Optimal Algorithms for Compact Linear Layouts

Willem Sonke* Kevin Verbeek* Wouter Meulemans* Eric Verbeek* Bettina Speckmann*

Department of Mathematics and Computer Science, TU Eindhoven, The Netherlands

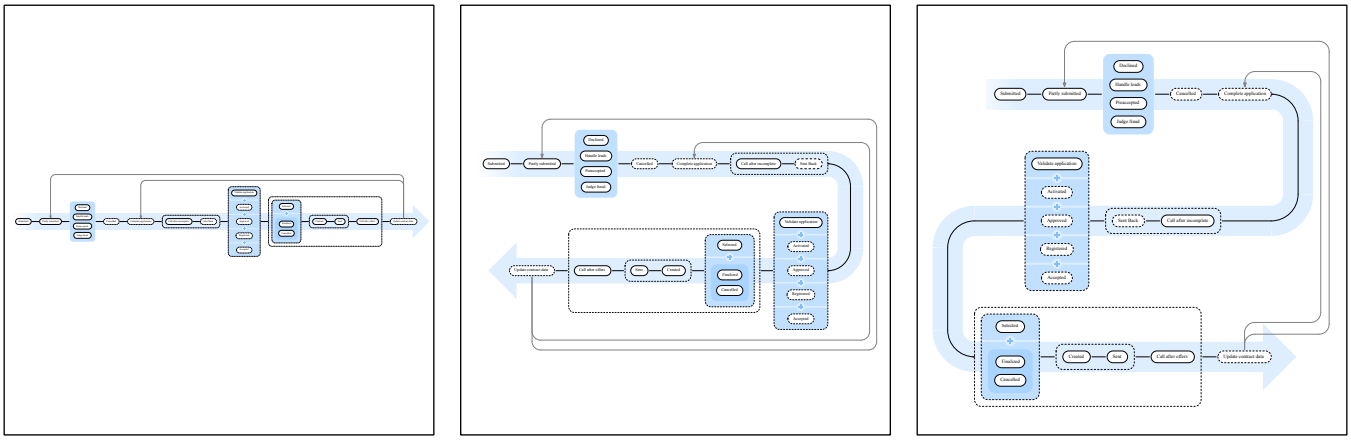


Figure 1: Optimally folding a linear layout: process tree [15] computed from the 2012 Business Process Intelligence Challenge [3].

ABSTRACT

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. 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. Our algorithm allows vertices to be drawn as blocks or rectangles of specified sizes to incorporate different drawing styles, label sizes, and even recursive structures. For reasonably-sized drawings the folded layout can be computed interactively. We demonstrate the applicability of our algorithm on graphs that represent process trees, a particular type of process model. Our algorithm arguably produces much more readable layouts than existing methods.

Index Terms: F.2.2 [Analysis of Algorithms and Problem Complexity]: Nonnumerical Algorithms and Problems—Routing and layout

1 INTRODUCTION

With the increase in storage capacity over the last decades, the logging of data in various forms has grown tremendously. A multitude of systems and devices produce time-stamped data while monitoring events 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. These enormous data logs have little value, unless they are analyzed for patterns, outliers and peculiarities. Here visualization can be instrumental to generate hypotheses, gain insight, and to communicate results. The time aspect of data logs imposes a natural order on the logged data. Hence an easily accessible way of visualizing data logs is using a *linear layout*: data ordered according to time in typical left-to-right fashion, matching the intuitive notion of the progress of time. In fact, such linear layouts work well as a visual paradigm whenever there is a concept of a (partial or total) order on the input.

Though linear layouts are a natural way of visualizing time-stamped data, restricting the layout to be left-to-right (or top-to-bottom) to communicate the order comes with some immediate drawbacks. Most significantly, the aspect ratio of linear layouts deteriorates quickly as the input grows. This makes linear layouts practically unusable for larger data sets, as visualization must often be constrained to some graphical display of bounded aspect ratio. That is, either the linear layout must be made small enough to fit the display—making the visualization unreadable—or shown only partially—therefore failing to provide an overview of the overall data. Though many visualization techniques are easily adapted to be compact and to match a target aspect ratio, the resulting visualizations typically do not show the order within the data anymore.

In this paper we attempt to address this issue: we present a novel and versatile algorithm to optimally fold a linear layout 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. Time-stamped data and other ordered structures are readily captured by graphs: each data element corresponds to a vertex and (sequential) interactions between elements correspond to edges. In the following we will hence abstract away from any particular type of data and 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

*e-mail: [w.m.sonke|k.a.b.verbeek|w.meulemans|h.m.w.verbeek|b.speckmann]@tue.nl

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.

Results and organization. We first review related work in Section 2. Section 3 then presents the main technical contribution of our paper. Specifically, we describe an algorithm which optimally "folds" a given input graph (with a specified order and vertex blocks) for a desired target aspect ratio. That is, for a given width we show how to minimize the necessary height of the drawing. Binary search then leads us to the optimal drawing for a given 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. By using blocks to represent the space needed to draw a vertex (and nearby edges), we can handle a wide variety of drawing styles, including labels or other information on vertices, recursive drawings inside vertices, and even extra information on edges in a clear and organized manner. Last but not least, our algorithm works at interactive speed for reasonably sized layouts, see the accompanying video for a demonstration.

In Section 4 we illustrate the use of our method with two examples. Our first example comes from the area of business process analysis, and more specifically, from process mining. Process mining analyzes event logs, trying to reconstruct the underlying (business) process that created the events. There are various representations for the resulting process models, which all have a comparatively clear notion of order, up to events that are performed in parallel. Such parallel events can readily be captured by our block structure, allowing us to visualize the linear structure of complex event logs in an effective way. Our second example is a timeline of historical events. This example demonstrates the versatility of our approach.

2 Related work

Timelines and other linear layouts which order data according to time are a frequent component of visualizations and visual analytics systems. However, to the best of our knowledge, no attempt has been made so far to optimally compact such visualization for a desired target aspect ratio, while keeping the order as the focus of the visualization. We are hence restricting ourselves to more closely related work which (1) explores linear layouts from a graph drawing perspective, and (2) discusses packing blocks (rectangles) optimally from an algorithmic point of view.

Linear layouts. 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 pair of edges in such a drawing can be either crossing or non-crossing. A *book embedding* is a linear layout of which the edges are partitioned into a number of sets (called *pages*) of non-crossing edges. The intuition is that we can draw the vertices on the spine of a book, in the given order, and draw every subset of edges on a separate page. 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 [9]. For a more complete overview of linear layouts, we refer to the survey by Dujmović and Wood [7]. A related problem is that of minimizing the number of crossings of a drawing of a graph G. It has been known for a long time that this problem is NP-hard [8]. Just like book thickness, this problem remains NP-hard if a fixed linear layout is given [14].

Packing rectangles. Packing rectangles has been an active area of research in both algorithms and operations research. In general, given some bounding shape and a list of objects, the objective of a packing problem is to fit the objects into the bounding shape while minimizing some objective function. For our purposes two types of packing problems are particularly relevant: (two-dimensional) *bin packing* and *strip packing*. In both problems, the objects to be packed are rectangles with a given width and height. In bin packing, the bounding shape consists of an unlimited number of rectangles of a fixed size called *bins*, and the objective is to minimize the number of bins used. In strip packing, the bounding shape is an infinitely-tall strip of a given width (a *strip*), and the objective is to minimize the height of the strip used.

For both these problems, many NP-hardness results are known, making finding the optimal solution infeasible in practice for large inputs. In particular, one-dimensional bin packing is already NP-hard (see for example [11]), which implies that both two-dimensional bin packing and strip packing are NP-hard as well [13].

In order to reduce the complexity, approximation algorithms have been studied. One approach is to limit the allowable packings to those that consist of layers of blocks called *levels*. This corresponds to our setting, as we aim to divide the blocks into rows. Several *level-based algorithms* for strip packing have been proposed [1, 6] as approximation algorithms for the general strip packing problem. More recently, the level strip packing problem has been studied in its own right [2]. However, even if we allow level-based layouts only, it is still NP-hard to minimize the height used, again by reduction from one-dimensional bin packing [2].

A major difference between the packing problems studied in literature and our setting is that packing problems allow reordering the blocks, while we want to display the blocks in order. This significantly reduces the complexity of the problem. Some algorithms for strip packing have been studied that do preserve the order of blocks, in the context of on-line algorithms. A natural algorithm is *next-fit*, that greedily places as many blocks as possible onto a row, before moving to a next row. While the approximation factor of this algorithm is unbounded in the worst case [6], the algorithm was found to perform reasonably well in the average case by Hofri [10]. However, to the best of our knowledge the order-preserving level strip packing problem has not been studied in the off-line setting. In Section 3.1, we propose an off-line exact algorithm for this problem.

3 Folding algorithm

Recall that our input consists of a graph $G = (V, E)$ with a total order on the vertices V. Furthermore, we are also given the desired aspect ratio ρ, or equivalently the width W and height H of the final drawing. Naturally, the goal is to draw G as large as possible, minimizing the amount of empty space in the drawing.

Our strategy to draw G with some specified aspect ratio is to fold a 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 different rows, such that all vertices in the same row are consecutive in the given order. We call this assignment a *folding* of G. Since the vertices of G are ordered, we can distinguish between two types of edges. *Spine edges* are the edges between two consecutive vertices in the order, and can hence be drawn along the folded path, or *spine*, itself. *Connectors* are the edges that are not spine edges, and thus these edges must be drawn next to the spine. We typically assume that the input graph G is directed and that all spine edges are directed in the same way along the spine, but our method does not rely on this restriction.

As mentioned above, we are actually solving the following problem: given a maximum width W, minimize the height H of the

resulting drawing. In Section 3.4 we explain how we can use a solution to this slightly different problem to compute a drawing with a desired aspect ratio.

In a classic graph drawing setting, where vertices are drawn as points in the plane, and edges as curves, minimizing the height for a given width is not very difficult. However, in practice, vertices are typically not drawn as points (they need some area, especially with labels), and edges should not be drawn too close to each other for the sake of readability. To incorporate these restrictions, we specify a *block* B_i for every vertex $v_i \in V$. The block B_i represents the area needed to draw the vertex v_i, which may represent the size of the corresponding label, or perhaps even a (recursive) drawing of a subgraph represented by v_i. More importantly though, it can also incorporate the space needed to draw the surrounding edges. If there are many connectors that need to be drawn between two rows of the folding, then our drawing needs to create enough room between the rows to draw the connectors nicely. We ensure that there is sufficient room by making the blocks of the corresponding vertices high enough. Note that in this case the height of a block depends on the chosen folding, but our algorithm can handle this case correctly.

Our input hence also includes an ordered set of blocks $B = \{B_1, \ldots, B_n\}$ corresponding to the vertices of G. To enable the full versatility of our drawing approach, 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. This allows us to reserve different amounts of space above and below a vertex to draw connectors. 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. In other words, we want to pack rectangles into different rows obeying a given order. This packing problem is the core of our algorithm and is described in Section 3.1. Note that packing the blocks is completely independent from the connectors. In Section 3.2 we show how to draw connectors and how to adapt the block sizes to create space for the connectors.

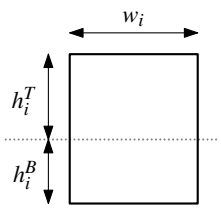


Figure 2: The width (w_i), top-height (h_i^T) and bottom-height (h_i^B) of a block, spine dotted.

3.1 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 order is correct, and the width of the drawing is at most W (see the top part of Fig. 3). 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.

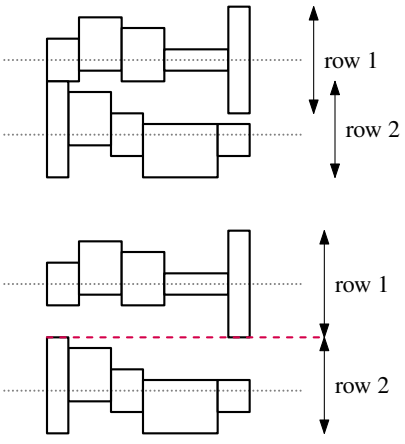


Figure 3: Packing blocks: *Top:* rows can overlap vertically as long as the blocks do not overlap. *Bottom:* rows cannot overlap vertically.

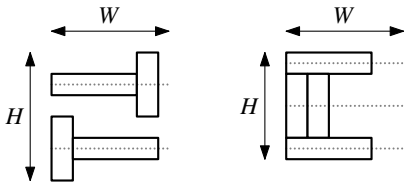


Figure 4: Two foldings of four blocks with maximum width W. *Left:* A greedy approach. *Right:* The optimal folding.

In this scenario there are two main problems. First, since blocks of consecutive rows can "interlock", any connectors between these two rows may need to zigzag to avoid crossings, leading to very complex drawings. Second, we can show that minimizing the height in this version of the problem is actually NP-hard by a reduction from 3-SAT, even if we assume that all blocks are vertically centered (that is, their top- and bottom-heights are the same) and the assignment of blocks to rows is given. The reduction is sketched in Appendix A. Therefore, we cannot expect to find efficient optimal algorithms for the most general version of the problem.

Instead we restrict the problem as follows: different rows cannot overlap in their y-coordinates (see the bottom part of Fig. 3). This restriction directly ensures that connectors can be drawn straight between two rows. Additionally, there is no need to reserve 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. In this setting, we can now try to minimize the height of the folding.

The most basic approach places as many blocks as possible on a row until we are forced to move to the next; this is the *next-fit* algorithm (see Section 2). However, such a greedy approach does not minimize the height: it can be helpful to leave a row half-empty to get two tall blocks onto the same line (see Fig. 4).

Instead we use dynamic programming to compute the optimal folding of the blocks. To that end, we first precompute the height $H[i,j]$ $(1 \le i \le j \le n)$ 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 something similar for $H^B[i,j]$).

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

It is easy to see that all entries of $H[i,j]$ can be computed in $O(n^2)$ time. Next, let $T[i]$ $(0 \le i \le n)$ 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 $T[i]$.

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

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.

3.2 Connectors

Spine edges can easily be drawn by adding a sufficient margin to the width of blocks and using the resulting space between blocks to draw

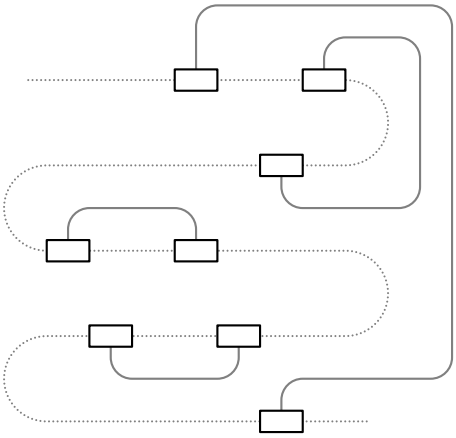


Figure 5: Connectors are not folded along with the blocks; instead they are routed along the right side of the drawing. (Blocks without incident connectors are omitted.)

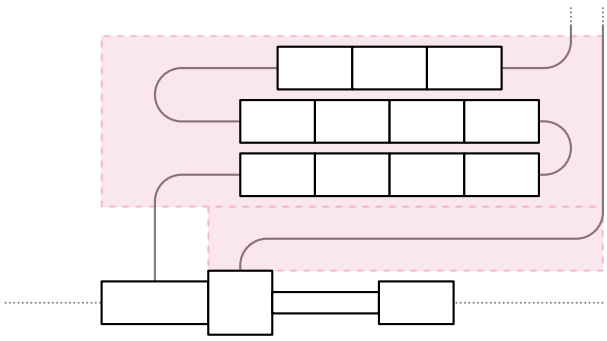


Figure 6: Drawing blocks on connectors.

the edges. However, connectors need to be drawn between rows, and thus we need to ensure that there is enough space to draw every connector. We assume that all connectors have a width w_{conn} that determines the empty space that needs to be around it. As already mentioned earlier, we can reserve this space by changing the height of the blocks in the dynamic programming formulation. In particular, we need to change the values of $H^T[i,j]$ and $H^B[i,j]$, and we need to reserve enough space on the side of the drawing to route connectors that span multiple rows.

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 directly 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 will discuss how to handle such crossings in Section 3.3.

Without loss of generality we assume that all connectors are routed along the right side of the drawing. This means that 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 $H_{\leftarrow}[i,j]$. We now show how to compute $H_{\rightarrow}[i,j]$ in the presence of connectors. The table $H_{\leftarrow}[i,j]$ can be computed in a similar way.

To compute the value of $H_{\rightarrow}[i,j]$ for some $i \leq j$, 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} between B_k and B_l we determine the set of blocks above which e_{kl} must be drawn. There are two cases. If e_{kl} starts and ends in the same row ($i \leq l \leq j$), e_{kl} is drawn above the blocks $B_k, B_{k+1}, \ldots, B_l$. If e_{kl} spans more than one row ($l > j$ or $l < i$), it is drawn above $B_k, B_{k+1}, \ldots, B_j$. 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. We can then easily 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. Therefore, 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. Note that 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 (see Fig. 6). 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.

3.3 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 clearly want to avoid as much as possible. Unfortunately, even though 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 2). We therefore employ the following heuristic to minimize the number of crossings. First, we compute the largest subset of connectors that is properly nested and extract them. We then repeat this process on the remaining set of connectors until no connectors are left. This results in a collection of sets $E_1, \ldots E_k$, where each E_i is a set of properly nested connectors. We then reserve space for each set of connectors separately as is described in Section 3.2. Finally, we also draw the sets of connectors separately, 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 $c_1, \ldots, c_m$ be the resulting ordered set of connectors. Furthermore, 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 $c_i, \ldots, c_m$ 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 then given by $T[1,n]$. This set can be computed in $O(mn)$ time, where n is the number of blocks and m is the number of connectors.

Finally note that we can draw a set of properly-nested connectors either on the right side or on the left side of the spine, where two sets of connectors cannot cross if they are drawn on different sides of the spine. On the other hand, if two sets of connectors are drawn on the same side of the spine, then we know exactly how many crossings this will create. We therefore may want to minimize the number of crossings by optimally choosing the sides on which to draw every set of connectors. Unfortunately, this problem then reduces to MAX-CUT, which is NP-hard. Instead, we can simply choose a side randomly for each set of connectors, which is known to be a good approximation of the optimal solution. To further improve upon the solution, we can iteratively move a set of connectors from one side to another if that reduces the number of crossings, until no such moves are possible anymore.

3.4 Aspect ratio

So far we have presented an algorithm that, given a maximum width W, can compute the minimum height $H(W)$ of a folding of the graph. Our ultimate goal is to find a folding that has a particular aspect ratio ρ. Therefore, we need to find a width W such that $W/H(W) = \rho$. Since $H(W)$ is the minimal height given a maximum width W, $H(W)$ is non-increasing as we increase W (see Fig. 7). Therefore 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.

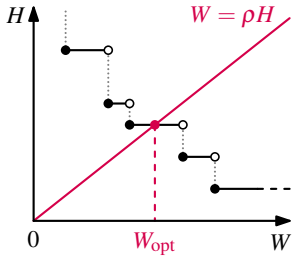


Figure 7: The height of a drawing is a descending function of its width.

Since the function $H(W)$ 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.

4 EXPERIMENTAL RESULTS

In this section we illustrate the practical use of our method with examples from the area of business process analysis, and more specifically, from process mining. Process mining analyzes sequences of events which are collected in event logs. Each event needs to be identified by (at least) a time stamp, an event name, and a case identifier, which identifies distinct event sequences in a log. For example, a case identifier may be the patient being treated in a hospital, the application number for decisions at municipalities, or the user performing the actions in a corporate system. Process mining analyzes event logs, trying to reconstruct the underlying (business) process that created the events.

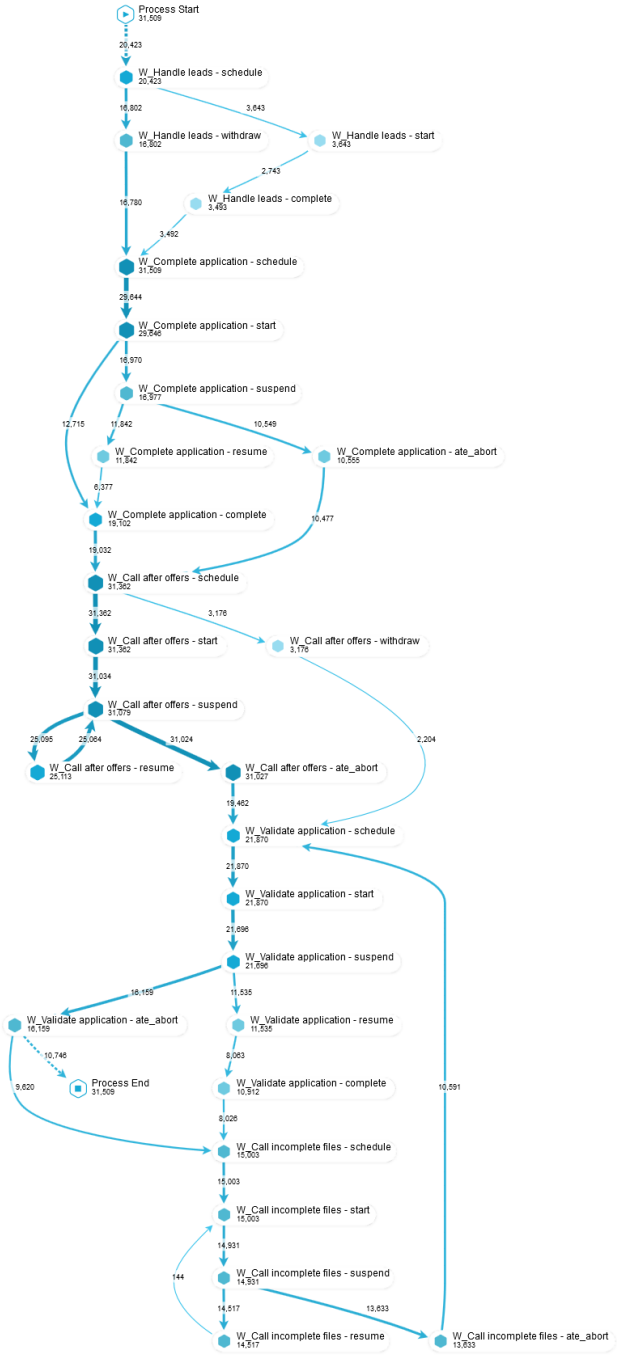


Figure 8: 2017 BPI Challenge: visualization by Celonis [5].

Our running example is the event log from the 2017 Business Process Intelligence Challenge [4]. This log file records the process of handling loan applications, and corresponding offers, in a financial institution. It consists of a total of 31,509 cases (identifiers) and 1,202,267 events. There are three types of events: application state changes (A), offer state changes (O), and work flow events (W). We focused on the work flow events and removed events of type A and O. The filtered log file contains 31,509 cases and 768,823 events; in the following 'log (file)' means the filtered log file.

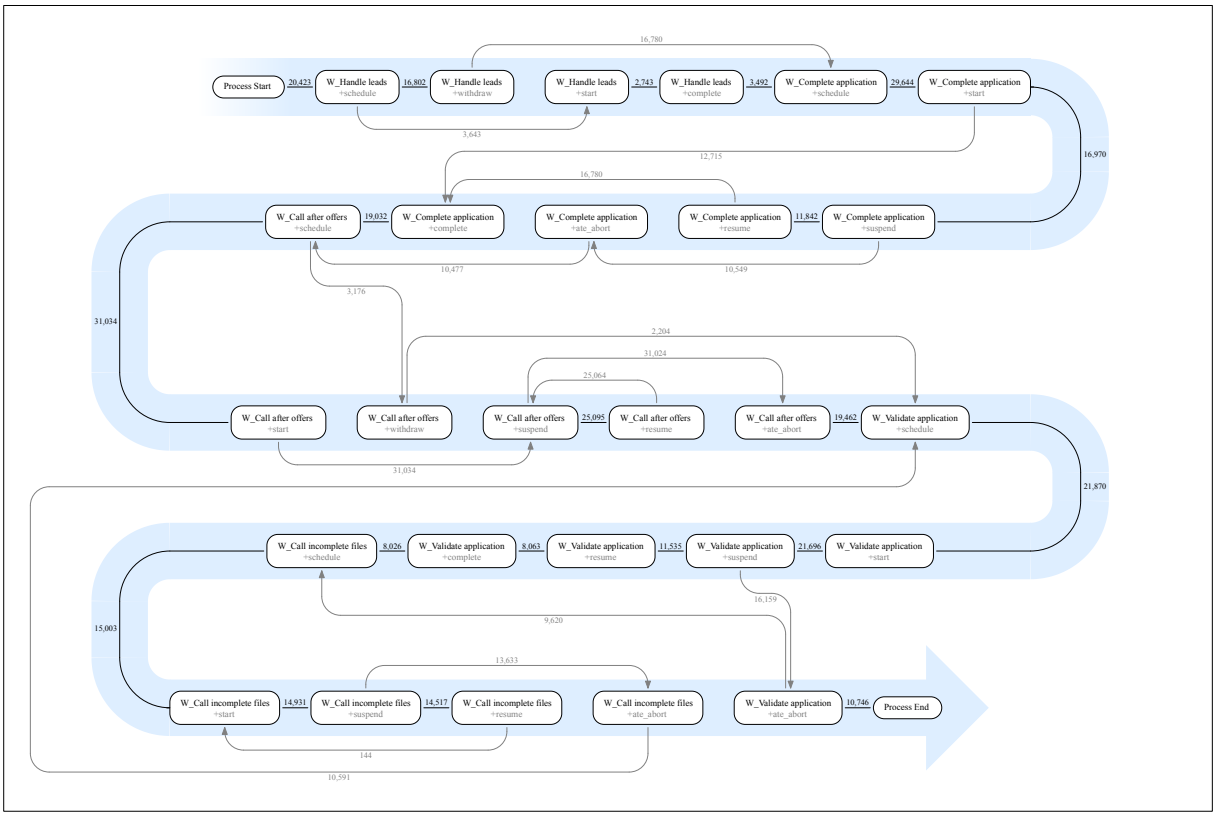


Figure 9: 2017 BPI Challenge: our algorithm applied to the 'Celonis graph' from Fig. 8.

Fig. 8 shows the log file as visualized by Celonis [5], a popular commercial process mining tool. Celonis supports only top-to-bottom linear layouts. The graph which Celonis shows is extracted from the log file using their own proprietary process mining algorithm. Celonis has two simple sliders to control this algorithm: the activities slider, which we set to 98.5%, and the connections slider, which we left on the default setting, which results in 83.2% of the connections being shown. The numbers on each edge indicate the frequency of the corresponding connection in the log file.

We then (manually) extracted the graph from Celonis and redrew it using our algorithm (see Fig. 9). A comparison of the two drawings shows that our algorithm uses the available space more effectively and hence makes it easier for the user to read the labels and follow the main thread of the process. In Celonis the start and end vertices are marked with special symbols. The start vertex is at the top of the drawing, but the end vertex not at the bottom and caught inside a loop. Our method clearly places the corresponding vertices at the beginning and end of the spine, thus more clearly communicating the linearity of the process. Finally, we can still control the aspect ratio of the drawing due to our folding algorithm, while the drawing in Celonis is necessarily high and narrow.

4.1 Process trees

Celonis (and other commercial tools) represent processes fairly directly. However, there are various other representations for process models, many of which have a comparatively clear notion of order, up to events that are performed in parallel. Such parallel events can readily be captured by our versatile block structure, allowing us to visualize the linear structure of complex event logs in an effective way. We illustrate this fact using *process trees* [15], a particular type of process model, which has a recursive structure.

Below we first define process trees. Then we show how to preprocess a process tree to match nodes in the tree to blocks. We also explain our visual encoding of blocks to reflect the corresponding process tree node types. Last but not least we describe how to assign heights to blocks and showcase our final drawing (see Fig. 14).

Definition. Let A be the set of actions that can be executed by a process. Process trees are rooted trees (see Fig. 10 for an example), where the semantics of each node are as follows [15]:

Empty (τ) The empty process.

Action Executing an action $a \in A$.

Sequence ($\rightarrow$) Executing the subtrees $T_1, \ldots, T_k$ in order.

Loop ($\circlearrowleft$) Executing the first subtree T_1 repeatedly, with executing one of $T_2, \ldots, T_k$ between every two executions of T_1.

Choice ($\times$) Executing one of the subtrees $T_1, \ldots, T_k$.

Parallel ($\wedge$) Executing the subtrees $T_1, \ldots, T_k$ in parallel (the actions are interleaved).

One can construct process trees from log files using the *inductive miner* by Leemans *et al.* [12]. The inductive miner, along with many other mining algorithms, has been implemented in the open-source process mining toolkit ProM [16]. This toolkit also visualizes process trees, using a (mostly) linear layout of a graph that represents the process tree (see Fig. 11). This visualization suffers from the aspect ratio problems mentioned earlier. Furthermore, it is difficult to see at a glance how the edges are directed.

We could now simply redraw the graph in Fig. 11 using our folding algorithm to increase its readability. Instead, we use the versatility of our algorithm and the block structure to draw a significantly more compact and clear representation of the process tree. To that end, we first convert the process tree into a sequence of blocks.

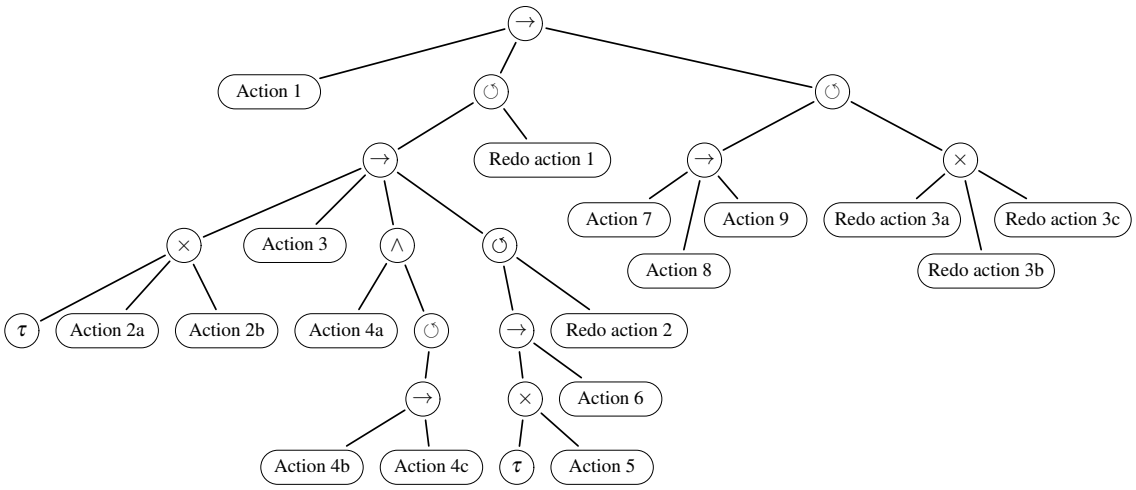


Figure 10: Example of a process tree.

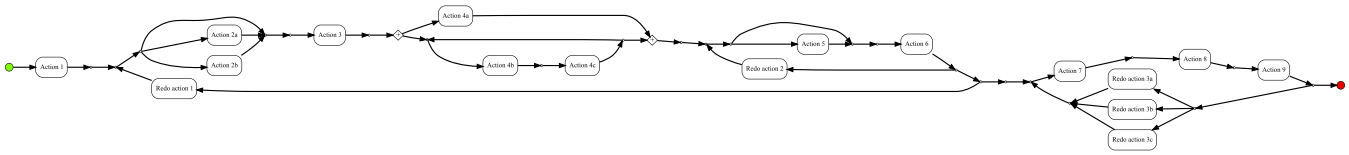


Figure 11: The output of ProM [16] for the process tree in Fig. 10.

Preprocessing. For every node v in the process tree we construct a block $B(v)$. Since the structure of a process tree is recursive, the blocks representing nodes in the process tree are recursive as well. Fig. 12 shows the visual encoding of the node types.

One special case is the subtree $\times(T_1, \tau)$, which occurs very often in process trees generated by the inductive miner. This subtree represents the process that either performs T_1, or it does not do anything (τ). In other words, T_1 is executed optionally. To show this more clearly, we do not generate a choice block containing τ and $B(T_1)$; instead, we generate an optional version of $B(T_1)$. Optional blocks are encoded by adding a dashed outline (see Fig. 12).

The block for the entire process is now given by $B(r)$, where r is the root of the process tree. However, now our entire process is represented by only one block, which would mean that the folding algorithm is not able to fold anything. To enable the power of our algorithm, we partially need to flatten the tree. We can naturally flatten sequence nodes into a linear sequence of blocks. We can also flatten a loop node by adding connectors (as backedges) between the last and first block in the first subtree. Note that here we need the flexibility to draw blocks on connectors, which we can handle by allocating more space for a connector (see Section 3.2). We do not flatten the remaining types of nodes (choice and parallel), but we draw the subtrees recursively (see Algorithm 1). We again consider a subtree $\times(T_1, \tau)$ as a special case and flatten it as well. We then add the dashed block after running the folding algorithm. This way we can easily draw this block as a rectilinear polygon, potentially spanning multiple rows of the folding.

Algorithm 1: TREE-TO-BLOCKS(u)

if *u is a sequence node* $\rightarrow(v_1, \ldots, v_k)$ **then**
  for $i = 1, \ldots, k$ **do** TREE-TO-BLOCKS(v_i)
else if *u is a loop node* $\circlearrowleft(v_1, \ldots, v_k)$ **then**
  TREE-TO-BLOCKS(v_1)
  first, *last* ← first and last block added in the previous call
  for $i = 2, \ldots, k$ **do**
    add connector from *last* to *first* with block $B(v_i)$
else if *M is an action, choice or parallel node* **then**
  add block $B(u)$

Drawing blocks. To apply the folding algorithm to our list of blocks, we need to compute the size (width, top-height, and bottom-height) for each block. This computation depends on the block type. The width and height of an action block are determined by measuring the size of its label and adding a margin to it. For the other block

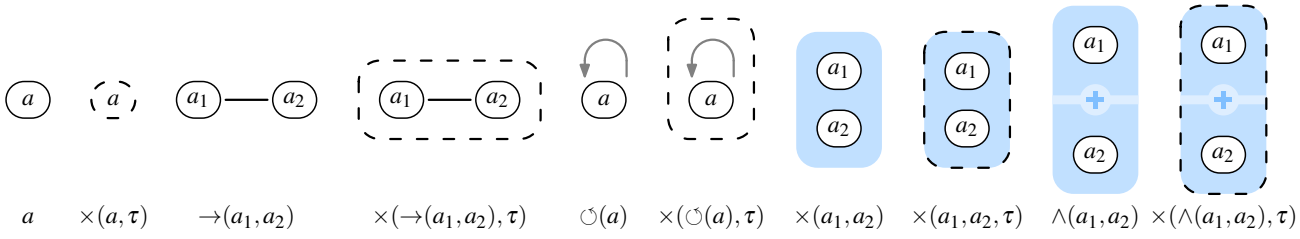
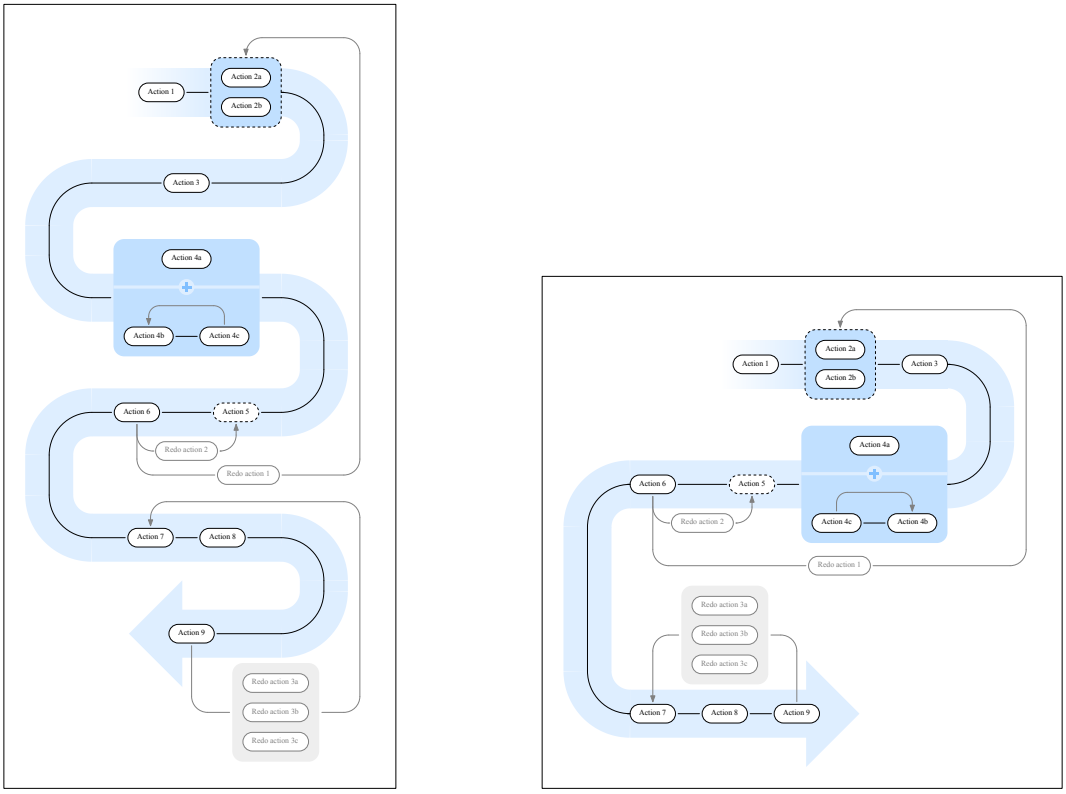


Figure 12: Block types (left-to-right): action, sequence, loop, choice, and parallel block; optional variants with dashed outlines.



(a) Aspect ratio 1/2.

(b) Aspect ratio 1.

Figure 13: Example process tree (Fig. 10): our algorithm for two aspect ratios.

types, we recursively layout their children and put them together to obtain a layout for the entire block. For example, a choice block is drawn by stacking its children on top of each other, so the width of a choice block is the maximum width of its children, and its height is the sum of the heights of its children, plus some margin. After we know the sizes of the blocks, we add a margin to every (top-level) block in the list. This margin keeps blocks separated, so that we can draw spine edges between the blocks. Two resulting drawings with different aspect ratios are shown in Fig. 13. It is apparent that our drawings are not only more flexible with regard to aspect ratio, but also more readable than the standard drawing produced by ProM.

The 2017 BPI Challenge. We now return to the 2017 BPI Challenge log. We used the inductive miner as implemented in ProM, with noise threshold 0.5, to construct a process tree. We then preprocessed the process tree as described in the proceeding paragraphs. Fig. 14 shows the result. Our drawing communicates the linear flow of the process well, and the blocks clearly indicate the features of the process. Furthermore, our drawing gives an organized and legible overview of the process at a glance.

4.2 Timeline

Our second example is a timeline of the Roman emperors from Augustus to Diocletian (see Fig. 15), as listed in the *List of Roman emperors* on Wikipedia.[1] The shade of gold encodes the length of each emperor's reign; we used dotted rectangles to group emperors whose reign overlapped for a significant period of time. Connectors indicate groups of related emperors. Without folding, this drawing would have aspect ratio approximately 20.

[1] `https://en.wikipedia.org/wiki/List_of_Roman_emperors`, accessed 2017-12-23.

5 Discussion and Conclusion

We presented a new algorithm that can optimally fold linear layouts. Regardless of the aspect ratio of the drawing area, our algorithm makes efficient use of space, which increases the label sizes and generally improves the legibility of the drawing. By using blocks to represent the space needed by vertices, our algorithm is also very versatile and produces drawings that are clear and organized.

Due to the versatility of our algorithm, we can even support (recursive) drawings inside blocks. However, we are not able to do so optimally yet. Our algorithm requires that the sizes of the blocks are known beforehand. If a block contains another drawing, this drawing can be folded in different ways, resulting in different dimensions (aspect ratios) for the block. A wider block may reduce the height of the row the block is on, but it can also cause fewer blocks to fit on a row. To minimize the height of the drawing, we could try all possible aspect ratios of the block, but this is not efficient, especially if multiple blocks contain (recursive) drawings. We do not know how to minimize the height of the drawing efficiently under these circumstances. We leave this as an open problem.

Finally, we have shown that our algorithm works particularly well for visualizing processes, and process trees in particular. However, there is some information about the processes that we are currently not visualizing, like frequencies (how often do events happen?) or deviations (which things happen in the log that violate the structure of the process tree?). Although our algorithm has the versatility to show additional information on connectors, it is not directly clear what the best encoding of this information would be. This is outside of the scope of this paper, but we believe that this is interesting future work in the context of visualizing processes or process trees.

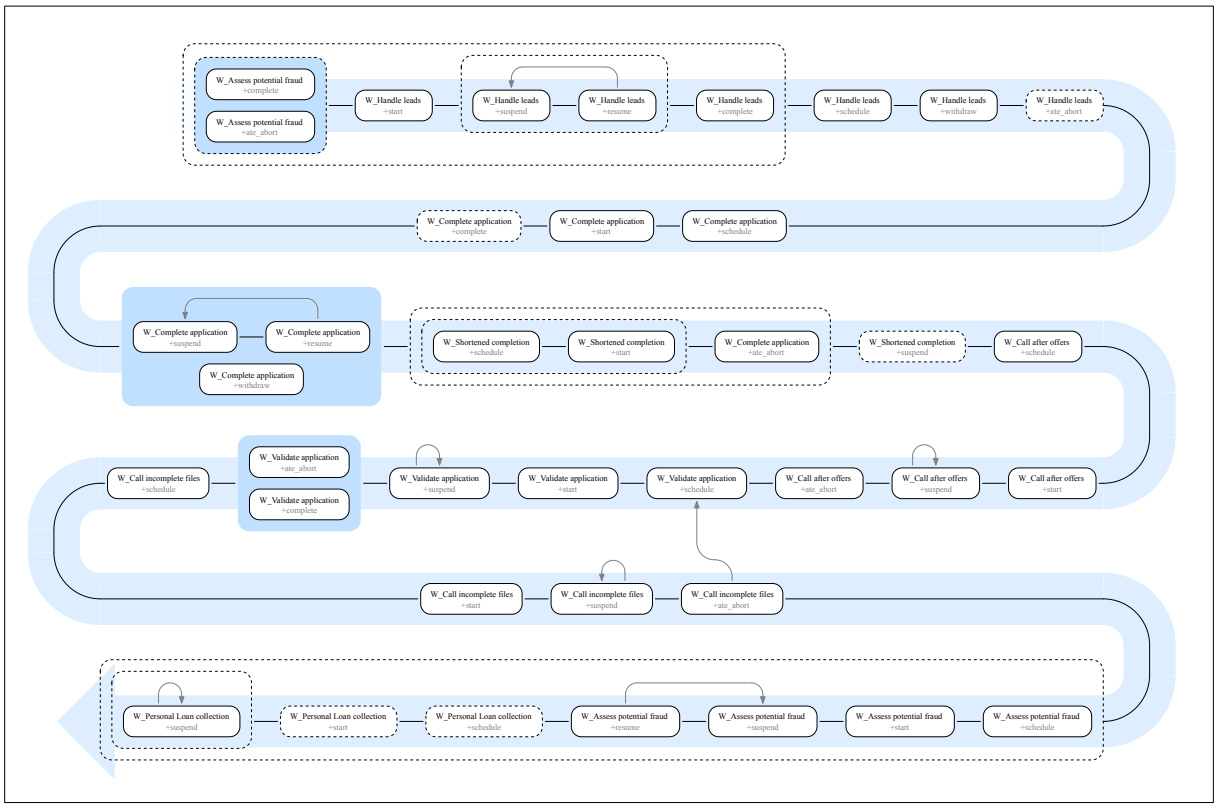


Figure 14: 2017 BPI Challenge: output of our algorithm drawing the process tree extracted with the inductive miner in ProM, aspect ratio 1.5.

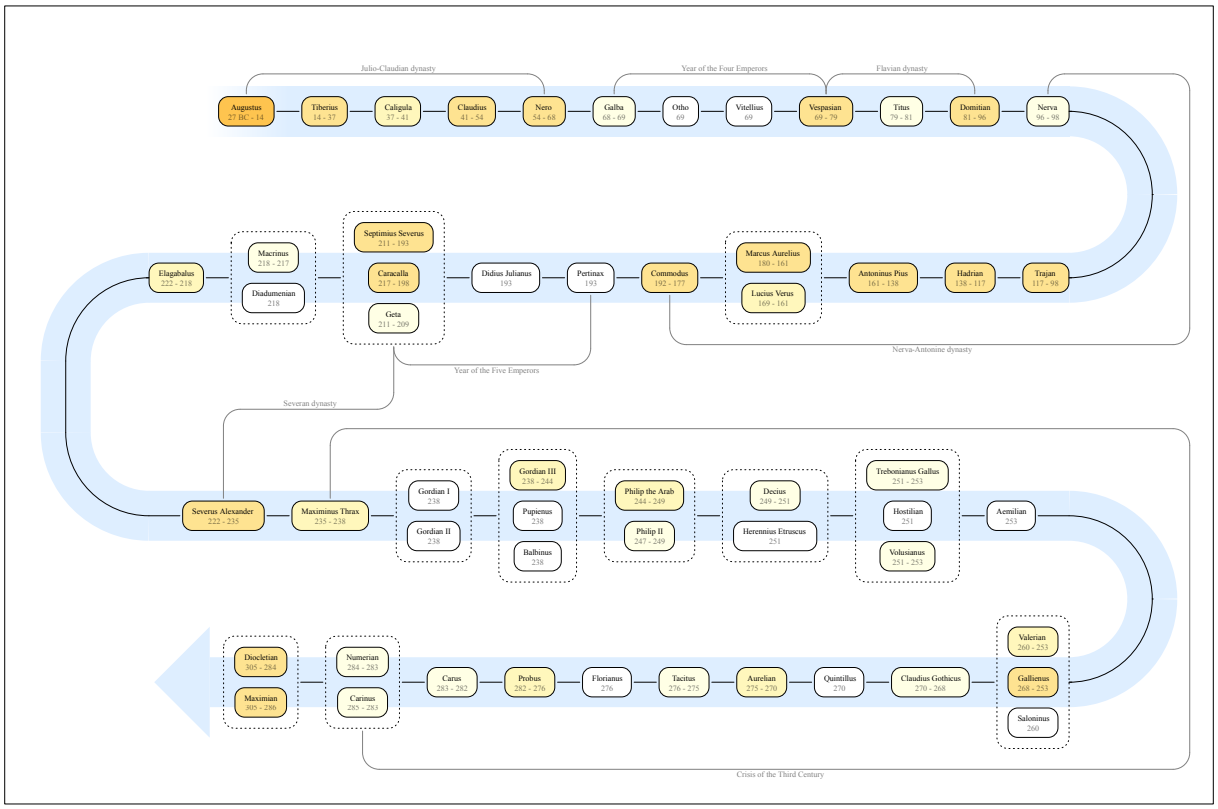


Figure 15: Timeline of Roman emperors from Augustus to Diocletian, drawn using our algorithm with aspect ratio 1.5.

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] B. S. Baker and J. S. Schwarz. Shelf algorithms for two-dimensional packing problems. *SIAM Journal on Computing*, 12(3):508–525, 1983.

[2] A. Bettinelli, A. Ceselli, and G. Righini. A branch-and-price algorithm for the two-dimensional level strip packing problem. *4OR*, 6(4):361–374, 2008.

[3] BPI Challenge 2012. `http://www.win.tue.nl/bpi/doku.php?id=2012:challenge`.

[4] BPI Challenge 2017. `http://www.win.tue.nl/bpi/doku.php?id=2017:challenge`.

[5] Celonis. `https://celonis.com`.

[6] J. Csirik and G. J. Woeginger. On-line packing and covering problems. In A. Fiat and G. J. Woeginger, eds., *Online Algorithms*, pp. 147–177. Springer, 1998.

[7] V. Dujmović and D. R. Wood. On linear layouts of graphs. *Discrete Mathematics and Theoretical Computer Science*, 6:339–358, 2004.

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

[9] 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.

[10] M. Hofri. Two-dimensional packing: Expected performance of simple level algorithms. *Information and Control*, 45:1–17, 1980.

[11] B. Korte and J. Vygen. *Combinatorial Optimization*, chap. Bin-Packing, pp. 426–441. Springer, 2005.

[12] S. Leemans, D. Fahland, and W. van der Aalst. Discovering block-structured process models from event logs – a constructive approach. In *International Conference on Applications and Theory of Petri Nets and Concurrency*, pp. 311–329, 2013.

[13] S. Martello, M. Monaci, and D. Vigo. An exact approach to the Strip-Packing problem. *INFORMS Journal on Computing*, 15(3):310–319, 2003.

[14] S. Masuda, K. Nakajima, T. Kashiwabara, and T. Fujisawa. Crossing minimization in linear embeddings of graphs. *IEEE Transactions on Computers*, 39(1):124–127, 1990.

[15] W. M. P. van der Aalst. *Process Mining: Data Science in Action*. Springer, 2012.

[16] H. M. W. Verbeek, J. C. A. M. Buijs, B. F. van Dongen, and W. M. P. van der Aalst. XES, XESame, and ProM 6. In P. Soffer and E. Proper, eds., *Information Systems Evolution (CAiSE Forum 2010)*, pp. 60–75. Springer, 2010.

A Overlapping rows

Theorem 1. *In the setting where rows are allowed to overlap in their y-coordinates, the problem of minimizing the drawing height is NP-hard.*

Proof sketch. By reduction from 3-SAT (see Fig. 16). 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. □

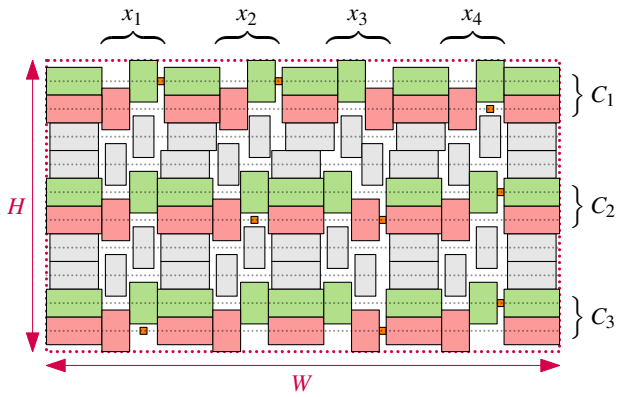


Figure 16: Instance corresponding to the 3-SAT formula $\underbrace{(x_1 \vee x_2 \vee \neg x_4)}_{C_1} \wedge \underbrace{(\neg x_2 \vee \neg x_3 \vee x_4)}_{C_2} \wedge \underbrace{(\neg x_1 \vee \neg x_3 \vee x_4)}_{C_3}$.