Optimal Algorithms for Compact Linear Layouts

Wouter Meulemans, Willem Sonke, Bettina Speckmann, Eric Verbeek, and Kevin Verbeek

Dep. of Mathematics and Computer Science, TU Eindhoven, The Netherlands,
[w.meulemans|w.m.sonke|b.speckmann|h.m.w.verbeek|k.a.b.verbeek]@tue.nl

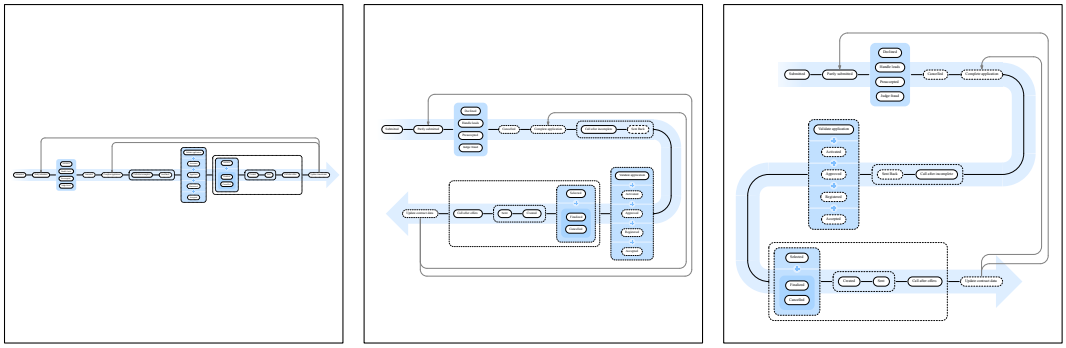

Figure 1 Optimal foldings of a linear layout with three different aspect ratios, as computed by our algorithm: process tree [8] computed from the 2012 Business Process Intelligence Challenge [1].

1 Introduction

Linear layouts are a simple and natural way to draw a graph: all vertices are placed on a single line and edges are drawn as arcs between the vertices. Despite its simplicity, a linear layout can be a very meaningful visualization if there is a particular order defined on the vertices. Common examples of such ordered—and often also directed—graphs are event sequences and processes: public transport systems tracking passenger check-in and check-out, banks checking online transactions (see Fig. 1 for an abstracted view of such a log), or hospitals recording the paths of patients through their system, to name a few. A main drawback of linear layouts are the usually (very) large aspect ratios of the resulting drawings, which prevent users from obtaining a good overview of the whole graph. In this paper we present a novel and versatile algorithm to optimally fold a linear layout of a graph such that it can be drawn effectively in a specified aspect ratio, while still clearly communicating the linearity of the layout (see Fig. 1).

Exact problem statement. We focus on the linear layout of graphs which have an order defined on their vertices. Specifically, our input consists of a graph $G = (V, E)$ with a total order on the vertices V. We are also given the desired aspect ratio ρ, or equivalently the width W_d and height H_d, of the drawing. Our goal is now to draw G as clearly as possible, in a way that communicates the total order of the vertices effectively, while minimizing the unused (empty) space in the drawing. In a classic graph drawing setting vertices are points in the plane and edges are drawn arbitrarily close to each other as (thin) lines. In any practical scenario, however, vertices carry associated data, often visualized as labels, and lines need to be spaced well for readability. We capture both constraints by associating a *block* B_i of a specified width and height with each vertex v_i. This block represents the area needed to draw the vertex v_i, which may represent the size of the corresponding label, or even a (recursive) drawing of a subgraph represented by v_i. B_i also reserves the necessary space to draw the edges surrounding v_i clearly.

34th European Workshop on Computational Geometry, Berlin, Germany, March 21–23, 2018.
This is an extended abstract of a presentation given at EuroCG'18. It has been made public for the benefit of the community and should be considered a preprint rather than a formally reviewed paper. Thus, this work is expected to appear eventually in more final form at a conference with formal proceedings and/or in a journal.

Results. We describe an algorithm which optimally "folds" a given input graph (with a specified order and vertex blocks) for a desired target aspect ratio. The main ingredient of our approach is an algorithm which computes an optimal partition of the input graph and its associated blocks over the various folds, without changing the order. That is, we are solving a packing problem (packing blocks onto rows) while respecting a given order of the blocks. Our algorithm works at interactive speed for reasonably sized layouts.

Related work. A *linear layout* of a graph G is an ordering on its vertices. A linear layout can be visualized by drawing all vertices on a line, in the given order, and drawing the edges as arcs on one side of the line. A *book embedding* is a linear layout of which the edges are partitioned into a number of sets (called *pages*) of non-crossing edges. For any graph the minimum number of pages needed for a book embedding (over all possible linear layouts) is called the *book thickness*. Determining the book thickness of a graph is NP-hard, and the problem stays NP-hard even if we are given a fixed linear layout [4]. For a more complete overview of linear layouts, we refer to the survey by Dujmović and Wood [3].

Packing rectangles has been an active area of research in both algorithms and operations research. For our purposes two types of packing problems are particularly relevant: (two-dimensional) *bin packing* and *strip packing*. Bin packing is already NP-hard in one dimension (see for example [6]), which implies that both two-dimensional bin packing and strip packing are NP-hard as well [7].

Generally packing problems allow reordering of the blocks, while we have to display the blocks in order, which significantly reduces the complexity of the problem. There are some algorithms for on-line strip packing which preserve the order of the blocks. A natural approach here is *next-fit*, which greedily places as many blocks as possible onto a row, before moving to the next row. While there are no bounds on the quality of the solution obtained [2], it performs reasonably well in the average case [5]. To the best of our knowledge the exact variant which we are studying in this paper has not been treated in the literature yet.

2 Folding algorithm

Our strategy is to fold the linear layout into multiple rows, where the vertices are ordered alternatingly from left to right and from right to left. In other words, we assign the vertices to rows, such that all vertices in the same row are consecutive in the given order. We call this assignment a *folding* of G. We can distinguish between two types of edges: *spine edges* (those between two consecutive vertices in the order) and *connectors* (the other edges). Spine edges can be drawn along the folded path (*spine*) itself; connectors must be placed next to it.

We first consider the following problem: given a maximum width W, minimize the height H of the resulting drawing. For each vertex $v_i \in V$ we specify a *block* B_i, that represents the area needed to draw the vertex, including possibly its label. We specify a block B_i by its width w_i, its top-height h_i^T, and its bottom-height h_i^B (see Fig. 2). We separate top-height and bottom-height so that blocks do not need to be centered vertically on the spine. Our goal is now to compute a folding for the blocks B_i such that all blocks are disjoint and the total height is minimized. This packing problem is the core of our algorithm and is described in Section 2.1. Then, in Section 2.2 we show how to draw connectors and how to adapt the block sizes to create space for the connectors.

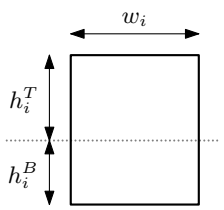

Figure 2 Width (w_i), top-height (h_i^T) and bottom-height (h_i^B) of a block, spine dotted.

2.1 Packing blocks

We first consider the problem in its full generality. That is, we can place the blocks anywhere we want along the spine, as long as the width of the drawing is at most W (see Fig. 3 (left)). In this version of the problem it can be beneficial to leave extra space between two consecutive blocks along the spine to avoid two high blocks sharing the same x-coordinate. Unfortunately we can show that minimizing the height is then NP-hard.

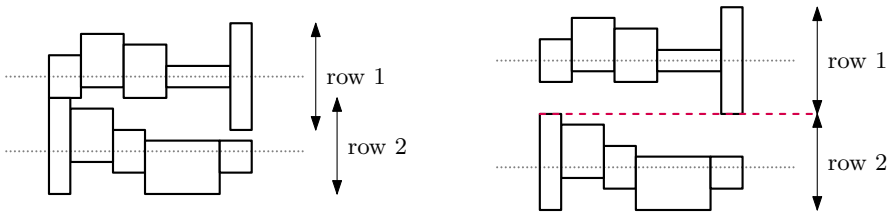

Figure 3 Packing blocks: rows can overlap vertically as long as the blocks do not overlap (left), rows cannot overlap vertically (right).

▶ **Theorem 2.1.** *If rows are allowed to overlap vertically, the problem of minimizing the drawing height is NP-hard, even if we assume that all blocks are vertically centered on the spine (their top and bottom heights are equal) and the assignment of blocks to rows is given.*

Proof sketch. By reduction from 3-SAT (see Fig. 4). We create a grid of blocks in which a column represents a variable and a pair of rows represents a clause. Between the variable columns, we put columns containing "spacer blocks" that are slightly less tall than the blocks in the variable columns. We set H and W such that spacer blocks need to be stacked on top of each other and that variable blocks on consecutive rows need to be next to each other. Necessarily, the variable blocks on even rows are on top of each other and the variable blocks on odd rows as well, forming "zigzag" configurations. A zigzag that begins on the left (on the top row of each clause) is interpreted as *true*, and one that begins on the right is interpreted as *false*. We represent each literal in a clause by a tiny block in the corresponding variable column. We place this tiny block (top row for positive and bottom row for negative literals) such that it requires additional horizontal space on a row if and only if the literal is false. Hence, to ensure that in every clause at least one literal is satisfied, we set the width W such that the rows just fit with two extra tiny blocks, but not with three. ◀

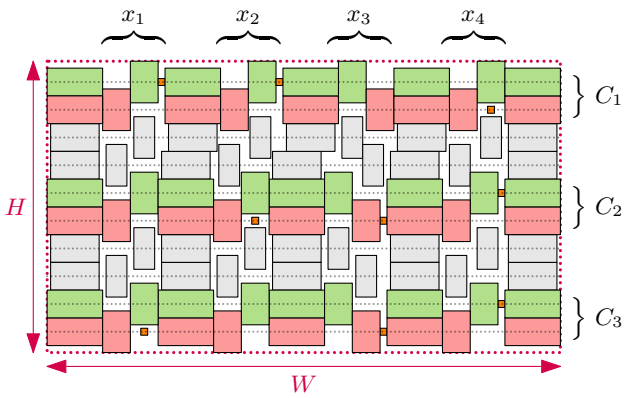

Figure 4 Instance corresponding to $\underbrace{(x_1 \vee x_2 \vee \neg x_4)}_{C_1} \wedge \underbrace{(\neg x_2 \vee \neg x_3 \vee x_4)}_{C_2} \wedge \underbrace{(\neg x_1 \vee \neg x_3 \vee x_4)}_{C_3}$.

We restrict the problem so that different rows cannot overlap vertically (see Fig. 3 (right)). In that setting there is no need to put extra space between two consecutive blocks on the same row, as the height of a row is simply determined by the maximum (top or bottom) height of the blocks in a single row. We use dynamic programming to compute the optimal folding of the blocks. To that end, we first precompute the height $H[i,j]$ $(1 \leq i \leq j \leq n)$ of a row that contains the blocks $B_i, \ldots, B_j$. Since we separate the top-height and the bottom-height of a block, we define $H[i,j] = H^T[i,j] + H^B[i,j]$, where $H^T[i,j]$ and $H^B[i,j]$ are the top-height and bottom-height of a row consisting of blocks $B_i, \ldots, B_j$, respectively. If the total width of the blocks $B_i, \ldots, B_j$ is larger than W, then we set $H^T[i,j]$ and $H^B[i,j]$ to ∞. We thus get the following for $H^T[i,j]$ (and similar for $H^B[i,j]$).

$$H^T[i,j] = \begin{cases} \max_{i \leq k \leq j} h_k^T & \text{if } \sum_{k=i}^{j} w_k \leq W; \\ \infty & \text{otherwise.} \end{cases}$$

All entries of $H[i,j]$ can be computed in $O(n^2)$ time. Next, let $T[i]$ $(0 \leq i \leq n)$ describe the minimum height of a folding involving the blocks $B_1, \ldots, B_i$. We then need to choose how many blocks we will place on the last row. This results in the following recurrence for $T[i]$.

$$T[i] = \begin{cases} 0 & \text{if } i = 0; \\ \min_{0 \leq k < i} \{T[k] + H[k+1, i]\} & \text{otherwise.} \end{cases}$$

The minimum height is then given by $T[n]$. As a result, the minimum height and the corresponding folding can be computed in $O(n^2)$ time.

2.2 Connectors

Spine edges can be drawn by adding a sufficient margin to the width of blocks and using the resulting space between blocks to draw the edges. However, connectors need to be drawn between rows, and we need to ensure that there is enough space to draw them. We can reserve this space by changing the height of the blocks in the dynamic programming formulation, to include the width of adjacent connectors.

We first assume that the connectors are properly nested. That is, if e_{ij} $(i < j)$ is a connector between B_i and B_j, and e_{kl} $(k < l)$ is another connector between B_k and B_l where $i \leq k$, then $j \leq k$ or $l \leq j$. This implies that the connectors can be drawn without crossings on one side of the spine. If the connectors are not properly nested, crossings may be needed; we discuss how to handle such crossings in Section 2.3.

We assume that all connectors are routed along the right side of the drawing. Hence on left-to-right rows, incoming and outgoing connectors go along the top of the row, while on right-to-left rows, they go along the bottom (see Fig. 5). Therefore, the height of a row can differ depending on whether it is drawn left-to-right or right-to-left. To accommodate for this we split $H[i,j]$ into two different tables: $H_{\rightarrow}[i,j]$ and $H_{\leftarrow}[i,j]$.

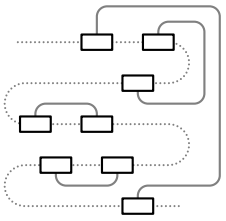

Figure 5 Connectors are routed along the right side of the drawing. (Blocks without incident connectors omitted.)

We show how to compute $H_{\rightarrow}[i,j]$ in the presence of connectors ($H_{\leftarrow}[i,j]$ can be computed similarly). We consider all connectors that start or end at a block B_k with $i \leq k \leq j$. For each such connector e_{kl} we determine the interval of blocks above which e_{kl} must be drawn. Now, for every block B_k, we add $r_k\, w_{\text{conn}}$ to h_k^T to represent the space needed by connectors above B_k, where r_k is the number of connectors that need

to be drawn above B_k and w_{conn} is the space needed per connector. We can then compute $H_{\rightarrow}[i,j]$ by taking the maximum of h_k^T over all $i \leq k \leq j$. To compute r_k efficiently for every block, note that r_k is simply the number of connector intervals that contain k. Since the intervals are nested, we can build a tree (or forest in general) on the intervals where an interval I_1 is a descendant of an interval I_2 if and only if I_1 is contained in I_2. The leaves of this tree are formed by the individual blocks. The value r_k is then simply the depth of B_k in this tree, which can easily be computed for all blocks in $O(m)$ time, where m is the number of connectors. Thus, we can compute a single entry of $H_{\rightarrow}[i,j]$ in $O(m + |j - i + 1|)$ time.

Finally, to draw the connectors that span multiple rows, we need to reserve space on the right side of the drawing. Unfortunately we cannot incorporate this into the dynamic programming algorithm. Instead we compute the nesting depth of the connectors, that is, the size of the largest set of connectors where, for every two connectors, one is always properly contained in the other. This is the largest number of connectors that we may need to draw next to each other on the right side of the drawing in the worst case. The nesting depth is independent from the folding and can hence be precomputed. We then subtract w_{conn} times the nesting depth from W before we compute the optimal folding. Note that, based on the folding, we may not need all of this additional space on the right side of the drawing. In that case we push the connectors as far to the right as possible to create some visual separation.

We note that, due to our versatile setup, we can also show additional information on connectors. In fact, we can add an additional block or even sequences of blocks on a single connector by using our algorithm recursively. We can incorporate blocks on connectors by changing the width w_{conn} of a connector. As a result, connectors can have different widths; our algorithm can easily be adapted to this scenario.

2.3 Crossing connectors

We now consider the case where the connectors are not properly nested. Here we may have connectors that cross each other, which we want to avoid as much as possible. Even if we already know the order of the vertices along the spine, minimizing the number of crossings in this situation is still known to be NP-hard (see Section 1). We therefore use a heuristic to obtain a low number of crossings: we compute a maximum set of properly-nested connectors, remove them and iterate until no connectors are left. This results in a collection of sets $E_1, \dots E_k$. We then draw each set of connectors separately as described in Section 2.2, ignoring any crossings among the different sets.

To find the largest subset of properly-nested connectors, we use the following dynamic programming formulation. We first order the connectors such that $e_{ij} < e_{kl}$ if $i < k$, or if $i = k$ and $j > l$. Let $c_1, \dots, c_m$ be the resulting ordered set of connectors, and let $f(i)$ be the index of the first connector in the order that has B_i as a starting block. Now we define $T[i,j]$ $(1 \leq i \leq m+1,\ 0 \leq j \leq n)$ as the size of the largest subset of connectors among $c_i, \dots, c_m$ that are properly-nested and all end at a block before or at B_j. Now, for every connector in order, we simply need to choose whether we want to include the connector in our set or not. We obtain the following recurrence (here we assume that $c_i = e_{kl}$).

$$T[i,j] = \begin{cases} 0 & \text{if } i = m+1; \\ T[i+1,j] & \text{if } l > j; \\ \max\{T[i+1,l] + T[f(l),j] + 1, T[i+1,j]\} & \text{otherwise.} \end{cases}$$

The size of the largest subset of properly-nested connectors is given by $T[1,n]$. It can be computed in $O(mn)$ time, where n is the number of blocks and m is the number of connectors.

2.4 Aspect ratio

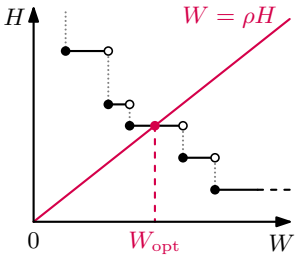

Figure 6 The height of a drawing is a descending function of its width.

So far we have presented an algorithm that, given a maximum width W, computes the minimum height $H(W)$ of a folding of the graph. Our goal is to find a folding that has a particular aspect ratio ρ: we need to find a width W such that $W/H(W) = \rho$. As $H(W)$ is non-increasing as W increases (see Fig. 6), we can use a binary search to find the width W for which $W/H(W) = \rho$. As the initial lower bound for W we take the maximum width of all blocks, because the drawing can never be narrower than that; as the upper bound we use the sum of the widths of all blocks. Since $H(W)$ is not continuous, we may not be able to obtain the exact correct aspect ratio, but the binary search will at least find the width W at which our folding algorithm jumps over the aspect ratio ρ. The resulting drawing then may have some unused height, but the drawing is as close to the correct aspect ratio as possible. More precisely, the binary search maximizes the size of the vertices (labels) in the resulting drawing. That is, if we are given a drawing area of size $W_d \times H_d$ (with aspect ratio ρ, so $W_d/H_d = \rho$), and we scale our drawing by a factor α to fit the drawing area (that is, $\alpha \cdot W \leq W_d$ and $\alpha \cdot H \leq H_d$), the binary search results in a drawing that maximizes α.

Acknowledgments. The authors wish to thank Tim Ophelders for the helpful discussions. Willem Sonke, Bettina Speckmann and Kevin Verbeek are supported by the Netherlands Organisation for Scientific Research (NWO) under project no. 639.023.208 (W.S. and B.S.) and no. 639.021.541 (K.V.). Wouter Meulemans is (partially) supported by the Netherlands eScience Center (NLeSC) under grant number 027.015.G02.

References

1 BPI Challenge 2012. `http://www.win.tue.nl/bpi/doku.php?id=2012:challenge`.
2 János Csirik and Gerhard J. Woeginger. On-line packing and covering problems. In Amos Fiat and Gerhard J. Woeginger, editors, *Online Algorithms*, pages 147–177. Springer, 1998.
3 Vida Dujmović and David R. Wood. On linear layouts of graphs. *Discrete Mathematics and Theoretical Computer Science*, 6:339–358, 2004.
4 M. R. Garey, D. S. Johnson, G. L. Miller, and C. H. Papadimitriou. The complexity of coloring circular arcs and chords. *SIAM Journal on Matrix Analysis and Applications*, 1(2), 1980.
5 Micha Hofri. Two-dimensional packing: Expected performance of simple level algorithms. *Information and Control*, 45:1–17, 1980.
6 Bernhard Korte and Jens Vygen. *Combinatorial Optimization*, chapter Bin-Packing, pages 426–441. Springer, 2005.
7 Silvano Martello, Michele Monaci, and Daniele Vigo. An exact approach to the Strip-Packing problem. *INFORMS Journal on Computing*, 15(3):310–319, 2003.
8 Wil M. P. van der Aalst. *Process Mining: Data Science in Action*. Springer, 2012.