A Fast Layout Algorithm for k-Level Graphs

Christoph Buchheim, Michael Jünger, and Sebastian Leipert

Universität zu Köln, Institut für Informatik,
Pohligstraße 1, 50969 Köln, Germany
{buchheim,mjuenger,leipert}@informatik.uni-koeln.de

Abstract. We present a fast layout algorithm for k-level graphs with given permutations of the vertices on each level. The algorithm can be used in particular as a third phase of the Sugiyama algorithm [8]. In the generated layouts, every edge has at most two bends and is drawn vertically between these bends. The total length of short edges is minimized levelwise.

1 Introduction

When displaying hierarchical network structures, one usually has a partition of the vertices into k levels such that in the drawing all vertices of a common level are required to receive the same y-coordinate. This leads to the concept of k-level graphs, that is also used to draw arbitrary graphs in the algorithm of Sugiyama et al. [8]. This algorithm serves as a frame for many graph drawing algorithms, processing a graph in three phases. In a first phase, the vertices are assigned to levels $1, \dots, k$, thus transforming the graph into a k-level graph. In the second phase, the number of edge crossings is reduced by permuting the vertices within the levels. Finally, a nice layout based on the results of the previous phases has to be determined, assigning y-coordinates to the levels and x-coordinates to the vertices and edge bends.

The first two phases of the Sugiyama algorithm have been examined intensively, see e.g. [5] for the crossing minimization, while the third phase has only been studied rarely, see e.g. [3] or [7]. In this paper, we present a new algorithm LEVEL_LAYOUT for the third phase. Every edge that traverses more than one level is drawn vertically except for its outermost segments. This improves readability (compare Fig. 1 and Fig. 2). Furthermore, the total length of short edges is minimized levelwise as described in Sect. 5.2. If the k-level graph is connected, LEVEL_LAYOUT performs in $O(\overline{m}(\log \overline{m})^2)$, where $\overline{m}$ is the number of edge segments in the k-level graph, i.e., the number of edges after introducing virtual vertices wherever an edge crosses a level. An implementation of the algorithm is contained in the AGD-Library [6].

2 Preliminaries

A *graph* G is a pair (V, E) where V is an arbitrary finite set and E is a subset of $\{\{v, w\} \mid v, w \in V,\ v \neq w\}$. Elements of V and E are called *vertices* and *edges*,

J. Marks (Ed.): GD 2000, LNCS 1984, pp. 229–240, 2001.

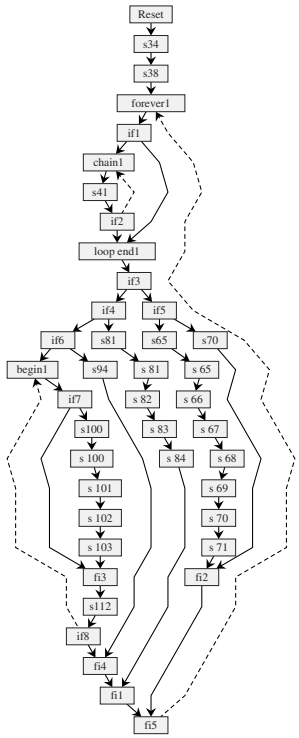


Fig. 1. Layout of an embedded 25-level graph generated by a straightforward algorithm

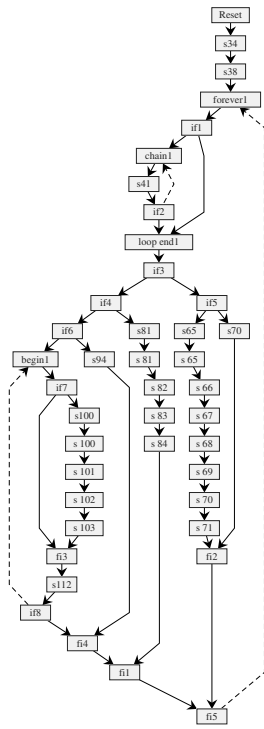


Fig. 2. The same graph drawn by LEVEL_LAYOUT. Every edge has at most two bends

respectively. We usually denote an edge $\{v, w\}$ by (v, w). For a vertex $v \in V$, $\delta_G(v) = \{w \in V \mid (v, w) \in E\}$ is the set of its *neighbors*. For any nonnegative integer k, a *k-level graph* $G = (V, E, \lambda)$ is a graph $G = (V, E)$ equipped with a mapping $\lambda : V \to \{1, \ldots, k\}$ such that $\lambda(v) \neq \lambda(w)$ for every edge $(v, w) \in E$. If $v \in V$ is a vertex, $\lambda(v)$ is called the *level* of v.

An edge $e = (v, w)$ is called *short* if $|\lambda(v) - \lambda(w)| = 1$, otherwise *long*. Let e be a long edge and assume that $\lambda(w) > \lambda(v)$. We introduce a *virtual* vertex $\overline{v}_l$ for every level $l \in \{\lambda(v) + 1, \ldots, \lambda(w) - 1\}$ and set $\lambda(\overline{v}_l) = l$. We split up e into *edge segments* $(v, \overline{v}_{\lambda(v)+1}), (\overline{v}_{\lambda(v)+1}, \overline{v}_{\lambda(v)+2}), \ldots, (\overline{v}_{\lambda(w)-1}, w)$. Applying this to every long edge, we obtain a set of virtual vertices, disjoint from V, which is denoted by $\overline{V}$. Furthermore, we obtain a set $\overline{E}$ of edge segments. Obviously, this yields a new k-level graph $\overline{G} = (V \cup \overline{V}, \overline{E}, \lambda)$ without long edges. For the following, let $\delta = \delta_G$ and $\overline{\delta} = \delta_{\overline{G}}$. We will call the vertices $v \in V$ *original* vertices to distinguish them from virtual vertices. An edge segment is called *outer segment* if it is incident to an original vertex, otherwise it is called *inner segment*.

A *level embedding* of a k-level graph is a mapping that assigns to each $l \in \{1, \ldots, k\}$ a permutation of $\lambda^{-1}(l) = \{v \in V \cup \overline{V} \mid \lambda(v) = l\}$. For every vertex $v \in V \cup \overline{V}$, we define the *left direct sibling* of v to be the vertex preceding v in $\lambda^{-1}(\lambda(v))$, according to the given permutation. If the left direct sibling of v exists, it is denoted by $s_-(v)$, otherwise we set $s_-(v) = \star$. The *right direct sibling* $s_+(v)$ is defined analogously.

In a drawing of the graph, two direct siblings v and w must be separated by a minimal distance $m(v, w) > 0$ (which may be given by the user). The extension of m to arbitrary pairs of vertices on the same level is straightforward: Let $v_1, v_2, \ldots, v_r$ be a consecutive sequence of vertices on a common level, then we define $m(v_1, v_r) = \sum_{i=1}^{r-1} m(v_i, v_{i+1})$.

3 Outline of the Algorithm

For an arbitrary k-level graph with given level embedding, LEVEL_LAYOUT computes a layout having the following properties:

(1) The minimal distance between direct siblings is respected and the given permutations on the levels are not changed.
(2) Vertices belonging to the same level get the same y-coordinate.
(3) The minimal distance between neighboring levels is respected.
(4) All edge segments are drawn straight-line.
(5) Inner segments of long edges are drawn vertically.

For simplicity, we treat the distances mentioned in (1) and (3) as distances between the centers of the vertices. To avoid overlapping, the vertex dimensions have to be included. In the following, we assume that long edges never intersect at inner segments. Otherwise, we cannot satisfy (1), (4), and (5) simultaneously. We propose to rule out this case by a preprocessing step, see [2] for details.

The algorithm LEVEL_LAYOUT performs in three phases. In the first phase, the x-coordinates of virtual vertices are determined. In the second phase, the x-coordinates of original vertices are computed, keeping the virtual vertices in fixed positions. Finally, the y-coordinates of the levels are determined.

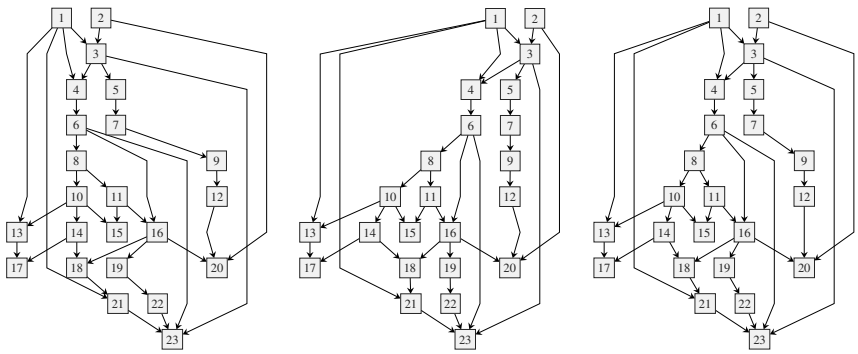
```
LEVEL_LAYOUT

PLACE_VIRTUAL(x);
PLACE_ORIGINAL(x);
PLACE_LEVELS(x,y);
```

In the following sections, we demonstrate the proceeding of LEVEL_LAYOUT by a 10-level graph, see Fig. 3, Fig. 5 and Fig. 6.

4 Placement of the Virtual Vertices

The function PLACE_VIRTUAL places the virtual vertices as close to each other as possible in horizontal direction subject to properties (1) and (5) of Sect. 3.

First, all vertices are placed as far as possible to the left with respect to (1) and (5). This is done as explained by Sander [7], considering the segment ordering graph S of $\overline{G}$. While Sander needed this step only to get a preliminary placement with vertical inner segments, we compute final x-coordinates for virtual vertices. Since the placement is asymmetric, we determine a second placement of the vertices by placing them as far as possible to the right, analogously. Then we take the average positions of the two placements as final x-coordinates for the virtual vertices. See Fig. 3.



(a) Left (b) Right (c) Average

Fig. 3. Placement of the virtual vertices

If the segment ordering graph S is not connected, the placements of vertices belonging to parts of $\overline{G}$ induced by different components of S are not related to each other. This may lead to deformed drawings, since there can be short edges of $\overline{G}$ connecting these parts. To avoid this, we process each connected component of S separately and adjust the placements afterwards by minimizing the total length of these edges. We skip the technical details here, see [2].

5 Placement of the Original Vertices

The function PLACE_ORIGINAL places the original vertices. The positions of the virtual vertices computed by PLACE_VIRTUAL are denoted by $x \in \mathbf{R}^{\overline{V}}$ and regarded as fixed. We have a decomposition of V into maximal consecutive sequences of original vertices belonging to the same level. Let $S = v_1, \ldots, v_r$ be such a sequence. We define $b_- = s_-(v_1)$ and $b_+ = s_+(v_r)$, thus b_- is the virtual vertex bounding S to the left, or $\star$ if S has no siblings to the left, analogously for b_+. Observe that the positions of $v_1, \ldots, v_r$ are already fixed if $x(b_+) - x(b_-) = m(b_-, b_+)$, in this case S is called a *fixed sequence*. Usually,

most sequences are not fixed. Our strategy is to process the sequences successively. When processing a sequence, we compute a placement that minimizes the total length of all edges connecting the vertices of the current sequence with their neighbors in the previously placed sequences, subject to the fixed positions of b_- and b_+. Obviously, the layout depends on the order of processing; the more neighbors have already been fixed, the more edges can be taken into account. We next discuss the order of processing the sequences and how to find the optimal placements.

5.1 The Order of Processing the Sequences

We use an array $D \in \{1, -1, 0\}^{\overline{V}}$ to encode this order and initialize it to zero. The array D is updated dynamically by a function ADJUST_DIRECTIONS discussed below.

The function PLACE_ORIGINAL first traverses the graph level by level downwards, and then, in a second step, it traverses the graph upwards. The direction of traversal is given by $d \in \{1, -1\}$, where a 1 is used to indicate the downward direction and a -1 to indicate the upward direction. For every level, the maximal original sequences are traversed from left to right. The currently examined sequence $S = v_1, \ldots, v_r$, bounded by b_- and b_+, is placed by PLACE_SEQUENCE if and only if $b_- = \star$ or $b_+ = \star$ or $D(b_-) = d$. When placing a sequence, we regard the positions of all neighbors that belong to the preceding level as fixed, i.e., the vertices in $\overline{\delta}(v_i, d) = \{v \in \overline{\delta}(v_i) \mid \lambda(v) = \lambda(v_i) - d\}$ for $i = 1, \ldots, r$ (see below for a justification).

Hence, if $b_- = \star$ or $b_+ = \star$, the positions for $v_1, \ldots, v_r$ are determined twice. The distances between the vertices $b_-, v_1, \ldots, v_r, b_+$ resulting from the first traversal are used as lower bounds for the distances computed in the second traversal. By this strategy, we take both neighboring levels into account for the final placement.

On the other hand, if b_- and b_+ are virtual vertices, the sequence is placed only once. It depends on $D(b_-)$ whether the sequence is placed while traversing upwards or while traversing downwards (for technical reasons, the sequence $v_1, \ldots, v_r$ is represented by its left virtual sibling b_-). It remains to determine D. This is done by ADJUST_DIRECTIONS dynamically, using an array $P \in \{\text{true}, \text{false}\}^{\overline{V}}$. For a virtual vertex v, $P(v)$ is true if and only if the original sequence to the right of v has been placed already. At the beginning, only fixed sequences are regarded as placed.

PLACE_ORIGINAL(x)

for all $b_- \in \overline{V}$
  let b_+ be the next virtual vertex to the right of b_-;
  if $b_+ \neq \star$
    set $D(b_-) = 0$;
    if $x(b_+) - x(b_-) = m(b_-, b_+)$ set $P(b_-) = \text{true}$;
    else set $P(b_-) = \text{false}$;
to be continued...

```
for all d = 1, −1
    for all levels l traversed by direction d
        if level l contains a virtual vertex
            let b_− be the outermost left virtual vertex of level l;
            let v_1, ..., v_r be the vertices to the left of b_−;
        else
            set b_− = ⋆;
            let v_1, ..., v_r be all vertices of level l;
        PLACE_SEQUENCE(x,⋆,b_−,d,v_1,...,v_r);
        for i = 1 to r − 1 set m(v_i, v_{i+1}) = x(v_{i+1}) − x(v_i);
        if b_− ≠ ⋆ set m(v_r, b_−) = x(b_−) − x(v_r);
        while b_− ≠ ⋆
            let b_+ be the next virtual vertex to the right of b_−;
            if b_+ = ⋆
                let v_1, ..., v_r be the vertices to the right of b_−;
                PLACE_SEQUENCE(x,b_−,⋆,d,v_1,...,v_r);
                for i = 1 to r − 1 set m(v_i, v_{i+1}) = x(v_{i+1}) − x(v_i);
                set m(b_−, v_1) = x(v_1) − x(b_−);
            else if D(b_−) = d
                let v_1, ..., v_r be the vertices between b_− and b_+;
                PLACE_SEQUENCE(x,b_−,b_+,d,v_1,...,v_r);
                set P(b_−)=true;
            set b_− = b_+;
        ADJUST_DIRECTIONS(l,d,D,P);
```

After traversing the sequences of level l, the values of D for the next level $l + d$ are computed by ADJUST_DIRECTIONS. We first introduce the notion of a neighboring sequence. Let $S = v_1, \ldots, v_r$ be a maximal original sequence on level $l + d$. Let v_- be the next virtual vertex to the left of S that has a virtual neighbor w_- on level l. If no such vertex v_- exists, set $v_- = w_- = \star$. Analogously, we define v_+ and w_+. The *neighboring* sequences of S on level l are the maximal original sequences on level l between w_- and w_+. Furthermore, if $w_- \neq \star$ and $w_+ \neq \star$, S is said to be an *interior* sequence with respect to l. Otherwise, S is said to be an *exterior* sequence. Fig. 4 illustrates the definition of neighboring sequences.

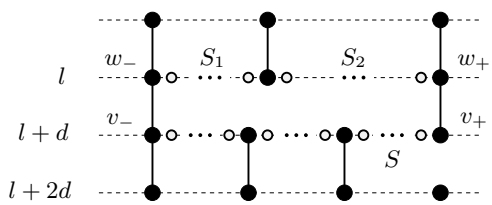


Fig. 4. Neighboring sequences of S. Filled and empty circles represent virtual and original vertices, respectively. Only inner edge segments are displayed. The neighboring sequences of S on level l are S_1 and S_2. The sequence S is interior with respect to l, but exterior with respect to $l + 2d$

We now give an informal description of ADJUST_DIRECTIONS. All interior maximal original sequences $S = v_1, \ldots, v_r$ on level $l + d$ are traversed. If all neighboring sequences of S on level l have already been placed (to check this we use the array P), we set $D(s_-(v_1)) = d$. This allows to place S in the next step of PLACE_ORIGINAL. To explain that this permission is justified, we have to show that every neighbor v on level l of a vertex v_i can be regarded as fixed. We distinguish three cases:

1. The neighbor v is virtual. Then its position is fixed by PLACE_VIRTUAL.
2. The neighbor v belongs to a neighboring sequence of S. Then v has been placed before, as checked by ADJUST_DIRECTIONS explicitly.
3. The neighbor v is original but does not belong to a neighboring sequence of S. In this case, the edge segment (v_i, v) crosses an inner edge segment that is drawn vertically by PLACE_VIRTUAL (like (v_-, w_-) or (v_+, w_+) in Fig. 4), so that the exact position of v does not affect the optimal placement of S.

Theorem 1. *PLACE_ORIGINAL applies PLACE_SEQUENCE to all maximal original sequences.*

Proof. Assume on the contrary that a sequence S on level l is not visited by PLACE_SEQUENCE. By construction of PLACE_ORIGINAL, there either exists a neighboring sequence of S on level $l-1$ that is unvisited as well, or S is exterior with respect to $l-1$. The same holds for level $l+1$. Applying this argument inductively, we get a chain of unvisited neighboring sequences, where the first and the last sequence is exterior. Now by construction of PLACE_VIRTUAL, every such chain must contain a fixed sequence, since otherwise the two parts of the graph divided by the chain could be placed closer to each other. Since fixed sequences are visited by construction, we have a contradiction.

In our example graph (see Fig. 5), the sequences are processed in the following order: The sequences `(1,2)`, `(13)`, `(17)`, `(21,22)`, and `(23)` are not bounded by virtual vertices, they are placed in both traversals. The sequences `(9)`, `(12)`, `(14,15,16)`, and `(20)` are fixed. Traversing downwards, the sequence `(18,19)` is the only bounded sequence placed by PLACE_ORIGINAL, its neighboring sequence `(14,15,16)` is fixed. Traversing upwards, the first bounded sequence is `(10,11)`, its only neighboring sequence `(14,15,16)` is fixed. The next one is `(8)`, since its neighboring sequence `(10,11)` has just been placed. By the same reason, the sequences `(6,7)`, `(4,5)`, and finally `(3)` are placed.

5.2 The Computation of Optimal Placements

For a sequence of original vertices $S = v_1, \ldots, v_r$, PLACE_SEQUENCE finds an optimal placement $x(v_1), \ldots, x(v_r)$ in the following sense:

(*) The placement $x(v_1), \ldots, x(v_r)$ minimizes $\sum_{i=1}^{r} \sum_{v \in \bar{\delta}(v_i, d)} |x(v) - x(v_i)|$ with respect to the minimal distances between $b_-, v_1, \ldots, v_r, b_+$.

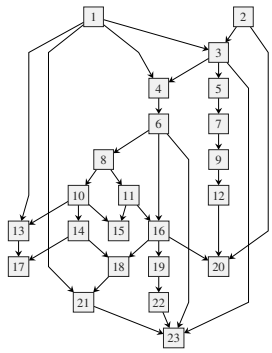


Fig. 5. Placement of the original vertices

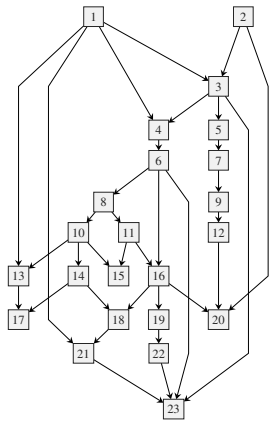


Fig. 6. Final placement

Since PLACE_SEQUENCE uses a divide and conquer strategy, S is not necessarily maximal and b_- now denotes the next virtual vertex to the left of v_1 instead of $s_-(v_1)$, analogously for b_+.

PLACE_SEQUENCE(x,b_-,b_+,d,v_1,...,v_r)

if $r = 1$
    PLACE_SINGLE(x,b_-,b_+,d,v_1);
if $r > 1$
    set $t = \lfloor r/2 \rfloor$;
    PLACE_SEQUENCE(x,b_-,b_+,d,v_1,...,v_t);
    PLACE_SEQUENCE(x,b_-,b_+,d,v_{t+1},...,v_r);
    COMBINE_SEQUENCES(x,b_-,b_+,d,v_1,...,v_r);

Finding a placement satisfying (*) for a single vertex is trivial, therefore we skip the description of PLACE_SINGLE. Next, we explain how to combine two optimal placements for $v_1, \dots, v_t$ and $v_{t+1}, \dots, v_r$ to an optimal placement of the sequence $v_1, \dots, v_r$. Let $m = m(v_t, v_{t+1})$. If $x(v_{t+1}) - x(v_t) \geq m$, then nothing has to be done. Otherwise, we transform the placement step by step, where in each step we increase the distance between v_t and v_{t+1} by either decreasing $x(v_t)$ or increasing $x(v_{t+1})$.

Let $p \in \mathbf{R}$ and $1 \leq i \leq t$. If $x(v_t)$ is decreased to position p, then $x(v_i)$ must be decreased to position $x_p(v_i) = \min\{x(v_i), p - m(v_i, v_t)\}$ in order to keep the two partial placements feasible. Let $j(p) \in \{1, \dots, t\}$ be minimal such that $x_p(v_{j(p)}) < x(v_{j(p)})$; hence decreasing $x(v_t)$ implies decreasing $x(v_{j(p)}), \dots, x(v_t)$. Let

$$r_-(p) = \sum_{i=j(p)}^{t} \left(\#\left\{ v \in \overline{\delta}(v_i, d) \mid x(v) \geq x_p(v_i) \right\} - \#\left\{ v \in \overline{\delta}(v_i, d) \mid x(v) < x_p(v_i) \right\} \right).$$

Thus $r_-(p)$ is the number of edge segments getting longer when decreasing $x(v_t)$ past position p minus the number of edge segments getting shorter. This is called

the *resistance* to decreasing $x(v_t)$ past p. Observe that $r_- : \mathbf{R} \to \mathbf{Z}$ is a piecewise constant monotone function with finitely many steps. Analogously, we define the resistance $r_+(p)$ to increasing $x(v_{t+1})$ past p. Now we proceed as follows. We decrease $x(v_t)$ if $r_-(x(v_t)) < r_+(x(v_{t+1}))$ or increase $x(v_{t+1})$ otherwise. If equality holds, we choose an arbitrary direction. Assume that $x(v_t)$ is decreased. Then we decrease $x(v_t)$ until either $x(v_{t+1}) - x(v_t) = m$ or $r_-(x(v_t))$ reaches a new step. In the latter case, we determine the new resistance and continue decreasing $x(v_t)$ or increasing $x(v_{t+1})$.

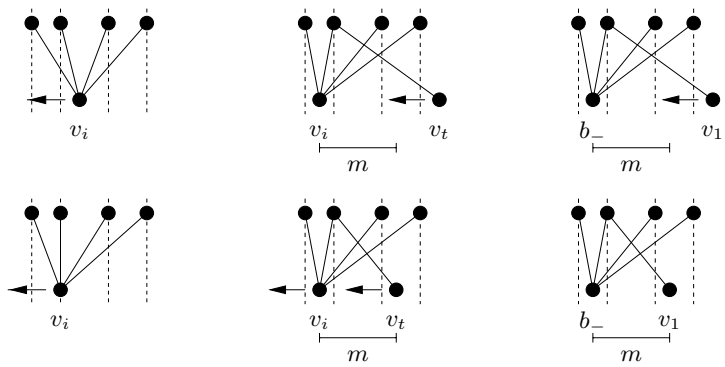
The function COMBINE_SEQUENCES computes the steps of r_- before starting to separate the vertices v_t and v_{t+1}. For every step of length c at position p, a pair (c, p) is stored on a heap R_- by CHANGES_LEFT. The heap R_- is sorted in a decreasing order with respect to the positions p. Analogously CHANGES_RIGHT stores the steps of r_+ on an increasing heap R_+. For runtime reasons, we only move v_t and v_{t+1}, and adjust the positions of $v_1, \ldots, v_{t-1}$ and $v_{t+2}, \ldots, v_r$ later.

COMBINE_SEQUENCES(x,b_-,b_+,d,v_1,...,v_r)

let R_- and R_+ be heaps;
CHANGES_LEFT(R_-);
CHANGES_RIGHT(R_+);
set $r_- = r_+ = 0$;
while $x(v_{t+1}) - x(v_t) < m$
    if $r_- < r_+$
        if $R_- = \emptyset$ set $x(v_t) = x(v_{t+1}) - m$;
        else
            pop $(c_-, x(v_t))$ from R_-;
            set $r_- = r_- + c_-$;
            set $x(v_t) = \max\{x(v_t), x(v_{t+1}) - m\}$;
    else
        if $R_+ = \emptyset$ set $x(v_{t+1}) = x(v_t) + m$;
        else
            pop $(c_+, x(v_{t+1}))$ from R_+;
            set $r_+ = r_+ + c_+$;
            set $x(v_{t+1}) = \min\{x(v_{t+1}), x(v_t) + m\}$;
for $i = t - 1$ down to 1
    set $x(v_i) = \min\{x(v_i), x(v_t) - m(v_i, v_t)\}$;
for $i = t + 2$ to r
    set $x(v_i) = \max\{x(v_i), x(v_{t+1}) + m(v_{t+1}, v_i)\}$;

Assuming that $x(v_t)$ is decreased, we explain the computation of the steps of r_-. Three different situations lead to a step in the resistance function: The resistance changes by 2, if a vertex v_i passes a neighbor v (Fig. 7(a)). This coincides with $x(v_t)$ being decreased to $x(v)+m(v_i, v_t)$. Hence CHANGES_LEFT stores $(2, x(v) + m(v_i, v_t))$ on R_-. When the minimal distance between v_t and a vertex v_i is reached, the position of v_i is decreased as well (Fig. 7(b)). The resistance changes by

$$c_i = \#\{v \in \overline{\delta}(v_i, d) \mid x(v) \geq x(v_i)\} - \#\{v \in \overline{\delta}(v_i, d) \mid x(v) < x(v_i)\},$$



(a) From 0 to 2 (b) From -1 to 1 (c) From -1 to ∞

Fig. 7. Changes of the resistance to moving v_t to the left

and $(c_i, x(v_i) + m(v_i, v_t))$ is stored on R_-. Finally, we enforce the minimal distance between b_- and v_1 by adding $(\infty, x(b_-) + m(b_-, v_t))$ to the heap R_- (Fig. 7(c)).

CHANGES_LEFT(R_-)

```
for i = 1 to t
    set c = 0;
    for all v ∈ δ̄(v_i, d)
        if x(v) ≥ x(v_i) set c = c + 1;
        else
            set c = c − 1;
            push (2, x(v) + m(v_i, v_t)) to R_−;
    push (c, x(v_i) + m(v_i, v_t)) to R_−;
if b_− ≠ ⋆ push (∞, x(b_−) + m(b_−, v_t)) to R_−;
```

From Theorem 1 we know that all maximal original sequences are visited. For the correctness of PLACE_ORIGINAL, it remains to show that placements computed by PLACE_SEQUENCE satisfy the minimality condition (*). Let $v_1, \ldots, v_r$ be the original sequence that has to be placed, and let x be a placement that satisfies (*) both for $v_1, \ldots, v_t$ and for $v_{t+1}, \ldots, v_r$. We show that COMBINE_SEQUENCES merges the two partial placements into a placement satisfying (*) for $v_1, \ldots, v_r$. We first give a lemma that allows us to restrict our attention to placements that are determined by the positions of v_t and v_{t+1}:

Lemma 1. *Let x be a placement satisfying (*) for $v_1, \ldots, v_t$ and for $v_{t+1}, \ldots, v_r$. Then there exists a placement x^* satisfying (*) for $v_1, \ldots, v_r$ such that the following conditions hold.*

(a) $x^*(v_i) = \min\{x(v_i), x^*(v_t) - m(v_i, v_t)\}$ *for* $i \le t$
(b) $x^*(v_i) = \max\{x(v_i), x^*(v_{t+1}) + m(v_{t+1}, v_i)\}$ *for* $i \ge t + 1$.

Proof. Starting with a placement satisfying (*) for $v_1, \ldots, v_r$ but not necessarily (a) and (b), one can transform this placement by successively adjusting the position of v_j to condition (a), for $j = t, \ldots, 1$, such that condition (*) is not violated. For $j = t+1, \ldots, r$, one can proceed analogously to obtain (b). See [2] for a precise proof.

Theorem 2. *The placement $\widetilde{x}$ computed by COMBINE_SEQUENCES satisfies (*) for $v_1, \ldots, v_r$.*

Proof. Assume that $\widetilde{x}(v_{t+1}) - \widetilde{x}(v_t) < m(v_t, v_{t+1})$, otherwise there is nothing to show. For $p \in \mathbf{R}$ let

$$f_-(p) = \sum_{i=1}^{t} \sum_{v \in \overline{\delta}(v_i, d)} |x(v) - \min\{x(v_i), p - m(v_i, v_t)\}| \,,$$

and analogously

$$f_+(p) = \sum_{i=t+1}^{r} \sum_{v \in \overline{\delta}(v_i, d)} |x(v) - \max\{x(v_i), p + m(v_{t+1}, v_i)\}| \,.$$

By Lemma 1, we only need to consider placements satisfying (a) and (b) in order to check the minimality of $\widetilde{x}$. By construction, it is clear that $\widetilde{x}$ satisfies (a) and (b) and is feasible for $v_1, \ldots, v_t$. Hence $\widetilde{x}$ satisfies (*) if $\widetilde{x}(v_t)$ and $\widetilde{x}(v_{t+1})$ minimize $f_-(\widetilde{x}(v_t)) + f_+(\widetilde{x}(v_{t+1}))$ subject to $\widetilde{x}(v_{t+1}) - \widetilde{x}(v_t) \geq m(v_t, v_{t+1})$. However, the function f_+ is convex and piecewise linear, and the gradient to the left of a position p is the resistance to moving v_{t+1} to position p (analogously for f_- and v_t). Thus moving to the direction with lower resistance until $\widetilde{x}(v_{t+1}) - \widetilde{x}(v_t) = m(v_t, v_{t+1})$ yields a minimal placement.

6 Placement of the Levels

In most algorithms, neighboring levels get a fixed distance. Thus the length $|x(w) - x(v)|$ of an edge segment $(v, w) \in \overline{E}$ with $\lambda(v) = l$ and $\lambda(w) = l + 1$ has no influence on the distance between l and $l + 1$. It is however easy to see that long edge segments require a larger level distance than short ones in order to obtain good readability. In this section we propose a method for computing the y-coordinates of the vertices that considers this by adjusting the distance between l and $l + 1$ to the longest edge segment connecting neighboring levels.

Let the *gradient* of (v, w) be defined as $\nabla(v, w) = |x(w) - x(v)|/|y(w) - y(v)|$. We use a fixed maximal gradient GRADIENT. Then we determine the distance between l and $l + 1$ by

$$\max\{\nabla(v, w) \mid (v, w) \in \overline{E} \text{ and } \lambda(v) = l \text{ and } \lambda(w) = l + 1\} = \text{GRADIENT} \,.$$

Explicitly, the distance can be computed as

$$\max\{\text{GRADIENT} \cdot |x(w) - x(v)| \mid (v, w) \in \overline{E} \text{ and } \lambda(v) = l \text{ and } \lambda(w) = l + 1\} \,.$$

Fig. 6 shows the final layout with the new y-coordinates, including some local improvements that have been automated.

7 Runtime

Let $G = (V, E, \lambda)$ be a k-level graph with a given level embedding and let $\overline{G} = (V \cup \overline{V}, \overline{E}, \lambda)$ be the k-level graph resulting from G by introducing virtual vertices as explained in Sect. 2. For $\overline{m} = |\overline{E}|$ and $\overline{n} = |V \cup \overline{V}|$ we have

Theorem 3. *The algorithm LEVEL_LAYOUT computes a layout for the k-level graph G in $O((\overline{m} + \overline{n})(\log(\overline{m} + \overline{n}))^2)$ time.*

Proof. The left and right placement of virtual vertices can be computed in $O(\overline{n})$ time, if both segment ordering graphs are connected. Otherwise, the adjustment of different connected components can be done in $O(\overline{n} + \overline{m} \log \overline{m})$ (see [2]). The first loop of PLACE_ORIGINAL needs $O(\overline{n})$ time in total. The second loop applies PLACE_SEQUENCE to all maximal original sequences at most twice. The function COMBINE_SEQUENCES combines two sequences including r vertices and t incident edge segments in $O((r+t)\log(r+t))$, since at most $r+t+2$ changes of resistance are stored on the heap. By the logarithmic depth of the applied divide and conquer strategy we see that placing a sequence of r vertices with t incident edges can be performed by PLACE_SEQUENCE in $O((r+t)\log(r+t)\log t)$. In total, all calls of PLACE_SEQUENCE take $O((\overline{m}+\overline{n})\log(\overline{m}+\overline{n})\log\overline{m})$ time. Since ADJUST_DIRECTIONS needs $O(\overline{n})$, PLACE_ORIGINAL can be performed in $O((\overline{m}+\overline{n})\log(\overline{m}+\overline{n})\log\overline{m})$ time. The y-coordinates are computed in $O(\overline{m}+\overline{n})$, so we have the desired result.

References

1. F. J. Brandenburg, M. Jünger, and P. Mutzel. Algorithmen zum automatischen Zeichnen von Graphen. *Informatik-Spektrum 20*, pages 199–207, 1997.
2. C. Buchheim, M. Jünger, and S. Leipert. A fast layout algorithm for k-level graphs. Technical report, Institut für Informatik, Universität zu Köln, 1999.
3. E. R. Gansner, E. Koutsofios, S. C. North, and K.-P. Vo. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19(3):214–230, 1993.
4. M. R. Garey and D. S. Johnson. Crossing number is NP-complete. *SIAM Journal on Algebraic and Discrete Methods*, 4(3):312–316, 1983.
5. M. Jünger and P. Mutzel. 2-layer straightline crossing minimization: Performance of exact and heuristic algorithms. *Journal of Graph Algorithms and Applications*, 1:1–25, 1997.
6. Petra Mutzel et al. A library of algorithms for graph drawing. In S. H. Whitesides, editor, *Graph Drawing '98*, volume 1547 of *Lecture Notes in Computer Science*, pages 456–457. Springer Verlag, 1998.
7. G. Sander. A fast heuristic for hierarchical Manhattan layout. In F. J. Brandenburg, editor, *Graph Drawing '95*, volume 1027 of *Lecture Notes in Computer Science*, pages 447–458. Springer Verlag, 1996.
8. K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical systems. *IEEE Transactions on Systems, Man, and Cybernetics*, 11(2):109–125, 1981.