Angewandte Mathematik und Informatik

Universität zu Köln

Report No. 99-368

A Fast Layout Algorithm for k-Level Graphs

by

Christoph Buchheim, Michael Jünger and Sebastian Leipert

1999

Partially supported by DFG-Grant Ju204/7-3, Forschungsschwerpunkt "Effiziente Algorithmen für diskrete Probleme und ihre Anwendungen".

Institut für Informatik
Universität zu Köln
Pohligstraße 1
50969 Köln

1991 Mathematics Subject Classification: 05C85, 68R10, 90C35
Keywords: Sugiyama Algorithm, Hierarchies, Graph Drawing

A Fast Layout Algorithm for k-Level Graphs

Christoph Buchheim * Michael Jünger † Sebastian Leipert ‡

Institut für Informatik, Universität zu Köln

Abstract

In this paper, 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 [STT81]. The Sugiyama algorithm computes a layout for an arbitrary graph by (1) converting it into a k-level graph, (2) reducing the number of edge crossings by permuting the vertices on the levels, and (3) assigning y-coordinates to the levels and x-coordinates to the vertices. In the layouts generated by our algorithm, every edge will have at most two bends, and will be drawn vertically between these bends.

*buchheim@informatik.uni-koeln.de
†mjuenger@informatik.uni-koeln.de
‡leipert@informatik.uni-koeln.de

Contents

Introduction

In various fields of research or business, graph structures arise naturally when dealing with certain objects and their relations. Often, graphs are used to visualize these relations. For example, chemists need to draw large molecules, and biologists need to draw evolutionary trees. Databases are designed using entity-relationship diagrams, and decision support systems for project management need to visualize PERT-networks and activity trees. Software engineers want data flow diagrams, subroutine-call graphs and object-oriented class hierarchies to be visualized.

Usually, the considered graphs are too large to be drawn by hand. For this reason, automatic graph drawing has become an important area of scientific research. The task is to generate a clearly arranged drawing of a given graph (which requires to formalize the term "clearly arranged"). For example, a small number of edge crossings or edge bends is desirable.

Many applications imply a partition of the vertices into k levels such that all edges connect different levels and in the drawing all vertices of a level receive the same y-coordinate. Such graphs are called k-level graphs. Another reason for considering k-level graphs is an idea presented by Sugiyama et. al. in [STT81] that uses k-level graphs in order to draw arbitrary graphs. The Sugiyama algorithm, that serves as a frame for many other graph drawing algorithms, processes a graph in three phases. In a first phase, the vertices are assigned to levels $1, \ldots, k$, 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, the x-coordinates of the vertices are determined in order to produce a nice drawing.

The first two phases of the Sugiyama algorithm have been examined intensively. For phase one we refer to [Sug84], [GKNV93], or [EL91]. Considering phase two, Garey and Johnson [GJ83] showed that the problem of minimizing the number of edge crossings is $\mathcal{NP}$-complete for k-level graphs, even if $k = 2$. According to Eades et al. [EMW86], the problem remains $\mathcal{NP}$-complete even if the order of vertices on one of the two levels is fixed. A lot of effort was spent to design efficient heuristics or exact methods such as branch and cut algorithms for crossing reduction in 2-level graphs (see [JM97], or consult [DETT94] for a list of references). For $k > 2$, the common strategy is to apply a 2-level heuristic consecutively. However, this produces unnecessary crossings in general.

Up to now, only little attention has been paid to the third phase. Two approaches have been presented by Gansner et. al. in [GKNV93]. Assigning nonnegative weights to the segments of edges between levels, both approaches reduce the weighted sum of all edge lengths. Long edges get large weights to avoid the so-called "spaghetti effect", i.e., to avoid long edges with too many bends. The first approach is heuristic, while the second computes an optimal placement using the network simplex method.

In this paper we present a new algorithm for the third phase. We draw every long edge vertically except for its outermost segments. This improves readability and avoids the spaghetti effect. Our algorithm performs in $O(m'(\log m')^2)$, where m' is the number of edge segments in the k-level graph, i.e., the number of edges after adding vertices wherever an edge crosses a level. An implementation is contained in the AGD-Library [AGD].

Finally, we want to demonstrate the importance of the third phase by an example. Figure 1 shows the drawing of a graph that has been taken from [BJM97], p.201. This drawing has been produced using a simple heuristic for the third phase.

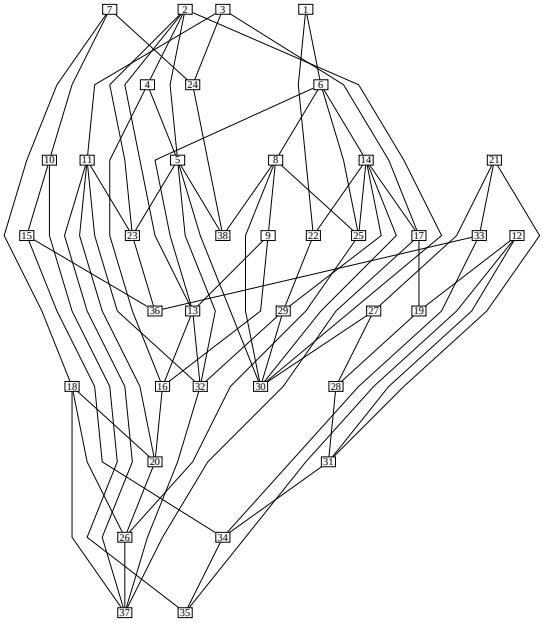

Figure 1: Old drawing.

Figure 2 shows a drawing of the same graph produced by our new algorithm. For generating the second drawing we used the same embedding as in the first drawing. Thus the levels and the permutations of the vertices on the levels are the same in both drawings.

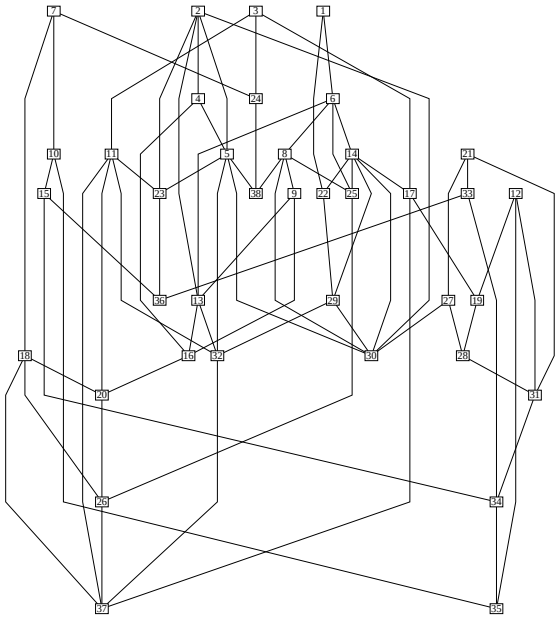

Figure 2: New drawing.

1 Preliminaries

A *graph* G is a pair (V, E) where V is a finite set and $E \subseteq \{\{v, w\} \mid v, w \in V,\ v \neq w\}$. Elements of V and E are called *vertices* and *edges*, 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. (This term is motivated by the idea of using the levels as y-coordinates.)

Let $e = (v, w) \in E$ be a *long edge*, that is, let e satisfy $|\lambda(v) - \lambda(w)| > 1$. Assume that $\lambda(w) > \lambda(v)$. For every level $l \in \{\lambda(v) + 1, \ldots, \lambda(w) - 1\}$ we introduce a *virtual* vertex $\bar{v}_l$ with $\lambda(\bar{v}_l) = l$. We split up e into *edge segments* $(v, \bar{v}_{\lambda(v)+1}), (\bar{v}_{\lambda(v)+1}, \bar{v}_{\lambda(v)+2}), \ldots, (\bar{v}_{\lambda(w)-1}, w)$. Applying this to every long edge, we obtain a set of virtual vertices, disjoint from V, which is denoted by $\bar{V}$. Furthermore, we obtain a set $\bar{E}$ of edge segments. Obviously, this yields a new k-level graph $\bar{G} = (V \cup \bar{V}, \bar{E}, \lambda)$ without long edges. For the following, let $\delta = \delta_G$ and $\bar{\delta} = \delta_{\bar{G}}$. 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 $\lambda^{-1}(l) = \{v \in V \cup \bar{V} \mid \lambda(v) = l\}$. For every vertex $v \in V \cup \bar{V}$, 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 have been given by the user). Usually, a vertex v has a certain diameter d_v in a drawing of G. To avoid vertex overlapping, we assume that $m(v, w) > (d_v + d_w)/2$. 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, then we define $m(v_1, v_r) = \sum_{i=1}^{r-1} m(v_i, v_{i+1})$.

A *layout* or *level drawing* of an embedded k-level graph is a pair (x, y) of functions $V \cup \bar{V} \to \mathbf{R}$, where y satisfies the following conditions: We have $y(v) = y(w)$ if and only if $\lambda(v) = \lambda(w)$, and $y(v) > y(w)$ if and only if $\lambda(v) > \lambda(w)$. Since y-coordinates are given by the levels, we *place* a vertex v by determining $x(v)$. A *placement* of the vertices $v_1, \ldots, v_r$ is thus given by a vector $x \in \mathbf{R}^r$. If $x(w) - x(v) \geq m(v, w)$ whenever $s_+(v) = w$, we call the layout or placement *feasible*.

A layout (x, y) induces a drawing of the graph if x and y are used as x- and y-coordinates and if every edge segment $e \in \bar{E}$ is drawn straightline.

2 The Layout Algorithm

In this section, we present our layout algorithm for embedded k-level graphs. We list the main layout properties in section 2.1. In section 2.2, we give a detailed description of our algorithm. In section 2.3, we apply the algorithm to an example graph. Correctness and runtime are examined in sections 2.4 and 2.5.

2.1 Properties of the layout

We now collect the main properties of all layouts (x, y) computed by our algorithm. We will refer to these properties in the following sections.

(A) The required minimal distances between direct siblings are respected and their order is not changed.

(B) The required minimal distances between neighboring levels are respected.

(C) Inner segments of long edges are drawn vertically.

There is one exception from these rules. Suppose that two long edges cross each other at inner segments $e_1 = (v_1, v_2)$ and $e_2 = (w_1, w_2)$. Then the properties (A) and (C) cannot be satisfied simultaneously, see figure 3. To solve this problem, we have to apply

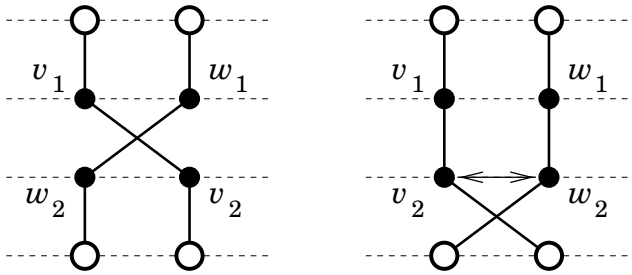

Figure 3: Contradiction between (A) and (C); solving this problem by exchanging v_2 and w_2.

a preprocessing step. We move the intersection downwards by exchanging the order of v_2 and w_2. By repeated application, at least one outer segment will be involved in the crossing of the two long edges. Traversing the graph downwards level by level, all such intersections can be removed using linear time. Observe that this strategy changes the level permutations and may increase the number of edge crossings. To avoid this, the original permutations can be restored after computing the layout.

We assume for the rest of the paper that long edges never intersect at inner segments.

2.2 Description of the algorithm

We now give a description of the layout algorithm LEVEL_LAYOUT. Given an embedded k-level graph $G = (V, E, \lambda)$, we want to compute a layout (x, y) for G satisfying properties (A) to (C) of section 2.1.

LEVEL_LAYOUT consists of three steps. First, PLACE_VIRTUAL computes the x-coordinates of virtual vertices, see section 2.2.1. Next, PLACE_ORIGINAL determines the x-coordinates of original vertices, see section 2.2.2. Finally, all y-coordinates are computed by PLACE_LEVELS explained in section 2.2.3.

> **LEVEL_LAYOUT**
>
> PLACE_VIRTUAL(x);
> PLACE_ORIGINAL(x);
> PLACE_LEVELS(x,y);

2.2.1 Placing the virtual vertices

In this section, we explain how to determine the x-coordinates of virtual vertices $v \in \bar{V}$. During the computation, the original vertices $v \in V$ are assigned to preliminary x-coordinates.

PLACE_VIRTUAL places the virtual vertices as close to each other as possible, respecting properties (A) and (C). Two placements $x_-, x_+ \in \mathbf{R}^{V \cup \bar{V}}$ are computed by functions COMPUTE_POS_LEFT and COMPUTE_POS_RIGHT, respectively. The final placement is $(x_- + x_+)/2$.

> **PLACE_VIRTUAL(x)**
>
> COMPUTE_POS_LEFT(x_-);
> COMPUTE_POS_RIGHT(x_+);
> for all $v \in V \cup \bar{V}$
> set $x(v) = (x_-(v) + x_+(v))/2$;

We only give the description of COMPUTE_POS_LEFT. The function COMPUTE_POS_RIGHT is analogous. Let

$$L(v) = \begin{cases} \{v\} & \text{for } v \in V \\ \{v' \in \bar{V} \mid v \text{ and } v' \text{ belong to the same long edge}\} & \text{for } v \in \bar{V}. \end{cases}$$

The set $V \cup \bar{V}$ is divided into classes, constructed as follows: Traverse the levels downwards. For each level l, consider its outermost left vertex v. If v is not contained in a class yet, we introduce a new class C that is minimal satisfying the following conditions:

(i) $v \in C$.

(ii) If $w \in C$, then $L(w) \subseteq C$.

(iii) If $w \in C$ and $s_+(w) \neq \star$ and $s_+(w)$ is not contained in a class yet, then $s_+(w) \in C$.

The classes are computed by COMPUTE_LEFT_CLASSES and stored in an array $c \in \mathbf{N}^{V \cup \bar{V}}$.

Figure 4 demonstrates the definition of the classes at an example. The original and virtual vertices of the graph are drawn white and black, respectively. Thick lines are inner edge segments. Outer segments do not affect the decomposition into classes, they are drawn as thin lines. The shaded areas comprise the classes, indexed by the numbers on the left.

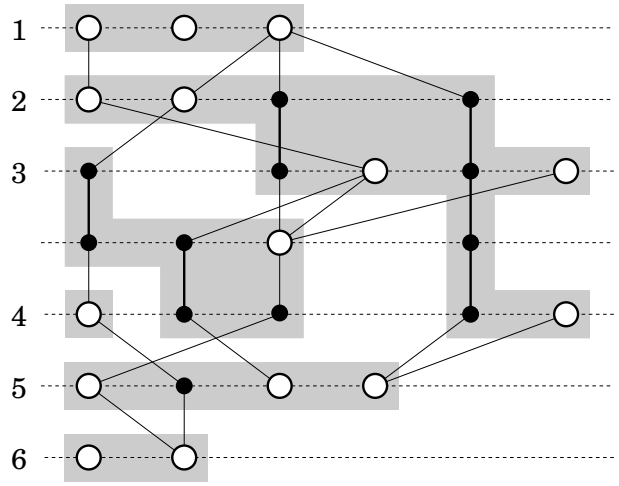

Figure 4: The decomposition into (left) classes.

The reason for decomposing the graph into classes is the idea of placing the virtual vertices as close to each other as possible. This is easy if only vertices of a single class are considered. In this case, all vertices can be placed as far as possible to the left, such that the leftmost vertex gets position zero. Our strategy is to traverse all classes C by the order of construction. First, all vertices of C are placed by PLACE_LEFT without respecting vertices of other classes. Then the computed positions are adjusted to those of previously placed classes by ADJUST_LEFT_CLASS.

COMPUTE_POS_LEFT(x_-)

```
COMPUTE_LEFT_CLASSES(c);
for all i = 1 to the number of classes
    for all vertices v of class i
        if v is not placed yet
            PLACE_LEFT(v,x_-,c);
    ADJUST_LEFT_CLASS(i,x_-,c);
```

COMPUTE_LEFT_CLASSES(c)

for all levels $l = 1$ to k
  set $c' = l$;
  for all vertices v on level l traversed from left to right
    if $c(v)$ is not initialized yet
      for all $v' \in L(v)$ set $c(v') = c'$;
    else set $c' = c(v)$;

For a vertex v of class C, PLACE_LEFT places all vertices in $L(v)$ simultaneously. In order to satisfy property (C), all vertices in $L(v)$ are placed to the same x-coordinate. Let $W = C \cap \{s_-(w) \mid w \in L(v)\}$ be the set of vertices in C that are left direct siblings of vertices in $L(v)$. PLACE_LEFT first places all vertices in W recursively. If $W = \emptyset$, all vertices in $L(v)$ are placed to 0. Otherwise, they are placed to $\max\{x(w) + m(w, s_+(w)) \mid w \in W\}$, i. e., as far as possible to the left respecting properties (A) and (C).

PLACE_LEFT(v,x_-,c)

set $p = -\infty$;
for all $v' \in L(v)$
  if $s_-(v') \neq \star$ and $c(s_-(v')) = c(v')$
    if $s_-(v')$ is not placed yet PLACE_LEFT($s_-(v')$);
    set $p = \max\{p, x_-(s_-(v')) + m(s_-(v'), v')\}$;
if $p = -\infty$ set $p = 0$;
for all $v' \in L(v)$ set $x_-(v') = p$;

As mentioned above, PLACE_LEFT computes relative positions for the vertices of a class C without respecting vertices of other classes. ADJUST_LEFT_CLASS adjusts the positions to previously placed classes. Again, we want to place the classes as close to each other as possible. Let $W' = ((V \cup \bar{V}) \setminus C) \cap \{s_+(w) \mid w \in C\}$ be the set of vertices not contained in C that are right direct siblings of vertices in C. All vertices in W' have been placed before. ADJUST_LEFT_CLASS moves all vertices of C by the same distance d to the right (note that d may be negative):

If $W' \neq \emptyset$, d is set to $\min\{x_-(w) - x_-(s_-(w)) - m(s_-(w), w) \mid w \in W'\}$, i.e., the class is moved as far as possible to the right according to the positions of the vertices in W'. This method is applied to the classes 3 and 4 of figure 4.

If $W' = \emptyset$, the class C is moved to a position that minimizes $\sum |x(v) - x(w)|$, where the sum ranges over edge segments $(v, w) \in \bar{E}$ such that $v \in C$ and w belongs to a class placed in a previous step. To find this position, a heap D collects all these values $x(v) - x(w)$. Then d is chosen as the median in D, or 0, if D is empty. This method is applied to the classes 1, 2, 5, and 6 of figure 4.

ADJUST_LEFT_CLASS(i,x_-,c)

set $d = \infty$;
for all vertices v of class i
  if $s_+(v) \neq \star$ and $c(s_+(v)) \neq i$
    set $d = \min\{d, x_-(s_+(v)) - x_-(v) - m(v, s_+(v))\}$;
if $d = \infty$
  let D be a heap;
  for all vertices v of class i
    for all $w \in \bar{\delta}(v)$
      if $c(w) < i$ push $x(w) - x(v)$ to D;
  let $d_1, \ldots, d_s$ be the values in D;
  if $s = 0$ set $d = 0$;
  else set $d = d_{\lceil s/2 \rceil}$;
for all vertices v of class i
  set $x_-(v) = x_-(v) + d$;

2.2.2 Placing the original vertices

In this section we describe how to place original vertices $v \in V$. We regard the positions of virtual vertices computed by PLACE_VIRTUAL as fixed.

Consider a maximal original sequence $v_1, \ldots, v_r$, that is, a consecutive sequence of original vertices such that both $s_-(v_1)$ and $s_+(v_r)$ are virtual or do not exist. We search for a placement $x(v_1), \ldots, x(v_r)$ that minimizes $\sum_{i=1}^{r} \sum_{v \in \bar{\delta}(v_i)} |x(v) - x(v_i)|$. However, not all neighbors $v \in \bar{\delta}(v_i)$ can be regarded as fixed in their position. Since our strategy is to process all original sequences successively, the layout depends on the order of processing. This order is encoded by an array $D \in \{1, -1, 0\}^{\bar{V}}$, which is initialized to zero and actualized dynamically by the function ADJUST_DIRECTIONS.

PLACE_ORIGINAL traverses the graph level by level, first in a downward direction, then, in a second step, in an upward direction. The direction of traversal is given by $d \in \{1, -1\}$, where 1 is used for downward direction and -1 for upward direction. For every level, the maximal original sequences are traversed from left to right. The currently examined sequence $v_1, \ldots, v_r$, bounded by $b_- = s_-(v_1)$ and $b_+ = s_+(v_r)$, is placed by PLACE_SEQUENCE if and only if $b_- = \star$ or $b_+ = \star$ or $D(b_-) = d$ (see below). If the sequence is placed, the neighbors regarded as fixed are those of the preceding level, i.e., the vertices in $\bar{\delta}(v_i, d) = \{v \in \bar{\delta}(v_i) \mid \lambda(v) = \lambda(v_i) - d\}$ for $i = 1, \ldots, r$.

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

If otherwise $b_-, b_+ \in \bar{V}$, 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 therefore represented by its left virtual sibling b_-). It

remains to determine D. This is done by ADJUST_DIRECTIONS dynamically, using an array $P \in \{\text{true}, \text{false}\}^{\bar{V}}$. 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 original sequences with $x(b_+) - x(b_-) = m(b_-, b_+)$ can be regarded as placed; we will refer to such sequences as fixed sequences.

PLACE_ORIGINAL(x)

```
for all b_- ∈ V̄
    let b_+ be the next virtual vertex to the right of b_-;
    if b_+ ≠ ⋆
        set D(b_-) = 0;
        if x(b_+) − x(b_-) = m(b_-, b_+) set P(b_-) =true;
        else set P(b_-) =false;
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 need to define 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 on level l, and let w_- be this neighbor. 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.

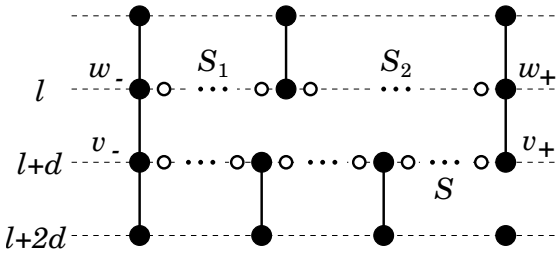

Figure 5: The neighboring sequences of S.

Figure 5 illustrates the definition of neighboring sequences. Again, filled and unfilled circles are virtual and original vertices, respectively. 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 $l + 2d$.

The strategy of ADJUST_DIRECTIONS is to traverse all maximal original sequences $S = v_1, \ldots, v_r$ on level $l + d$ where $b_- = s_-(v_1)$ and $b_+ = s_+(v_r)$ are virtual. If S is interior with respect to l and all neighboring sequences of S on level l have been placed already (to check this we use the array P), we set $D(b_-)$ to d. This forces the function PLACE_ORIGINAL to place the sequence S while processing the next level $l+d$. However, for processing S, all original neighbors w on level l of vertices in S need to be fixed in their positions. If w belongs to a neighboring sequence of S, this is checked explicitly by the function ADJUST_DIRECTIONS. Otherwise, since S is interior, the edge segment (v, w) crosses an inner edge segment $e \in \bar{E}$. (In figure 5, e is either (v_-, w_-) or (v_+, w_+).) Since e is drawn vertically by PLACE_VIRTUAL, the position of w does not affect the optimal placement of S in this case.

ADJUST_DIRECTIONS(l,d,D,P)

```
set v_- = ⋆;
for all virtual vertices v_+ on level l + d traversed left to right;
    if the neighbor w_+ of v_+ on level l is virtual
        if v_- ≠ ⋆
            set p = P(w_-);
            for all virtual vertices w between w_- and w_+
                set p = (p and P(w));
            if p
                set D(v_-) = d;
                for all virtual vertices v between v_- and v_+
                    set D(v) = d;
        set v_- = v_+;
        set w_- = w_+;
```

Lemma 1
PLACE_ORIGINAL applies PLACE_SEQUENCE to all maximal original sequences.

Proof. Let S_0 be such a sequence on level l that is bounded by virtual siblings. Assume that S_0 is not placed in any of the two traversals. If S_0 is interior with respect to $l-1$, it has a neighboring sequence S_{-1} on level $l-1$ that is not placed in either traversal. Indeed, no neighboring sequence of S_0 on level $l-1$ may be placed while traversing upwards, since this requires that S_0 is placed before. Hence if all neighboring sequences of S_0 on level $l-1$ were placed by PLACE_ORIGINAL, they must have been placed while traversing downwards. By construction of the function PLACE_ORIGINAL, S_0 would have been placed afterwards. Thus S_{-1} is not placed either. Iterated application of the same argument yields a chain of sequences $S_0, S_{-1}, \ldots, S_{-p}$ with S_{-p} being exterior with respect to $l-p-1$. Starting at S_0 again and applying the same argument in downward direction, sequences $S_1, \ldots, S_q$ can be found analogously. Thus we have constructed a chain of sequences $S_{-p}, \ldots, S_0, \ldots, S_q$ with the following properties:

(i) For $-p \leq i \leq q$, S_i is not placed by PLACE_ORIGINAL.

(ii) S_{-p} and S_q are exterior sequences.

(iii) For $-p \leq i < q$, S_i and S_{i+1} are neighboring sequences.

See figure 6 for an illustration. The filled circles are virtual vertices and vertical lines are inner edge segments. A dashed line between two virtual vertices indicates that the sequence between these vertices is placed by PLACE_ORIGINAL.

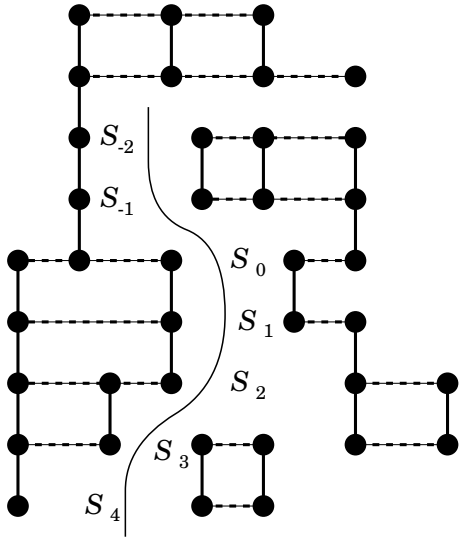

Figure 6: The chain of sequences $S_{-p}, \ldots, S_q$.

Because of (i), none of the sequences $S_{-p}, \ldots, S_q$ is fixed, i.e., for any of these sequences, the bounding virtual vertices have a larger distance than necessary. Because of (ii) and (iii), this is a contradiction, since the virtual vertices are placed as close to each other as possible by the function PLACE_VIRTUAL. □

Next we describe how to place a (not necessarily maximal) consecutive sequence of original vertices $v_1, \ldots, v_r$, such that b_- is the next virtual vertex to the left of v_1 and b_+ is the next virtual vertex to the right of v_r. This is done by PLACE_SEQUENCE. The result is

an optimal placement $x(v_1), \ldots, x(v_r)$ with respect to

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

Our strategy is to apply a divide & conquer algorithm. First, subdivide the sequence $v_1, \ldots, v_r$ at $t = \lfloor r/2 \rfloor$. Then apply PLACE_SEQUENCE recursively to the sequences $v_1, \ldots, v_t$ and $v_{t+1}, \ldots, v_r$. Finally, combine the two optimal placements to an optimal placement for $v_1, \ldots, v_r$.

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);

PLACE_SINGLE finds a placement for a single vertex that satisfies (*).

PLACE_SINGLE(x,b_-,b_+,d,v_1)

let $w_1, \ldots, w_s$ be the vertices in $\bar{\delta}(v_1, d)$ from left to right;
if $s \neq 0$
  set $x(v_1) = x(w_{\lceil s/2 \rceil})$;
  if $b_- \neq \star$ set $x(v_1) = \max\{x(v_1), x(b_-) + m(b_-, v_1)\}$;
  if $b_+ \neq \star$ set $x(v_1) = \min\{x(v_1), x(b_+) - m(v_1, b_+)\}$;

Now let $x(v_1), \ldots, x(v_t)$ and $x(v_{t+1}), \ldots, x(v_r)$ be optimal placements. We have to combine these placements to an optimal placement of the sequence $v_1, \ldots, v_r$. Let $m = m(v_t, v_{t+1})$. If $x(v_{t+1}) - x(v_t) \geq m$, nothing is to do. Otherwise, we transform the placement step by step. 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, $x(v_i)$ must be decreased to position $x_p(v_i) = \min\{x(v_i), p - m(v_i, v_t)\}$ in order to keep the partial placements feasible. Let $j(p) \in \{1, \ldots, t\}$ be minimal with $x_p(v_{j(p)}) < x(v_{j(p)})$. Hence decreasing $x(v_t)$ implies decreasing $x(v_{j(p)}), \ldots, x(v_t)$. Let

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

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

against decreasing $x(v_t)$ past p. Observe that $r_- : \mathbf{R} \to \mathbf{Z}$ is a nonincreasing function with finitely many salti. Analogously, we define the resistance $r_+(p)$ against increasing $x(v_{t+1})$ past p.

Now 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 may chose an arbitrary direction, the computed layout is affected by this decision. We may assume that we decided to decrease $x(v_t)$. Then we do this until we have $x(v_{t+1}) - x(v_t) = m$ or until $x(v_t)$ arrives at a saltus of the resistance function r_-. In the latter case, we determine the new resistance and continue decreasing $x(v_t)$ or increasing $x(v_{t+1})$ as above.

COMBINE_SEQUENCES computes the salti of r_- before starting to separate the vertices v_t and v_{t+1}. For every saltus of length c at position p, COLLECT_LEFT_CHANGES stores a pair (c, p) on a heap R_-. The heap R_- is sorted in a decreasing order with respect to the positions p. Analogously COLLECT_RIGHT_CHANGES stores the salti 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;
COLLECT_LEFT_CHANGES(R_-);
COLLECT_RIGHT_CHANGES(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 $x(v_i) = \max\{x(v_i), x(v_{t+1}) + m(v_{t+1}, v_i)\}$;

Under the assumption of decreasing $x(v_t)$, we need to explain how to compute the salti of r_-. These are saved to the heap R_- by COLLECT_LEFT_CHANGES. For $i = 1, \ldots, t$,

we first compute

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

Starting with $c_i = 0$, we traverse all neighbors $v \in \bar{\delta}(v_i, d)$. If $x(v) \geq x(v_i)$, we set $c_i = c_i + 1$. Otherwise, we set $c_i = c_i - 1$. In the second case, we have to consider the change of resistance induced by v_i passing v (see figure 7). This coincides with $x(v_t)$ being decreased to $x(v) + m(v_i, v_t)$. Resistance changes by 2, since the number of edge segments getting shorter is decreased and the number of edge segments getting longer is increased by 1. Hence, we have to store $(2, x(v) + m(v_i, v_t))$ on R_-. Finally, by definition, c_i is the change of resistance when $x(v_i)$ starts to be decreased, which coincides with $x(v_t)$ being decreased to $x(v_i) + m(v_i, v_t)$ (see figure 8). Hence we store $(c_i, x(v_i) + m(v_i, v_t))$ on R_-. Finally, we enforce the minimal distance $m(b_-, v_1)$ between b_- and v_1 by adding $(\infty, x(b_-) + m(b_-, v_t))$ to the heap R_- (see figure 9).

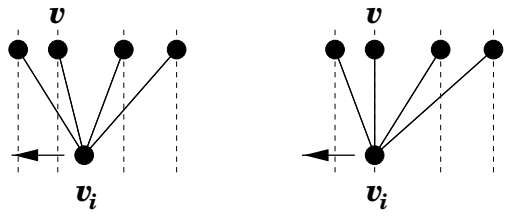

Figure 7: Resistance is 0 (left) and 2 (right).

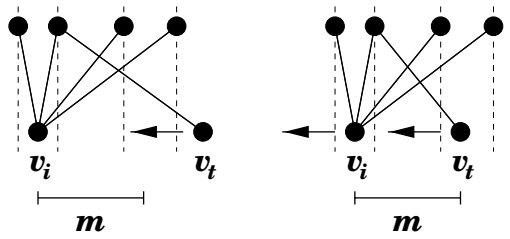

Figure 8: Resistance is -1 (left) and 1 (right).

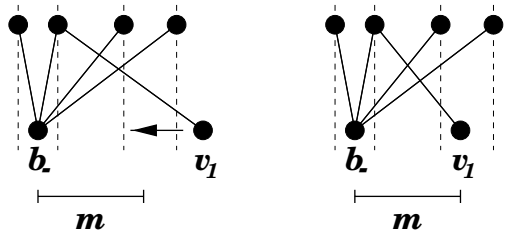

Figure 9: Resistance is -1 (left) and ∞ (right).

COLLECT_LEFT_CHANGES(R_-)

for $i = 1$ to t
  set $c = 0$;
  for all $v \in \bar{\delta}(v_i, d)$
    if $x(v) \geq x(v_i)$ 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 $b_- \neq \star$ push $(\infty, x(b_-) + m(b_-, v_t))$ to R_-;

2.2.3 Placing the levels

In this section we propose two methods of computing the y-coordinates of the levels.

Let MIN_LEVEL_DISTANCE be the minimal level distance given by (B) in section 2.1. In the first method, we just place the levels as close to each other as allowed.

PLACE_LEVELS(x,y) - fixed distance

set $c = 0$;
for all levels $l = 1$ to k
  for all vertices v on level l set $y(v) = c$;
  set $c = c$+MIN_LEVEL_DISTANCE;

Let $l \in \{1, \ldots, k-1\}$ and consider an edge segment $(v, w) \in \bar{E}$ with $\lambda(v) = l$ and $\lambda(w) = l + 1$. In the algorithm above, the length $|x(w) - x(v)|$ of (v, w) has no influence on the distance between l and $l + 1$. However, long edge segments require a larger level distance than short ones for a better readability. We consider this by adjusting the level distance to the longest edge segment connecting the levels.

Define the gradient of (v, w) as $\nabla(v, w) = |x(w) - x(v)|/|y(w) - y(v)|$. Instead of a fixed level distance, we determine a maximal gradient MAX_GRADIENT. The distance between l and $l + 1$ is then determined by

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

Explicitly, it is given by

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

If necessary, the computed level distance is increased to MIN_LEVEL_DISTANCE.

PLACE_LEVELS(x,y) - fixed maximal gradient

set $c = 0$;
for all vertices v on level 1 set $y(v) = c$;
for all levels $l = 2$ to k
  set d=MIN_LEVEL_DISTANCE;
  for all vertices v on level $l - 1$
    for all neighbors w of v on level l
      set $d = \max\{d, \text{MAX\_GRADIENT} * |x(w) - x(v)|\}$;
  set $c = c + d$;
  for all vertices v on level l set $y(v) = c$;

2.3 An example

We will now demonstrate the algorithm by an example. Figure 10 shows an embedded 10-level graph. All vertices are drawn as far as possible to the left, observing minimal vertex distances.

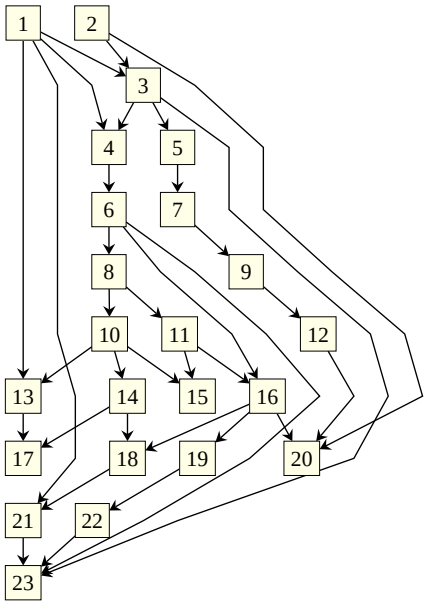

Figure 10: 10-Level graph with a given level embedding.

First we apply PLACE_VIRTUAL to place the virtual vertices. Figure 11 shows the placement induced by x_-, figure 12 shows the placement induced by x_+. In figure 13 we have the final placement of the virtual vertices induced by $(x_- + x_+)/2$.

Figure 14 shows the results of PLACE_ORIGINAL for placing the original vertices. The maximal original sequences are placed as follows:

The sequences `(1,2)`, `(13)`, `(17)`, `(21,22)`, and `(23)` are not bounded by virtual vertices, they are placed in both traversals.

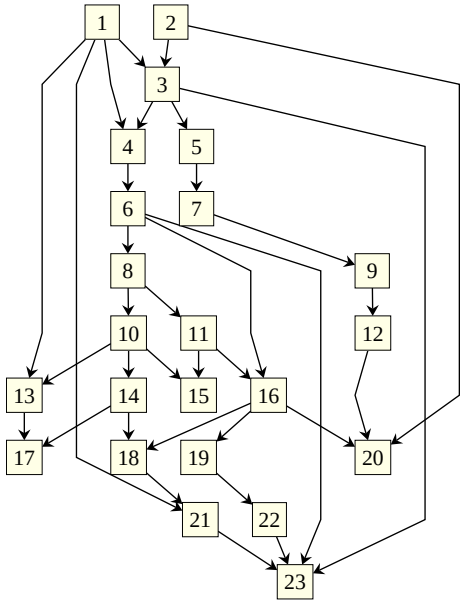

Figure 11: The placement x_-.

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 been placed before. By the same reason, the sequences `(6,7)`, `(4,5)`, and finally `(3)` are placed next.

To compute the y-coordinates we use the second algorithm presented in section 2.2.3. The layout may still have unnecessary bends. Certain local improvements can be performed now. For all layout examples occurring in this paper, we reduced the number of unnecessary bends by moving long edges horizontally (e.g. edge `(1,13)`) and by straightening long edges with only one virtual vertex (e.g. edge `(1,4)`). Figure 15 shows the final layout.

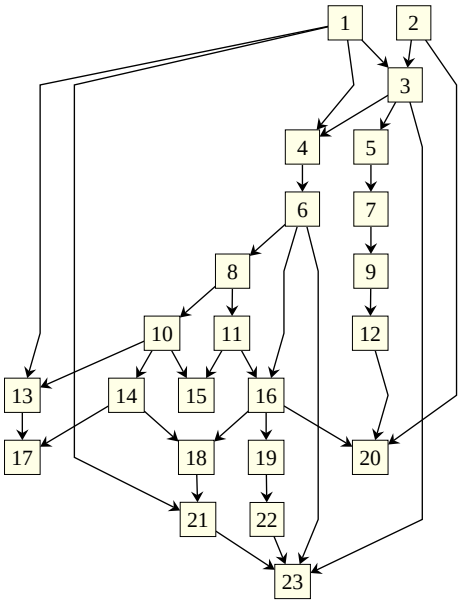

Figure 12: The placement x_+.

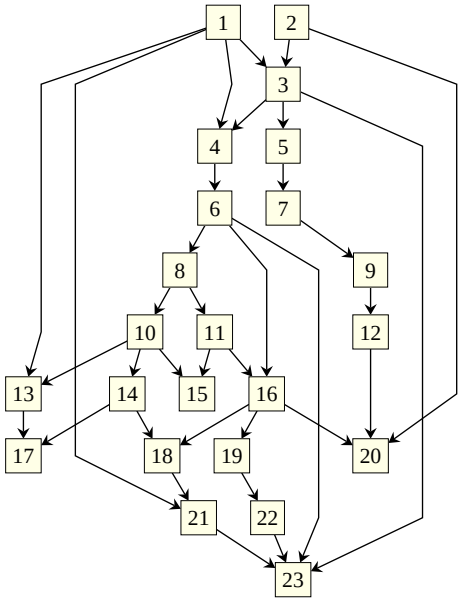

Figure 13: The placement of virtual vertices.

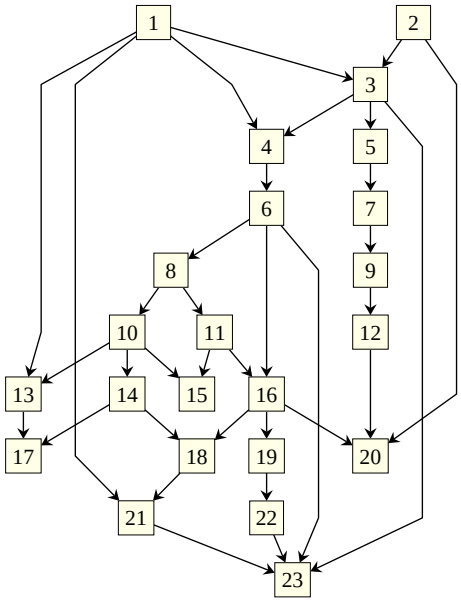

Figure 14: All vertices are placed.

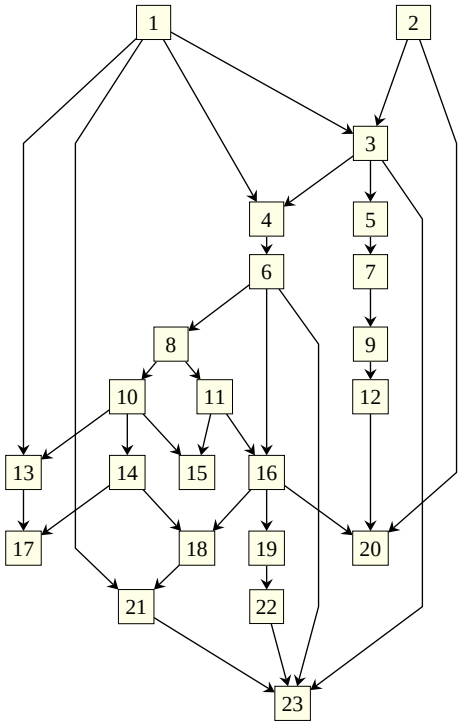

Figure 15: Final Layout.

2.4 Correctness

In this section we will prove that placements of original sequences computed by PLACE_SEQUENCE satisfy the minimality condition (*) of section 2.2.2. The correctness of the other parts of our algorithm is obvious.

We use the notation of section 2.2.2. In particular, $v_1, \ldots, v_r$ is the original sequence to be placed, and x is a placement that satisfies (*) for $v_1, \ldots, v_t$ and for $v_{t+1}, \ldots, v_r$, where $1 \leq t \leq r$. We show that COMBINE_SEQUENCES combines the two partial placements to a placement satisfying (*) for $v_1, \ldots, v_r$.

To simplify the notation, for a placement $\bar{x}$ and integers i_-, i_+ with $1 \leq i_- < i_+ \leq r$ let

$$f(\bar{x}, i_-, i_+) = \sum_{i=i_-}^{i_+} \sum_{v \in \bar{\delta}(v_i, d)} |x(v) - \bar{x}(v_i)|.$$

We say that $\bar{x}$ is feasible for $v_{i_-}, \ldots, v_{i_+}$ if the condition $\bar{x}(v_{i+1}) - \bar{x}(v_i) \geq m(v_i, v_{i+1})$ holds for $i = i_-, \ldots, i_+ - 1$. We have that $\bar{x}$ satisfies (*) for $v_{i_-}, \ldots, v_{i_+}$ if and only if $\bar{x}$ minimizes $f(\bar{x}, i_-, i_+)$ for feasible placements.

We first prove 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 2
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* $i \leq t$

(b) $x^*(v_i) = \max\{x(v_i), x^*(v_{t+1}) - m(v_{t+1}, v_i)\}$ *for* $i \geq t+1$.

Proof. Let x^* be a placement satisfying (*) but not necessarily (a) and (b). Set $x^*_{t+1} = x^*$. We transform x^*_{t+1} into the desired placement by successively adjusting $x^*_{j+1}(v_j)$ to condition (a), for $j = t, \ldots, 1$, obtaining new placements $x^*_t, \ldots, x^*_1$. In every step, the placement remains feasible, and we have $x^*_j(v_i) = x^*_{j+1}(v_i)$ for $i = j+1, \ldots, t$, furthermore $f(x^*_j, 1, t) \leq f(x^*_{j+1}, 1, t)$. Thus, the final placement x^*_1 satisfies (a) and is feasible with $f(x^*_1, 1, t) \leq f(x^*_{t+1}, 1, t) = f(x^*, 1, t)$. For $j = t+1, \ldots, r$, we proceed analogously, obtaining (b).

Let $j \leq t$ and $m = m(v_j, v_{j+1})$. For $j < t$, we may assume by induction that $x^*_{j+1}(v_{j+1}) = \min\{x(v_{j+1}), x^*_{j+1}(v_t) - m(v_{j+1}, v_t)\}$. First, consider the (not necessarily feasible) placements

$$x'(v_i) = \begin{cases} x(v_i) & \text{for } 1 \leq i \leq j \\ x^*_{j+1}(v_i) & \text{for } j < i \leq t \end{cases}$$

and

$$x''(v_i) = \begin{cases} x^*_{j+1}(v_i) & \text{for } 1 \leq i \leq j \\ x(v_i) & \text{for } j < i \leq t. \end{cases}$$

If the placement x'' is feasible for $v_1, \ldots, v_t$, we have $f(x,1,t) \leq f(x'',1,t)$, since x satisfies (*) for $v_1, \ldots, v_t$. Thus

$$\begin{aligned} f(x^*_{j+1},1,t) &\geq f(x^*_{j+1},1,t) + f(x,1,t) - f(x'',1,t) \\ &= f(x^*_{j+1},1,t) + f(x,1,t) - f(x^*_{j+1},1,j) - f(x,j+1,t) \\ &= f(x^*_{j+1},j+1,t) + f(x,1,j) \\ &= f(x',1,t). \end{aligned}$$

For obtaining $x^*_j(v_j) = \min\{x(v_j), x^*_j(v_t) - m(v_j,v_t)\}$, we distinguish two cases:

(1) $x^*_{j+1}(v_j) > \min\{x(v_j), x^*_{j+1}(v_t) - m(v_j,v_t)\}$

(2) $x^*_{j+1}(v_j) < \min\{x(v_j), x^*_{j+1}(v_t) - m(v_j,v_t)\}$.

In case (1), we set $x^*_j = x'$. Since x^*_{j+1} is a feasible placement, the inequality (1) implies $x(v_j) < x^*_{j+1}(v_j) \leq x^*_{j+1}(v_t) - m(v_j,v_t)$. In particular, if $j < t$, we have $x'(v_{j+1}) - x'(v_j) = x^*_{j+1}(v_{j+1}) - x(v_j) \geq x^*_{j+1}(v_j) + m - x(v_j) \geq m$. Hence x' is feasible for $v_1, \ldots, v_t$ and $x'(v_j) = x(v_j) = \min\{x(v_j), x^*_{j+1}(v_t) - m(v_j,v_t)\}$. It remains to show that $f(x',1,t) \leq f(x^*_{j+1},1,t)$. Using the calculation above, we only have to prove the feasibility of x''. This is trivial if $j = t$. If $j < t$, we have $x''(v_{j+1}) - x''(v_j) = x(v_{j+1}) - x^*_{j+1}(v_j) \geq x^*_{j+1}(v_{j+1}) - x^*_{j+1}(v_j) \geq m$ by the induction hypothesis.

In case (2) we consider for $p \in [0,1]$ the placement

$$x^p(v_i) = \begin{cases} (1-p)x^*_{j+1}(v_i) + px(v_i) & \text{for } 1 \leq i \leq j \\ x^*_{j+1}(v_i) & \text{for } j < i \leq t. \end{cases}$$

By (2) we have $x^0(v_j) = x^*_{j+1}(v_j) < \min\{x(v_j), x^*_{j+1}(v_t) - m(v_j,v_t)\} \leq x(v_j) = x^1(v_j)$, so that there exists a $p_0 \in [0,1]$ with $x^{p_0}(v_j) = \min\{x(v_j), x^*_{j+1}(v_t) - m(v_j,v_t)\}$. We set $x^*_j = x^{p_0}$. The placement x^{p_0} is feasible if $j = t$. If $j < t$, the same follows by the induction hypothesis since $x^{p_0}(v_{j+1}) - x^{p_0}(v_j) \geq x^*_{j+1}(v_{j+1}) - x(v_j) \geq x^*_{j+1}(v_{j+1}) - x(v_{j+1}) + m \geq m$. Again, it remains to show $f(x^{p_0},1,t) \leq f(x^*_{j+1},1,t)$. Since $x''(v_{j+1}) - x''(v_j) = x(v_{j+1}) - x^*_{j+1}(v_j) \geq x(v_{j+1}) - x(v_j) \geq m$ by (2), the placement x'' is feasible, so that the above calculation yields $f(x^1,1,t) = f(x',1,t) \leq f(x^*_{j+1},1,t) = f(x^0,1,t)$. Finally, $p \mapsto f(x^p,1,t)$ is a convex function, so that $f(x^{p_0},1,t) \leq f(x^*_{j+1},1,t)$. □

Theorem 3
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

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

and analogously

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

By lemma 2, we only have to consider placements satisfying (a) and (b) in order to check the minimality of $\widetilde{x}$. By the strategy of COMBINE_SEQUENCES, 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 against moving v_{t+1} to position p, as defined in 2.2.2 (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. □

2.5 Runtime

Theorem 4
The algorithm LEVEL_LAYOUT presented in section 2.2 can be implemented to run in $O((m' + n')(\log(m' + n'))^2)$ *time, where* $m' = |\bar{E}|$ *and* $n' = |V \cup \bar{V}|$.

Proof. The function COMPUTE_LEFT_CLASSES needs $O(n')$ time for computing the classes of all vertices. COMPUTE_POS_LEFT applies PLACE_LEFT once to each vertex. This is clear since PLACE_LEFT is only applied if the vertex has not been processed before. Applied to a class c, ADJUST_LEFT_CLASS needs $O(r)$ for the first and third loop, where r is the number of vertices in class c. In the second loop, it needs $O(r + t \log t)$, where t is the number of edge segments incident to a vertex of c, since the heap D can contain at most t items. Hence, ADJUST_LEFT_CLASS needs $O(n' + m' \log m')$ time in total. Altogether, COMPUTE_POS_LEFT needs $O(n' + m' \log m')$. Since COMPUTE_POS_RIGHT performs analogously, we see that PLACE_VIRTUAL can be performed in $O(n' + m' \log m')$ time.

The first loop of PLACE_ORIGINAL needs $O(n')$ time. The second loop applies PLACE_SEQUENCE to all maximal original sequences at most twice. PLACE_SINGLE places a single vertex with s incident edge segments in $O(s)$. COMBINE_SEQUENCES combines two sequences including r vertices and s incident edge segments in $O((r + s)\log(r + s))$, since at most $r + s + 2$ changes of resistance are stored on the heap. By the logarithmic depth of the applied divide & conquer strategy we see that placing a sequence of r vertices with s incident edges can be performed by PLACE_SEQUENCE in $O((r + s)\log(r+s)\log s)$. Hence, all calls of PLACE_SEQUENCE take $O((m'+n')(\log(m' + n'))^2)$ time in total. Since ADJUST_DIRECTIONS needs $O(n')$, PLACE_ORIGINAL can be performed in $O((m' + n')(\log(m' + n'))^2)$ time.

The function PLACE_LEVELS needs $O(m' + n')$, so we have the desired bound for the runtime of LEVEL_LAYOUT. □

References

[AGD] AGD - a library of algorithms for graph drawing. `http://www.mpi-sb.mpg.de/AGD`.

[BJM97] F. J. Brandenburg, M. Jünger, and P. Mutzel. Algorithmen zum automatischen Zeichnen von Graphen. *Informatik-Spektrum 20*, pages 199–207, 1997.

[DETT94] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. Algorithms for drawing graphs: an annotated bibliography, 1994. Via `ftp` from `wilma.cs.brown.edu`, file `/pub/papers/compgeo/gdbiblio.ps.Z`.

[EL91] P. Eades and X. Lin. Notes on the layer assignment problem for drawing directed graphs. In *Proc. 14th Australian Computer Science Conference*, 1991.

[EMW86] P. Eades, B. D. McKay, and N. C. Wormald. On an edge crossing problem. In *Proc. 9th Australian Computer Science Conference*, pages 327–334, 1986.

[GJ83] M. R. Garey and D. S. Johnson. Crossing number is NP-complete. *SIAM Journal on Algebraic and Discrete Methods*, 4(3):312–316, 1983.

[GKNV93] 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.

[JM97] 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.

[STT81] 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.

[Sug84] K. Sugiyama. A readability requirement on drawing digraphs: Level assignment and edge removal for reducing the total length of lines. International Institute for Advanced Study of Social Information Science, 1984.