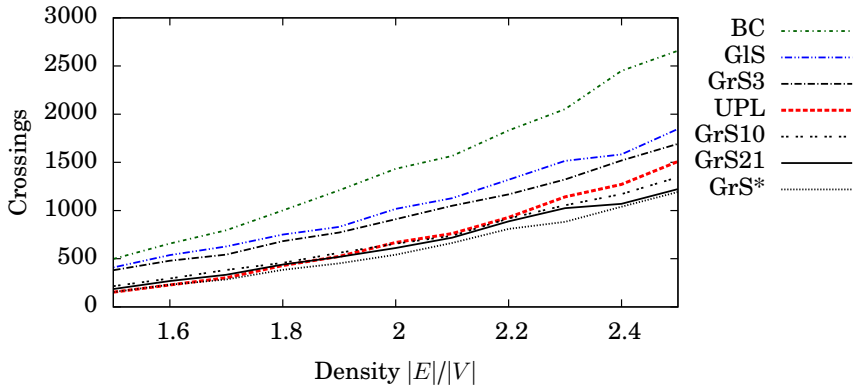


Fig. 14. Random graphs with 100 vertices: density vs. number of crossings.

In Figure 11, yet at a density of 2.5, all grid sifting variants give less crossings than UPL, with throughout a lower number of levels (Figure 12). GrS21 is a good trade-off between running time and result. The Wilcoxon signed rank test [Wilcoxon 1945] indicates with a confidence of 99% that GrS21 yields 8.1% less crossings than UPL for graphs with density 2.5, 13.1% for density 4.5, and 9.1% for density 10.5. Aggregating all tested densities, GrS21 yields 11.8% less crossings than UPL. The number of levels is in both algorithms rather high (Figure 12), which is a consequence of optimizing the number of crossings first.

To further shed light on the performance of the algorithms at comparatively low densities, we tested another set of random graphs. This time we fixed the number of vertices of each graph to 100, following the style of benchmarks for random graphs of Chimani et al. [2011]. We drew 10 instances each from all connected DAGs with an exact density of $d = \frac{|E|}{100} \in \{1.5, 1.6, \ldots, 2.4, 2.5\}$. The tests clearly indicate that while UPL wins at sparsest graphs, GrS21 dominates other approaches regarding runtime (Figure 13) and crossings (Figure 14) for all densities above a certain threshold (about 2.18 in the very case of 100 vertices).

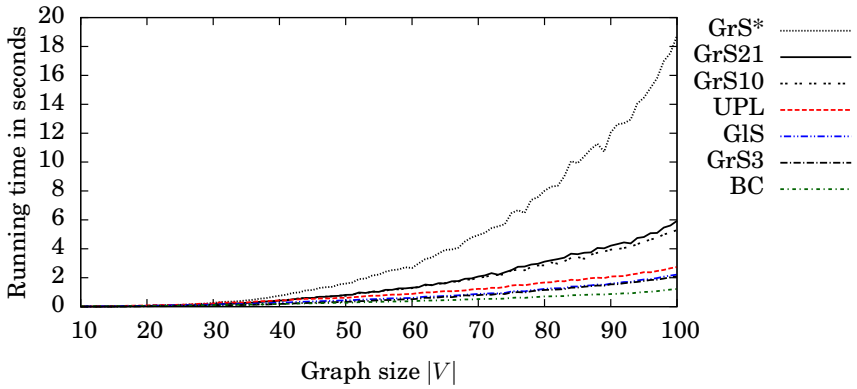


Fig. 15. Rome graphs: number of vertices vs. running times.

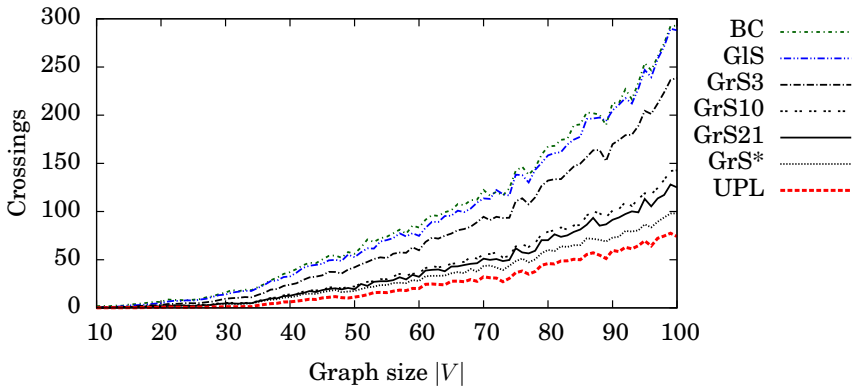


Fig. 16. Rome graphs: number of vertices vs. number of crossings.

5.2. Rome Graphs

Analogously to Chimani et al. [2011], we additionally compare the algorithms on the widely used Rome graphs library [Di Battista et al. 1997] in Figures 15–17. The set contains 11,528 instances with 10–100 vertices and 9–158 edges. Although these graphs are originally undirected, we interpret them as directed by artificially directing the edges according to the vertex order given in the input files from former to later vertex definitions (see Chimani et al. [2011]). The remaining parameter set-up is identical to Section 5.1.

As expected for the Rome graphs, which are, with an average density of about 1.3, very sparse and almost planar, the planarization approach UPL results in fewer crossings than grid sifting. However, grid sifting produces fewer levels. The running times are comparable.

5.3. Example Drawings

As examples, we visualize the embeddings of two Rome graphs [Di Battista et al. 1997] computed by grid sifting. The edge directions are determined as described in Section 5.2. For the fourth phase of the framework, that is, for computing real coordinates, we employed the algorithm of Brandes and Köpf [2002].

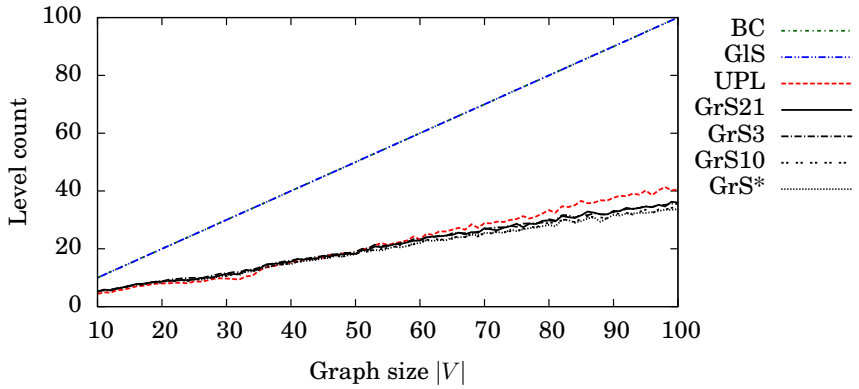
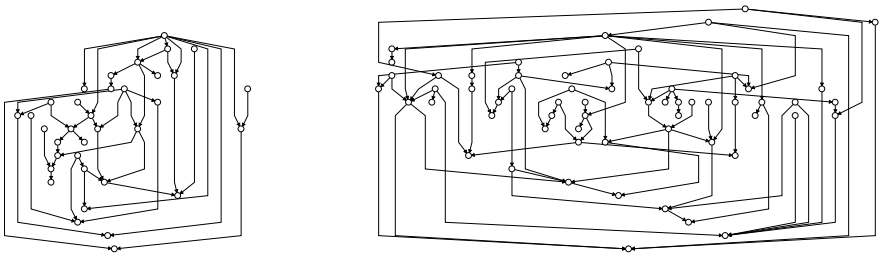


Fig. 17. Rome graphs: number of vertices vs. number of levels.



(a) grafo2379.35 with 35 vertices and 45 edges on 17 levels with 6 crossings.

(b) grafo4562.60 with 60 vertices and 79 edges on 19 levels with 27 crossings.

Fig. 18. Drawings of two Rome graphs produced by grid sifting.

6. SUMMARY

Combining the leveling and crossing reduction phases of the Sugiyama framework delivers fewer crossings. The planarization approach by Chimani et al. [2010; 2011] and the presented two-dimensional grid sifting follow this idea. The latter delivers similar results regarding levels and crossings to the former, however, its time complexity is roughly $\mathcal{O}(|E|^2)$ instead of $\mathcal{O}(|E|^5)$. We suggest using planarization for very sparse and grid sifting for more dense graphs. While the basic strategy of moving vertices across levels during the sifting is easy to perceive, in detail it turns out to be tricky to maintain a decent time complexity. Therefore, we provided a comparatively extensive description of the algorithm, which makes a straightforward implementation possible. Our implementation is available under the GPL within the Gravisto framework [Bachmaier et al. 2004].

REFERENCES

BACHMAIER, C., BRANDENBURG, F. J., BRUNNER, W., AND HÜBNER, F. 2011. Global k-level crossing reduction. *J. Graph Algor. Appl. 15,* 5, 631–659.

BACHMAIER, C., BRANDENBURG, F. J., EFFINGER, P., GUTWENGER, C., KATAJAINEN, J., KLEIN, K., SPÖNEMANN, M., STEGMAIER, M., AND WYBROW, M. 2012. The open graph archive: A community-driven effort. In *Proceedings of the International Conference on Graph Drawing, GD'11*. Lecture Notes in Computer Science Series, vol. 7034. Springer, 435–440. `http://www.graphdrawing.org/grapharchive/`.

BACHMAIER, C., BRANDENBURG, F. J., FORSTER, M., HOLLEIS, P., AND RAITNER, M. 2004. Gravisto: Graph visualization toolkit. In *Proceedings of the International Conference on Graph Drawing, GD'04*. Lecture Notes in Computer Science Series, vol. 3383. Springer, 502–503. http://gravisto.fim.uni-passau.de/.

BARTH, W., MUTZEL, P., AND JÜNGER, M. 2004. Simple and efficient bilayer cross counting. *J. Graph Algor. Appl. 8,* 2, 179–194.

BAUR, M. AND BRANDES, U. 2004. Crossing reduction in circular layout. In *Proc. Workshop in Graph-Theoretic Concepts in Computer Science (WG'04)*. Lecture Notes in Computer Science Series, vol. 3353. Springer, 332–343.

BRANDES, U. AND KÖPF, B. 2002. Fast and simple horizontal coordinate assignment. In *Proceedings of the International Conference on Graph Drawing, GD'01*. Lecture Notes in Computer Science Series, vol. 2265. Springer, 31–44.

CHIMANI, M., GUTWENGER, C., MUTZEL, P., AND WONG, H.-M. 2010. Layer-free upward crossing minimization. *ACM J. Exp. Algor. 15*, 2.2.1–2.2.27.

CHIMANI, M., GUTWENGER, C., MUTZEL, P., AND WONG, H.-M. 2011. Upward planarization layout. *J. Graph Algor. Appl. 15,* 1, 127–155.

COFFMAN, E. G. J. AND GRAHAM, R. L. 1972. Optimal scheduling for two processor systems. *Acta Informatica 1,* 3, 200–213.

DI BATTISTA, G., GARG, A., LIOTTA, G., TAMASSIA, R., TASSINARI, E., AND VARGIU, F. 1997. An experimental comparison of four graph drawing algorithms. *Comput. Geom. Theory Appl. 7,* 5–6, 303–325. Graphs available at http://www.graphdrawing.org/.

EADES, P. AND WORMALD, N. C. 1994. Edge crossings in drawings of bipartite graphs. *Algorithmica 11,* 1, 379–403.

ESCHBACH, T., GÜNTHER, W., DREXLER, R., AND BECKER, B. 2002. Crossing reduction by windows optimization. In *Proceedings of the International Conference on Graph Drawing, GD'02*. Lecture Notes in Computer Science Series, vol. 2528. Springer, 285–294.

GANSNER, E. R., KOUTSOFIOS, E., NORTH, S., AND VO, K.-P. 1993. A technique for drawing directed graphs. *IEEE Trans. Software Eng. 19,* 3, 214–230.

JÜNGER, M., LEE, E. K., MUTZEL, P., AND ODENTHAL, T. 1997. A polyhedral approach to the multilayer crossing minimization problem. In *Proceedings of the International Conference on Graph Drawing, GD'97*. Lecture Notes in Computer Science Series, vol. 1353. Springer, 13–24.

KAUFMANN, M. AND WAGNER, D. 2001. *Drawing Graphs*. Lecture Notes in Computer Science Series, vol. 2025. Springer.

KUNTZ, P., PINAUD, B., AND LEHN, R. 2006. Minimizing crossings in hierarchical digraphs with a hybridized genetic algorithm. *J. Heuristics 12,* 1–2, 23–26.

LAGUNA, M., MARTÍ, R., AND VALLS, V. 1997. Arc crossing minimization in hierarchical digraphs with tabu search. *Comput. Oper. Res. 24,* 12, 1175–1186.

MATUSZEWSKI, C., SCHÖNFELD, R., AND MOLITOR, P. 1999. Using sifting for k-layer straightline crossing minimization. In *Proceedings of the International Conference on Graph Drawing, GD'99*. Lecture Notes in Computer Science Series, vol. 1731. Springer, 217–224.

RODIONOV, A. S. AND CHOO, H. 2004. On generating random network structures: Connected graphs. In *Proc. International Conference on Information Networking, ICOIN'04*. Lecture Notes in Computer Science Series, vol. 3090. Springer, 483–491.

RUDELL, R. 1993. Dynamic variable ordering for ordered binary decision diagrams. In *Proc. IEEE/ACM International Conference on Computer Aided Design, ICCAD'93*. IEEE, Los Alamitos, CA, 42–47.

SUGIYAMA, K., TAGAWA, S., AND TODA, M. 1981. Methods for visual understanding of hierarchical system structures. *IEEE Trans. Syst., Man, Cybern. 11,* 2, 109–125.

UTECH, J., BRANKE, J., SCHMECK, H., AND EADES, P. 1998. An evolutionary algorithm for drawing directed graphs. In *Proc. International Conference on Imaging Science, Systems, and Technology, CISST'98*. CSREA, 154–160.

WILCOXON, F. 1945. Individual comparisons by ranking methods. *Biometrics Bull. 1,* 6, 80–83.

Received September 2011; revised June 2012; accepted July 2012