Graph layout stability in process mining

Mennens, Robin

Award date:
2018

Technische Universiteit
Eindhoven
University of Technology

Department of Mathematics and Computer Science
Visualization Research Group

Graph layout stability in process mining

Master Thesis

Robin J.P. Mennens

Supervisors:
Michel Westenberg
Roeland Scheepens

Thesis Committee:
Wouter Meulemans

Eindhoven, September 2018

Abstract

Process mining allows organizations to gain insight into their internal/external process(es) and thereby make data-driven decisions. Visualization of this internal/external process, which is done using a graph layout, plays an important role in process mining. Ideally, the process visualization satisfies two properties. Firstly, it should be of high quality, i.e., the graph layout is readable and understandable, and thereby properly shows the underlying process. Secondly, since process analysis is often an iterative task of interactively filtering the process data, the mental map of the user should be preserved. The current industry standard used for process visualization, is *dot*, a Sugiyama-based hierarchical graph layout algorithm. *Dot*, however, often fails to satisfy these two properties. In this work, we present the global ranking and global order, which are computed based on the process data, and are used during the graph layout computation. Additionally, we present a novel crossing minimization algorithm that satisfies the order constraints specified by the global order. Finally, we use phased animation to further improve mental map preservation. A quantitative and qualitative evaluation show that our approach is better at preserving the mental map of the user and computes layouts of high quality (compared to *dot*). Additionally, our approach is significantly faster than *dot*, especially for graphs with more than about 250 edges.

Preface

The results of my master thesis *Graph layout stability in process mining*, which was performed at ProcessGold (Eindhoven, The Netherlands), are presented in this report. The master thesis forms the last component of my master Computer Science and Engineering, which is provided by Eindhoven University of Technology.

I would like to thank my supervisors Michel Westenberg and Roeland Scheepens for their supervision, guidance, and extensive feedback throughout. The work of this thesis would not have been possible without your supervision. Additionally, I would like to thank Wouter Meulemans for being part of the assessment committee. Also, thanks to all my ProcessGold colleagues for their openness, enthusiasm, and great atmosphere. Thanks to Roel Vliegen for all the interesting/helpful discussions and brainstorm-sessions. Finally, I would like to thank my family for supporting and encouraging me throughout.

Eindhoven, Wednesday September 5, 2018
Robin Mennens

Contents

Chapter 1

Introduction

Process mining [Aal16] is a field that is quickly gaining popularity in many areas. With process mining, organizations are moving from a 'gut-feeling' approach, in which organizations have to spend many hours to understand their internal/external process(es), towards a fact-based approach, where organizations can make data-driven decisions based on facts. Visualization of the process, which allows organizations to analyze their internal/external process(es), plays an important role in process mining. Process visualization is done by transforming the process data into a directed, weighted (process) graph for which a hierarchical graph layout [STT81] is computed. The visualization of this graph layout allows a user to analyze the process, which is often an iterative task of interactively filtering the process data such that only the data of interest is visualized. Consequently, process visualization essentially comes down to computing graph layouts for a sequence of graphs. For this sequence of graphs, there are two important aspects to consider.

The first is that we want an individual graph layout to be of high quality: the layout is readable and understandable, and thereby properly represents the actual process. The graph layouts in Figure 1.1 A1 and A2, for example, are of poor quality. In terms of readability, we observe that some edges overlap, making them hard to read. Understandability is also poor because, in both layouts, node 2 is positioned above node 1, implying that activity 2 occurs before activity 1, while in the actual process, this is not the case. The graph layouts in Figure 1.1 B1 and B2, on the other hand, are of high quality. Both layouts are readable and understandable, and thereby properly represent the actual process.

Secondly, as stated above, a result of the interactive filtering of a user is that we essentially obtain a sequence of graph layouts. For this sequence, we want to preserve the mental map [MELS95, CP96] of the user. Intuitively, the mental map represents the user's mental representation of the data. If the mental map of the user is not preserved, then the user has to reinvest the effort to understand the process (graph). For example, the graph layouts in Figure 1.1 A1 and A2 are computed by the same algorithm. The graph in A2 is the same as the graph in A1, except for the edge highlighted in red. As we can see, by removing a single edge, the layout changes drastically. Consequently, the mental map of the user is not preserved. In Figure 1.1 B1 and B2, we again have the same two graphs and the same edge is removed. But now, our novel graph layout algorithm is used to compute the graph layouts. As we can see, the graph layout in B2 barely changes compared to the graph layout in B1. Consequently, the mental map of the user is preserved. In general, there are two approaches towards preserving the mental map of the user [Bra01]. The first, is to minimize the change between consecutive layouts, i.e., to ensure graph layout stability [AP13]. The second, is to transition between two consecutive graph layouts, allowing the user to 'follow' the layout changes.

Computing graph layouts that are both of high quality and stability is not (always) possible. In order to obtain stable graph layouts, we have to minimize change. Minimizing change, however, means that we cannot always optimize layout quality. Finding the right balance between quality and stability, is one of the main issues addressed in this work.

1.1 Process Mining

Process mining consists of different techniques used to discover, monitor, and improve real processes.

"The goal of process mining is to use event data to extract process-related information" [Aal16].

Such event data is obtained during the execution of the process and is stored in a so-called event log. So, essentially, the data in the event log describes how the process actually takes place. By visualizing this data in a graph layout, the user can gain insight into the process. Obviously, we want this graph layout to properly represent the actual process. Existing graph layout algorithms, however, often fail to do this. The main reason is that the event log is essentially 'translated' into a graph. Consequently, graph layout algorithms can only use the information in the graph structure, and not the process data itself. Therefore, critical process information is lost, resulting in graph layouts that poorly represent the actual process. In Figure 1.1 A1 and A2, we have seen examples of poor process representation. In this work, we tackle this problem by bringing the areas of process mining and graph drawing together.

The work of this thesis was done at ProcessGold, a company that develops a business intelligence and process mining platform which can be used to create process analysis applications. Part of this platform is the process graph visualization, which shows the process data. Additionally, different interaction techniques such as filtering, zooming, and panning allow the user to interactively change/analyze the visualized process data. All algorithms presented in this work are implemented for this process graph visualization. Additionally, note that the visual attributes (as shown in Figure 1.1) of the nodes and edges, like color, shape, and edge width, are inherent to the ProcessGold platform, and therefore out of the scope of this work. In general, darker shades of blue imply a more frequent occurrence of a node/edge. Similarly, wider edges occur more frequently.

Throughout this work, we use a dataset called Invoices as a running example. This dataset contains process data of a process that includes receiving, approving, and paying invoices.

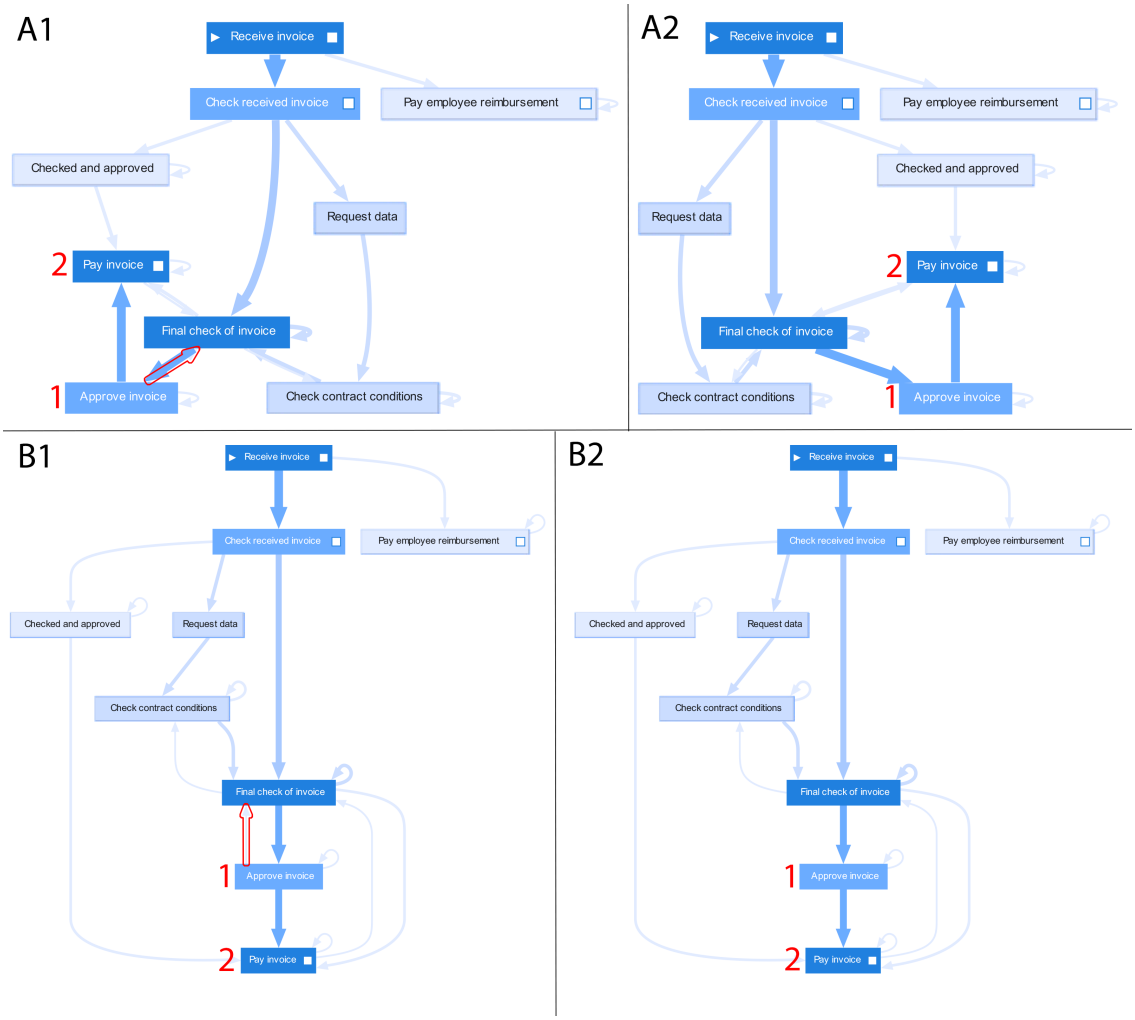

Figure 1.1: Illustration of four graph layouts. A1 and A2 are computed by the same algorithm and are of poor quality: edges overlap and the actual process is poorly represented. B1 and B2 are computed by our novel graph layout algorithm and are of high quality: edges are easy to follow and the process is properly represented. A2 is obtained after removing the edge highlighted in red from A1. As we can see, A2 differs significantly from A1. Consequently, the mental map of the user is lost. Similarly, B2 is obtained after removing the edge highlighted in red from B1. As we can see, B2 barely differs from B1. Consequently, the mental map of the user is preserved.

1.2 Graph Layout Stability and the Mental Map

In order to reduce the cognitive effort of the user, we want to preserve the mental map [MELS95, CP96] of the user. In other disciplines, the mental map is also referred to as the cognitive map [Kit94], which essentially represents the user's mental representation of the data. In the area of graph drawing, however, the mental map is often defined in terms of the change between two graph layouts. The most commonly used definition is the definition of Coleman and Parker [CP96]:

"The placement of existing nodes and edges should change as little as possible when a change is made to the graph."

Other definitions of the mental map also exist. Therefore, in Chapter 3, we provide an in-depth analysis of research done into the mental map, and we define what we regard as the mental map.

As stated above, one technique which can be used to preserve the mental map of the user, is to transition between two consecutive graph layouts [Bra01]. In Section 1.4.1, we discuss several transition techniques. Another mental map preservation technique, is to minimize the change between two consecutive graph layouts, i.e., to ensure graph layout stability [AP13]. By minimizing the change between two graph layouts, we make sure only a limited number of nodes/edges is modified. Consequently, it is easier for the user to track these changes.

Both above-mentioned approaches already aid mental map preservation. Transitioning, however, can still be confusing if there are many changes. For example, a user can only follow up to 5 moving elements at the same time [PS88]. And even when changes are minimal, a transition without a transition technique can still be hard to follow. Therefore, in this work, we apply both techniques simultaneously for the best results.

1.3 Problem Description

Given an event log containing data that describes a process, we want to visualize this process using a weighted, directed graph layout. Moreover, we want layouts to be of high quality: they should be readable and understandable, and thereby show the underlying process. Additionally, the interactive filtering of the user changes the visualized graph layout. Under this interaction, we want to preserve the mental map of the user. Finally, we have the requirement that the graph layout algorithm should be deterministic: given a graph, we always want to compute the same layout for this graph, regardless of the previous graph layout.

In order to address these problems, we provide the following contributions:

- A novel ranking algorithm that computes the global ranking, which helps improve layout quality and layout stability (and thereby mental map preservation). (Chapter 4)
- A novel order constraint computation algorithm that computes the global order, which helps improve layout stability (and thereby mental map preservation). (Chapter 4)
- A novel crossing minimization algorithm that makes sure the order constraints of the global order are satisfied. (Chapter 4)

Additionally, in Chapter 6, we present a phased animation transition technique which further improves mental map preservation.

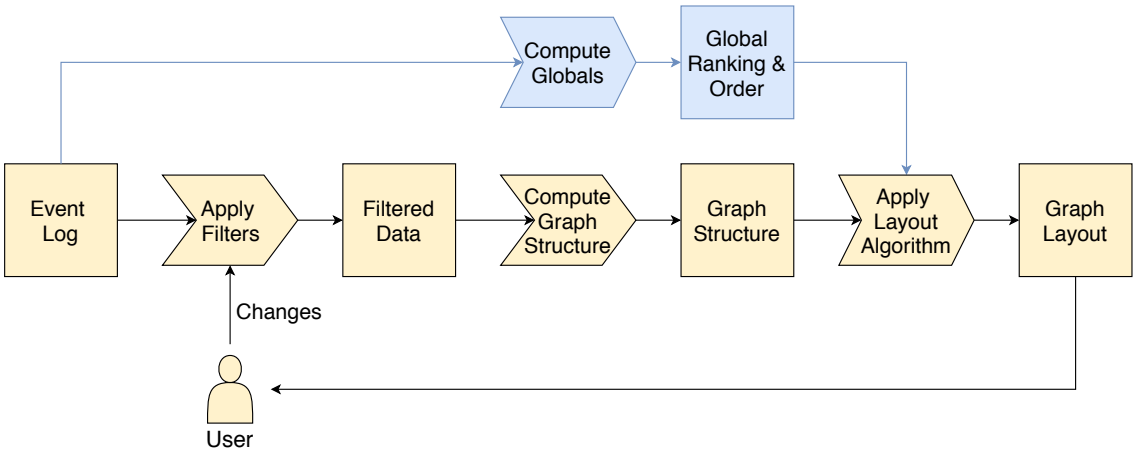

Figure 1.2: An overview of the system. Beige elements represent the current system while blue elements represent our contribution.

1.4 System Overview

Figure 1.2 contains an overview of the system. Elements in beige represent how the system currently works while blue elements highlight our novel contributions. The current process starts with an event log, which contains the process data that was stored while the process was running. The user can interactively apply filters to the event log such that only the data of interest remains. After that, the filtered data is converted into a graph structure. Activities, which are recordable steps in the process, are converted into nodes while the occurrence of two consecutive activities is converted into an edge. This conversion results in a weighted directed graph in which the weight of an edge (A, B) represents how often activity A occurred before activity B in the process. Finally, a graph layout algorithm computes a hierarchical layout [STT81] for the graph structure, which is shown on the screen. This whole process is repeated when the users applies new filters.

Given the current system, we observe two issues. The first relates to the computation of graph layouts that are readable and understandable, and thereby properly show the underlying process. We observe that there is no direct connection between the (filtered) process data and the graph layout algorithm. More specifically, the graph layout algorithm only obtains the graph structure, and not the process data. Consequently, the current system often fails to properly visualize the actual process. The second issue relates to mental map preservation. In the current system, there is no functionality that makes sure the changes between graph layouts are minimal (and transitioning is not implemented).

To address these issues, we introduce the concepts of a global ranking and a global order. Both of these are computed based on the event log. Consequently, by using the global ranking and global order in the graph layout algorithm, we are able to compute graph layouts that show the actual process. Additionally, the global ranking and global order essentially define 'global' constraints that are applied to every graph layout computation. By doing this, we make sure changes between two consecutive graph layouts are minimal. Finally, transitioning (which is not shown in Figure 1.2) is implemented to further improve mental map preservation.

1.4.1 Graph Transition

One of the approaches towards mental map preservation is to transition between different graph layouts [Bra01]. Three commonly used transition techniques are small multiples, difference maps, and animation. In the sections below, we discuss each of these techniques in more detail. After that, we discuss usability research done for animation and small multiples.

Small Multiples
In a small multiples approach, a matrix of graph layouts is used to show how the graph evolves over time [APP11]. The main advantage of this method is that users are able to see and compare all graph layouts at the same time. On the other hand, by showing all graph layouts at the same time, it may be difficult to understand how the graph evolves. The biggest disadvantage, however, with respect to our work, is the fact that the available screen space per graph layout is limited. Processes can be quite complex/large, resulting in large graph layouts. Therefore, we want as much screen space as possible to make these graph layouts as readable and understandable as possible.

Difference Maps
A difference map is an aggregation of two graph layouts G_1 and G_2 where colors are used to indicate the similarities and differences between the two graph layouts [APP10]. Consequently, the difference map represents G_1 and G_2 at the same time and allows the comparison of the two. Observe that this approach is not deterministic. For example, if we have the same graph G_2, but different graphs G_1, then we obtain different difference maps. Since we want a deterministic graph layout algorithm (see our problem description in Section 1.3), this is a major drawback of difference maps. Another disadvantage, is that an aggregation of two graph layouts can quickly become quite large, which decreases the readability and understandability of the computed graph layouts. Additionally, as stated before, process analysis is often an iterative task where users interactively filter the data such that only the data of interest is shown. Using a difference map would not work, because a difference map would show both the current and previous filtered data of interest. Obviously, this is not desirable. Therefore, considering these drawbacks, this approach is not applicable to our work.

Animation
In an animation between two graph layouts, the first graph layout is morphed over time until we obtain the second graph layout. An obvious drawback of animation is that it takes time [BB99]. The animation itself, however, helps the user understand how the graph structure evolves. Additionally, while only one graph layout is visualized at the same time, the whole screen can be used for this graph layout.

Usability Research
As stated above, difference maps are not applicable to our work. Considering the pros and cons of small multiples and animation, however, it is not immediately clear which technique works best. To decide between animation and small multiples, we consider research done into the usability of the two techniques.

For undirected graphs, Archambault et al. [APP11] tested the difference between no and high mental map preservation for both animation and small multiples presentation methods. Degree reading, graph appearance, graph growth, and path reading tasks were used in their experiment. Considering the difference between animation and small multiples, they found that user response is significantly faster when using small multiples. On the other hand, animation resulted in significantly fewer errors when the appearance of certain nodes and edges had to be determined. Overall, the authors suggest that when accuracy is more important than response time, animation should be used. Considering mental map preservation, the authors found no significant difference between the two techniques. In the context of social network visualization, Farrugia et al. [FQ11]

test the effectiveness of animation and small multiples for degree reading and edge detection tasks. Their results indicate that user response is faster (and sometimes more accurate) when small multiples are used. The main problem with the two above-mentioned works, is that they consider unnamed nodes. In these works, nodes of interest are highlighted using colors [APP11], or by using a combination of colors and shapes [FQ11]. Consequently, it is easier for the user to identify the node(s) of interest in a (static) graph layout [RM13]. In the work of Zaman et al. [ZKS11], this problem is addressed by considering named nodes in dynamic hierarchically drawn directed acyclic graphs. In their work, the DARLS system is introduced, which allows users to compare two hierarchically drawn graphs. In an experiment, users had to select nodes that changed position between the two graph layouts. The authors found animation to be superior to static graphs (small multiples). Archambault et al. [AP12b] present an experiment in which the ability of users to orient themselves in dynamically evolving graphs is tested. Node revisitation and path tracing tasks are considered. The authors find that, generally (although not significantly), animation performs better than small multiples. Archambault et al. [AP16] present an experiment where they compare animation and small multiples for graphs that have low stability, i.e., the mental map is poorly preserved. Tasks include node and path tracking. They find evidence that animation can improve task performance when the drawing stability of the graph is low and when nodes of importance cannot be highlighted.

Considering the research above, there is no agreement on which technique performs the best. It appears that the performance of a technique mainly depends on the type of graph and the tasks that are required. Since the work of Zaman et al. [ZKS11], which finds evidence in favor of animation, also considers hierarchical graphs with named nodes, the results of their work are the most applicable to our work. Additionally, considering again our statement the the available screen space per graph layout is limited for small multiples, we decide to use animation as the transition technique. With animation, we can use the complete screen to make the graph layouts as readable and understandable as possible.

1.5 Related Work

Ensuring graph layout stability is a well known problem that has been addressed in many works, especially in the area of dynamic graph drawing [BBDW17]. Dynamic graph drawing generally considers the evolution of a graph over time. For every time-step, a different graph layout needs to be computed. In this area, it is often essential to preserve the mental map of the user. In Section 1.5.1, we consider the approaches of existing works towards ensuring graph layout stability (mostly in the area of dynamic graph drawing). Considering our transition technique, animation of graph layouts can be done in different ways. Therefore, in Section 1.5.2, we consider existing graph layout animation techniques. Finally, since we are visualizing processes, in Section 1.5.3, we consider the field of business process visualization, which contains interesting work regarding visualizing process semantics.

1.5.1 Graph Layout Stability

In general, there are two approaches towards achieving graph layout stability [Bra01]. The first is to limit the movement of nodes, i.e., only certain nodes are allowed to move a certain amount. The second is to use metrics that measure the layout quality and stability. Based on these metrics, algorithms try to compute a layout that optimizes the trade-off between these metrics.

1.5.1.1 Movement restriction

The approaches towards restricting node movement can be further categorized based on the available information and whether or not the approach is independent of the graph layout algorithm used. Each of these different categories is discussed in more detail below.

Generic framework
Generic frameworks are designed such that they define movement restrictions independent of the layout algorithm used. Consequently, layout algorithms are modified such that the movement restrictions, as specified by the generic framework, are taken into account.

Bohringer et al. [BP90] introduce layout constraints. These are linear relations between coordinates of nodes in one dimension. For example, the constraint that node p should be vertically above node q is translated to two layout constraints: $p_x = q_x$ and $p_y < q_y$ (assuming that the y-axis goes from top to bottom). A constraint manager is responsible for maintaining the constraints, keeping the constraints consistent, and responding to constraint addition, deletion, and status queries. The authors show how the constraint manager can be used in a modified implementation of the Sugiyama framework such that graph stability is ensured. First, the layout produced by the layout algorithm is translated into a set of layout constraints. After that, once a change occurs (an edge/node is added/removed), constraints in the vicinity of the change are removed while other constraints remain active. Then, a new graph layout is computed such that all active constraints are satisfied. Consequently, stability is ensured because unchanged parts of the graph layout remain the same, while nodes/edges in the vicinity of a change can be altered. In other work, Paulisch et al. [PT90, Pau93] describe EDGE, an extendible graph editor, which implements the modified layout algorithm and constraint manager. Waddle et al. [Wad00] propose using layout constraints to better display the semantics of data structure graphs. Additionally, by using a similar approach as Bohringer et al. [BP90], their layout constraints can be used to achieve graph layout stability. He et al. [HM98] propose using linear (in)equality constraints, specified by the user, on node coordinates to achieve graph stability. They show how different undirected graph layout algorithms can be adapted such that the given constraints are taken into account.

In general, we believe that layout constraints are an intuitive approach towards restricting node movement. The main problem, however, lies in specifying constraints such that we obtain both graph layout quality and graph layout stability. When there are large changes to the graph, the approach of Bohringer et al. [BP90] will remove (almost) all constraints and therefore stability is lost. User specified constraints [BP90, HM98, Wad00] can be used in order to achieve graph layouts that properly represent the actual process. However, in our work, we assume that the user has no process information yet. In fact, the goal of a user is often to obtain process information by analyzing the computed graph layout.

Online
In an online setting, only information about the previously computed layouts is available. Often, algorithms try to ensure graph stability by using the most recent graph layout as a basis for the new layout to be computed. Hence, the new graph is essentially seen as a set of node/edge insertions/deletions/modifications.

North et al. [Nor95, NW01] introduce the DynaDAG heuristic. DynaDAG extends the Sugiyama framework such that incremental graph changes can be applied while preserving the mental map. Individual node/edge insertions/deletions/optimizations are supported by DynaDAG. If more complex graph changes occur, a combination of these individual techniques can be applied. Miriyala et al. [MHT93] describe a heuristic designed to route new edges in an already existing orthogonal layout without modifying the existing layout. First, the heuristic computes a rectangulation of the existing layout. After that, a variation of Dijkstra's algorithm is used to compute the sequence of rectangles through which the new edge should be routed. This is done such that the

edge length, number of crossings, and number of edge bends is minimized. Cohen et al. [CDBTT95] describe dynamic graph drawing algorithms for trees, series-parallel digraphs, planar ST-digraphs, and planar graphs. Update operations (e.g. node/edge insertions/deletions) to the graph are handled efficiently such that the existing layout is only changed when needed. Thereby, the algorithms aim for a trade-off between layout quality and stability. Saffrey et al. [SP08] extend a spring embedder [FR91] algorithm in order to preserve the mental map of the user. Essentially, when computing a new layout, the movement of existing nodes is restricted in terms of their old and (supposed to be) new position. Two restriction models are proposed. In the proportional model, a node is allowed to move a proportion of the distance between its old and (supposed to be) new position. In the geometric model, a node can only move a maximum geometric distance. By specifying the degree in which the restriction models are applied, the stability can be controlled. Frishman and Tal [FT08] present an efficient force-based layout algorithm which is implemented on the GPU. Layout stability is achieved by restricting node movement. Based on the changes in the graph, the algorithm computes a movement flexibility degree for every node. This degree can then be used to determine which nodes may have to be moved. After that, an approach similar to simulated annealing is used to compute the final node positions.

The main disadvantage of the online approaches [Nor95, NW01, MHT93, CDBTT95, SP08, FT08] is that they compute a new layout based on a previous layout. Because of this, there is no guarantee that we always compute the same layout for the same graph. Also, when graph changes are small, the online approaches are able to ensure stability. When changes are large, however, stability cannot be guaranteed. Moreover, once we have a layout of poor quality, the quality of successive layouts will likely also be poor.

Offline
In an offline setting, the whole (linear) sequence of graphs $G_1, ..., G_n$ for which a layout must be computed is known in advance. More specifically, for every graph G_i, we know G_{i-1} and G_{i+1} (if they exist).

Diehl et al. [DGK01] introduce their Foresighted Layout (FL) framework. FL computes a layout for a super graph by grouping nodes, and edges, such that no group contains two graph elements that are present together in any graph G_i. After that, every individual layout is based on the super graph layout. Diehl and Görg [DG02] introduce Foresighted Layout with Tolerance (FLT), which further extends FL by allowing some node movement (with respect to the previous layout) to improve layout quality. The degree of node movement is computed by a metric and is allowed within some threshold δ. In their work, node movement is implemented using a force-based algorithm. Görg et al. [GBPD04] show how FLT can be used in combination with adapted algorithms for orthogonal and hierarchical layouts. In the hierarchical setting, there are two phases: rank assignment and order assignment. For the rank assignment, the backbone of the super graph is defined as a set of nodes with highest importance. A ranking for these backbone nodes is computed and fixed for every individual layout. All other nodes can be freely assigned ranks, as long as they move within tolerance value δ compared to the previous layout. In the order assignment phase, the order on every rank is initialized to either the same order as in the previous layout, or to some globally defined order. After that, every rank is iteratively reordered in order to minimize edge crossings while preserving the mental map. Note that the approach of Görg et al. [GBPD04] is not deterministic because a graph layout is computed based on the previous layout. For dynamic hierarchical networks, Pohl and Birke [PB08] further extend FLT such that it supports hierarchical structures. A super tree, instead of a super graph, is used as a basis to compute the layout of every individual graph. Furthermore, their algorithm is implemented in XLDN, a visualization tool for large dynamic networks. Reitz et al. [RPD09] extend the work of Pohl and Birke [PB08] with a technique that helps the user focus on the interesting parts of the graph during a graph animation. Erten et al. [EHK+03] present GraphAEL, a system that implements force-based layout methods. In particular, they introduce a method to preserve the mental map. By connecting equivalent nodes in different time slices through virtual edges, the movement of nodes can be restricted.

Because these virtual edges attract the equivalent nodes, nodes are essentially attracted to their position in a different time slice. By setting the weight of these virtual edges, the degree of mental map preservation can be controlled. Baur and Schank [BS08] present Visone, a social network drawing tool that implements a modified version of the Kamada Kawai layout algorithm [KK89]. The modified algorithm adds extra energy factors that represent a node's distance to the positions of equivalent nodes in adjacent time steps. Consequently, a node remains closer to its positions in adjacent time steps, resulting in layout stability. Feng et al. [FWSL12] divide the sequence of graphs into time windows where each time window consists of several consecutive time steps. For every time window, a layout is computed for the corresponding super graph. The initial layout of every individual graph is then based on the layout of the super graph. After that, the initial layout is triangulated and morphed based on node/edge significance. This allows the user to enlarge focused nodes and their neighbourhoods.

All offline techniques [DGK01, DG02, EHK+03, GBPD04, BS08, PB08, RPD09, FWSL12] assume that the complete sequence of graphs is known in advance. While this extra information is advantageous in terms of creating stability, it can also be very expensive when this sequence of graphs changes, i.e., the proposed algorithms have to recompute everything over the modified sequence of graphs. Additionally, in our work, we do not know the complete sequence of graphs beforehand (because this depends on user interaction), we only know (based on the event log) which nodes and edges we can potentially encounter in a graph. Therefore, none of these techniques is directly applicable to our problem.

1.5.1.2 Metric Optimization

Metric optimization techniques use metrics to measure different aspects of layout quality and layout stability. These metrics are often incorporated in a cost function which is then optimized. Consequently, depending on the metrics used, a trade-off between layout quality and layout stability can be achieved.

Brandes and Wagner [BW97] introduce a framework based on Bayesian decision theory, which is independent of the layout algorithm used. The framework describes a cost model that represents the trade-off between individual layout quality and layout stability between layouts and their predecessors. By providing parameters, the weighting between layout quality and stability can be controlled. In their work, they show how their framework can be implemented for a force-directed and an orthogonal layout algorithm. Lee et al. [LLY06] propose a simulated annealing approach with a cost function that also takes mental map preservation into account. By adding metrics that represent the degree of change between layouts, graph stability is also taken into account when computing a new layout. Pinaud et al. [PKL04] introduce a hybridized genetic algorithm, designed to preserve the mental map. For each time step, the algorithm returns a set of potential new layouts (with optimized quality). The layout which is the most similar to the current layout (based on a similarity metric) is then picked as the new layout.

Because different metrics are used in the same cost function, weights [BW97, LLY06] are used to control the influence of every metric. This is a major drawback, because it is not clear which values produce good results. Additionally, the approach of Lee et al. [LLY06] only supports undirected graphs while Pinaud et al. [PKL04] mainly focus on edge crossings and consequently measure quality only in terms of edge crossings. Stability is also measured using only a single metric. Therefore, all of these approaches are either not clear in terms of what produces a good solution, or are not applicable to our problem.

1.5.2 Graph Layout Animation

Animation is a transition technique that can be used to help preserve the mental map of the user [Bra01]. The idea is that by animating the transition from one graph layout to another, the user can 'see' which changes occur. In this work, we use a phased animation technique. Therefore, in this section, we discuss work that uses phased animation in order to (help) preserve the mental map.

Zaman et al. [ZKS11] investigate mental map preservation in the context of dynamic hierarchically drawn directed acyclic graphs. They do this for their DARLS system, which allows users to compare two hierarchically drawn graphs. The animation can be started/stopped by the user using a play/pause button and consists of three phases: first, removed graph elements fade out, then, graph elements are moved to their new position while shape and color properties are updated, and finally, new graph elements fade in. Friedrich et al. [FE00, FE02] present Marey, a system that can morph one drawing of a graph into another without any restrictions on the type of graph or type of layout algorithm used. Animation is done in four phases: first, removed graph elements fade out, then, the complete graph is moved/scaled/rotated towards the new layout, after that, individual nodes are moved to their new positions, and finally, new graph elements fade in. In a later work [FH01], this approach is extended by detecting node clusters that have a similar motion and moving these together. In the area of dynamic networks, Bach et al. [BPF14] present GraphDiaries, a visual interface that uses staged animation, similar to Marey [FE00, FE02], to help the user understand the evolution of the network. To emphasize the removal and addition of graph elements, highlighting is used. Additionally, a timeline can be used by the user to control the animation. Frishman and Tal [FT08] present a stable force-based layout algorithm designed for online dynamic graphs. They use phased animation to transition between different graphs in order to (help) preserve the mental map. Reitz et al. [RPD09] present a system that implements animation for dynamic compound graphs. Animation is used to help users focus on important parts of the graph. The authors argue that movement is easy to recognize when it happens in a small area and when the number of moving objects is small. Therefore, they partition the set of nodes to be moved into 'animation groups'. Each of these groups is then animated separately in order to help the user keep track of the changes. Loubier et al. [LD08] present VisuGraph, a system in which a super graph is computed to present an overview of all data. When the user transitions between two graphs, instead of directly morphing the first graph to the second, VisuGraph first morphs to the super graph to 'recalibrate' the mental map.

Considering the above-mentioned works, all of them implement some form of phased animation. Some works [FE00, FH01, FE02, FT08, ZKS11, BPF14] are, because they also implement fade out, move, and fade in phases, more similar to our work than others [LD08, RPD09]. A difference with all above-mentioned works, however, is that none of them implement viewport animation during the move phase. Also, since our edges are drawn as splines, we implement spline morphing. The above-mentioned works, on the other hand, consider straight line edges.

1.5.3 Business Process Visualization

In the field of business process visualization, graph drawing techniques focus on optimizing layout semantics for processes. Often, process information is provided in the Business Process Execution Language (BPEL) [AAA+07] or Business Process Model and Notation (BPMN) [CT12] language.

Yang et al. [YLS+04] use an implementation of the Sugiyama framework to compute an initial layout for a workflow graph. To preserve the mental map under node insertions/deletions, the authors propose a force-scan algorithm. By applying forces on each axis between node-pairs, node overlaps are resolved and the orthogonal ordering of nodes is preserved. The authors do not discuss how to

handle edge insertions/deletions, however. Diguglielmo et al. [DDK+02] present ILOG JViews, a Sugiyama based workflow graph layout algorithm, implemented for the ILOG JViews graph layout module [SV01]. Their algorithm contains an incremental mode, in which the relative positions of nodes are preserved when the graph changes. Hence, small graph changes do not cause a significant rearrangement of the graph layout. Both above mentioned approaches [YLS+04, DDK+02] are incremental, because the next layout is based on the previous. Therefore, these techniques have the same disadvantages as those mentioned for the online approaches in Section 1.5.1.1. Effinger et al. [ESK09] present their 'BPMN-Layouter' tool, which is designed to compute aesthetically pleasing layouts for business processes. Additionally, they consider the preservation of the mental map. By using a Sketch-Driven-Layout approach, based on a Bayesian paradigm [BW97], the layout is incrementally changed while preserving the topology of the underlying model. Since a Bayesian paradigm is used, this approach has the same disadvantages as those mentioned for the metric based approaches in Section 1.5.1.2. Kabicher et al. [KKRM11] introduce Change Tracking Graphs (CTG) for directed series-parallel process graphs. Given a graph and a modified successor graph, the CTG is a union of both graphs. Differences between the two graphs are highlighted using green and red colors. By using the layout of the first or successor graph as a basis, the authors aim to preserve the mental map. Since the authors assume that the process graph is series-parallel, and since we do not necessarily have a series-parallel graph, this approach is not applicable to our work.

1.6 Overview

The remainder of this work is structured as follows. In Chapter 2, we provide the required preliminaries. After that, in Chapter 3, we provide an in-depth analysis of mental map research and define what we regard as the mental map. Additionally, we define the concepts of layout quality and layout stability more formally. Then, in Chapter 4, we provide a detailed description of the global ranking and global order. In Chapter 5, we describe a test framework which is used to extensively test and compare our novel graph layout algorithm. Then, in Chapter 6, we provide details on how we use graph layout animation. After that, in Chapter 7, we provide and discuss the results of a quantitative and qualitative evaluation. In Chapter 8, we discuss several aspects of our approach. Finally, in Chapter 9, we provide concluding remarks and list possible future work.

Chapter 2

Preliminaries

In this work, we bring process mining and graph drawing together. Therefore, in Section 2.1, we introduce the required process mining terminology and, in Section 2.2, we introduce the required graph drawing terminology. After that, in Section 2.3, we introduce the Sugiyama framework, which lies at the basis of our graph drawing approach.

2.1 Process mining

In the area of process mining, event data is obtained during the execution of the process and is stored in an event log. The event log contains cases consisting of sequences of events. These are defined as follows:

Definition 2.1.1 (Activity) *A recordable step in the process.*

Definition 2.1.2 (Event) *An event ε is an occurrence of a certain activity at a certain time, for some case. Event attributes are denoted by $\#_{activity}(\varepsilon)$, $\#_{timestamp}(\varepsilon)$ and $\#_{case}(\varepsilon)$, respectively.*

Definition 2.1.3 (Case) *Let $\mathbb{C}$ be the set of all cases in the event log. A case $c \in \mathbb{C}$ is a sequence of events, ordered ascending on $\#_{timestamp}(\varepsilon)$. Every case has a unique identifier $\#_{case}(c)$ and its ordered event sequence is denoted as $\#_{seq}(c) = \langle \varepsilon_1, \varepsilon_2, ..., \varepsilon_n \rangle$.*

Table 2.1 contains a fragment of the invoices event log. As we can see, we have cases containing ordered sequences of events which represent the occurrence of an activity at a certain time.

Definitions 2.1.2 and 2.1.3 are minimal in the sense that they define which attributes are at least required to define an event or case respectively. An event can potentially contain additional attributes. For example, the name of the person who performed the activity represented by the event attribute: $\#_{name}(\varepsilon)$. Similarly, a case can also contain additional case attributes.

In this work, we are interested in sets of cases that express the same behavior. In particular, sets of cases that followed the same sequence of events for which we ignore direct event repetitions. For a case c, we denote the sequence of events without direct event repetitions as $\#_{seq'}(c)$. For example, if $\#_{seq}(c) = \langle A, B, B, B, C, A \rangle$, then $\#_{seq'}(c) = \langle A, B, C, A \rangle$. In graph drawing, such

Case ID	Activity	Timestamp
IF-1500000	Receive invoice	28-08-2015 11:00:50
	Check received invoice	28-08-2015 11:00:51
	Checked and approved	28-08-2015 11:16:43
	Pay invoice	02-09-2015 12:52:06
IF-1500001	Receive invoice	09-09-2015 09:00:20
	Check received invoice	09-09-2015 09:00:22
	Final check of invoice	10-09-2015 10:04:24
	Approve invoice	10-09-2015 13:13:34
	Pay invoice	14-09-2015 15:29:08
...	...	...

Table 2.1: Fraction of the invoices event log.

direct event repetitions are drawn as self loops. Since such self loops do not affect the layout of a graph, we ignore them. We define a variation as follows:

Definition 2.1.4 (Variation) *A variation $v \subseteq \mathbb{C}$ is a set of cases which all contain the same ordered sequence of events for which we ignore direct event repetitions, denoted by $\#_{seq}(v)$. More specifically, all cases $c \in v$ have the same $\#_{seq'}(c)$ value.*

Let $\mathbb{V}$ be the set of all variations. Then, from definition 2.1.4, it follows that $\bigcup_{v \in \mathbb{V}} v = \mathbb{C}$.

2.2 Graph Drawing

In the area of graph drawing, there is a distinction between static and dynamic graph drawing. Static graph drawing considers the problem of computing a layout for a single graph. Dynamic graph drawing, on the other hand, considers the problem of computing layouts for a sequence of static graphs. Both of these concepts are explained in more detail in Section 2.2.1 and Section 2.2.2 respectively.

2.2.1 Static graph drawing

In static graph drawing, a single graph is provided and a layout must be computed for the given graph. In this work, we consider (static) graphs $G(V, E)$ as a pair consisting of a finite set V of nodes and a finite set E of directed weighted edges e. That is, an edge $(u, v) \in E$ is an ordered pair with $u, v \in V$ and has an associated positive integer weight $weight(e) \in \mathbb{N}^+$. Additionally, an edge (u, v) is a self loop when $u = v$ and a two loop is a pair of edges $e_1, e_2 \in E$ such that $e_1 = (u, v)$ and $e_2 = (v, u)$.

Drawing is done in two dimensions, where the x-axis goes from left to right, and the y-axis from top to bottom. Consequently, the goal of a graph layout algorithm is to compute a position $pos(v)$ for every node $v \in V$, which consists of an x and y coordinate, denoted by $x(v)$ and $y(v)$ respectively. Additionally, for every edge $e \in E$, a sequence of coordinates, representing the control points of a spline, are computed.

In this work, graphs are drawn in a hierarchical manner, that is, nodes are placed on so called ranks. Ranks are parallel layers, such that, for any two nodes u and v on the same rank, i.e.,

$rank(u) = rank(v)$, we have that $y(u) = y(v)$. We draw the hierarchical graphs vertically from top to bottom. That is, rank 0 has the lowest y-coordinate and subsequent ranks have increasingly higher y-coordinates. The rank assignment results in a direction for every edge (which is not a self loop). A forward edge is an edge $(u, v) \in E$ such that $rank(u) < rank(v)$. A backward edge is the inverse of a forward edge, i.e., $rank(u) > rank(v)$. Additionally, for edges that are not a self-loop, we do not allow them to be horizontal, i.e., $\forall_{(u,v)\in E,\, u\neq v}\, y(u) \neq y(v)$.

2.2.2 Dynamic graph drawing

A field, closely related to our work, is the field of dynamic graph drawing. A dynamic graph is defined as a sequence $S = (G_1, G_2, ..., G_n)$ of static graphs. Dynamic graph drawing algorithms often aim to compute a graph layout for every individual graph $G_i = (V_i, E_i)\, 1 \leq i \leq n$, such that the quality of individual layouts is balanced with layout stability over time. In the area of dynamic graph drawing, a distinction is often made between online and offline dynamic graph drawing [BBDW17]. In online dynamic graph drawing, graph layouts need to be computed for a sequence of graphs without knowing the full sequence from the beginning. An online dynamic graph layout algorithm tries to compute a graph layout for graph G_{n+1} such that the mental map is preserved. In contrast, in offline dynamic graph drawing, the whole sequence S of graphs G_i is known beforehand.

In this work, we essentially consider a combination of online and offline dynamic graph drawing. Given the event log, we can compute a (super) graph $\overline{G} = (\overline{V}, \overline{E})$ where $\overline{V}$ and $\overline{E}$ are the sets of all nodes and edges we can potentially encounter respectively. For any graph $G_i \subseteq \overline{G}$, $G_i = (V_i, E_i)$, we have $V_i \subseteq \overline{V}$ and $E_i \subseteq \overline{E}$. Hence, our setting is offline in the sense that we know which graphs we can potentially encounter. However, our setting is also online because we do not know the exact sequence of graphs beforehand.

2.3 Sugiyama framework

In this work, the graph layout algorithm we present is based on the Sugiyama framework [STT81] which provides a step-wise approach to hierarchical, directed graph drawing. The framework only specifies what should happen in each step while the implementation can be done in many different ways. Additionally, since each of these steps is executed in sequence, the results of each step influence later steps. The Sugiyama framework defines the steps listed below. Figure 2.1 provides an illustration of every step.

1. **Cycle Removal:** In the first step, cycles are removed by reversing edges and removing self loops. Consequently, a Directed Acyclic Graph (DAG) is obtained.
2. **Rank Assignment:** Because we are drawing graphs in a hierarchical manner, every node is positioned on a rank.
3. **Node Ordering:** After assigning nodes to ranks, edges that cross multiple ranks are split up such that no edge crosses a rank. This is done by replacing these edges by a chain of virtual nodes and edges. This concept is illustrated in Figure 2.1 step 3a. Next, the order in which nodes (both real and virtual) are placed on the ranks is determined. This order determines the number of edge crossings in the graph and should therefore be optimized.
4. **Node Positioning:** After the node order for each rank is known, actual positions can be computed for each node.
5. **Spline Drawing:** After positioning each node, in the final step, the edges are drawn.

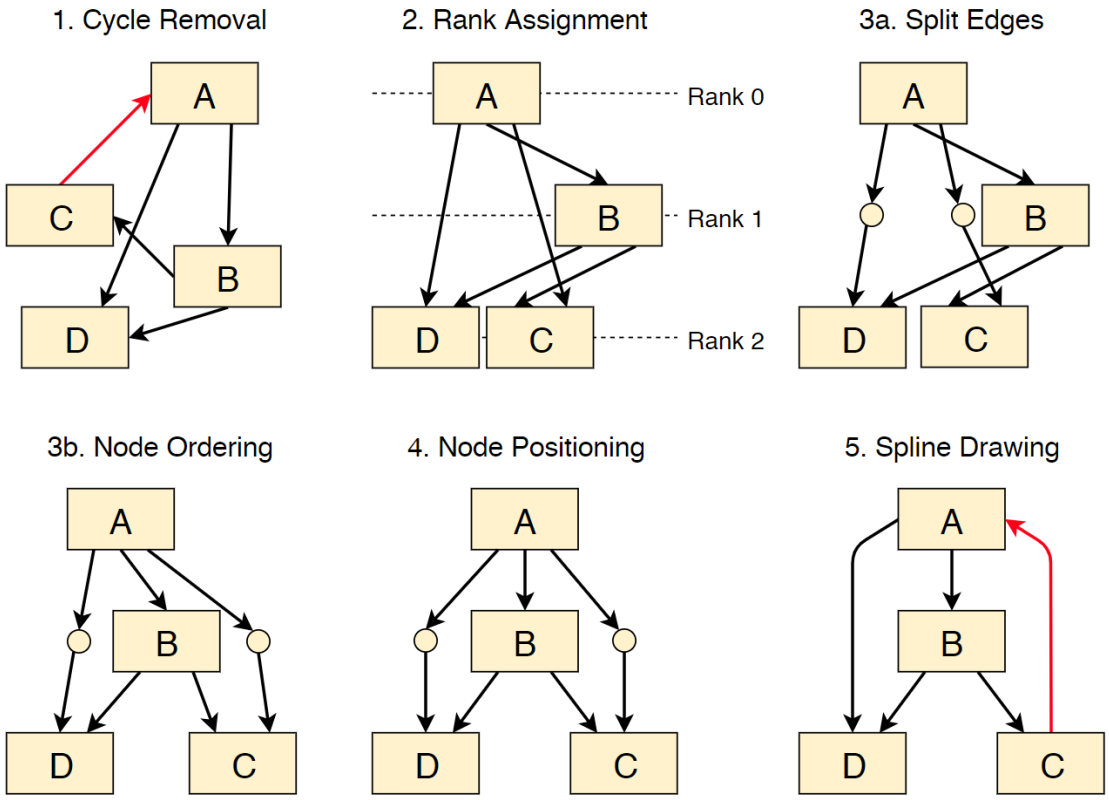

Figure 2.1: Illustration of the Sugiyama framework steps. The edge highlighted in red in step 1 is the edge which is reversed in order to remove cycles. Note that this same edge is drawn in its original direction in step 5. Circles represent virtual nodes.

2.3.1 Dot

A popular implementation of the Sugiyama framework is *dot* [GKN15], which is the successor of DAG [GNV88]. *Dot* is part of the open source, free to use, Graphviz [1] software package. Graphviz supports many in- and output formats and can therefore easily be integrated into existing systems. Consequently, many industrial tools integrate *dot* in order to draw hierarchical directed graphs that represent a process. Because of this, we evaluate our graph layouts against those computed by *dot*.

Dot is based on the work of Gansner et al. [GKNV93] and uses the following algorithms for each of the Sugiyama framework steps:

1. **Cycle Removal:** By doing a Depth First Search (DFS) traversal on the graph, all edges which create a cycle (back-edges) are found. These edges are reversed to create a DAG. Additionally, self-loops are removed.
2. **Rank Assignment:** The rank assignment is computed using the following cost function:

$$\sum_{e \in E} weight(e) \cdot length(e)$$

Where E is the set of edges in the graph, $weight(e)$ is the weight of an edge e, and $length(e)$ is the rank difference between the endpoints of e. By minimizing the cost function using

an integer linear program solver, an optimal rank assignment can be computed. For this purpose, *Dot* uses the Network Simplex algorithm of Gansner et al. [GKNV93].

3. **Node Ordering:** To minimize the number of edge crossings, *dot* repeatedly applies a rank-by-rank sweep of a one-rank crossing minimization algorithm until no further improvement is possible. That is, for some rank r, the nodes at the rank above and below are kept fixed. The node order at r is then permuted to decrease the edge crossings.
4. **Node Positioning:** Determining the y-coordinate of a node directly follows from the rank a node is placed on. Determining the x-coordinate of a node is done by first computing a so called auxiliary graph. The structure of the auxiliary graph is built such that it encodes two elements. The first is the constraint that the order within ranks should be preserved. The second element encodes, for every adjacent node pair, how 'important' it is to vertically align these nodes. Network Simplex is then run on this auxiliary graph. The values returned by Network Simplex are used as the x-coordinate.
5. **Spline Drawing:** *Dot* uses a spline drawing algorithm introduced by Gansner et al. [GKNV93]. This algorithm tries to find the smoothest curve between two points such that other nodes, edges, and labels are avoided. The algorithm draws every spline in two phases. First, a polygonal region is computed in which the spline may be drawn. After that, given this polygonal region, a spline within this region is computed. Since the first phase takes already drawn splines, nodes and labels into account, no accidental edge-edge, edge-node or edge-label crossings can occur.

Chapter 3

Quality vs. Stability

In this work, we consider the problem of mental map preservation. Therefore, in Section 3.1, we discuss research done into mental map preservation and we give a definition of what we consider to be the mental map. One approach we use to preserve the mental map, is to minimize change in the graph layouts, i.e., we create graph stability. However, only creating graph stability is not enough, we also want to have graph layouts of high quality. Unfortunately, these two concepts contradict each other. If we minimize change in order to obtain graph stability, we cannot always have graph layouts of high quality. Therefore, it is important to get a trade off between graph layout stability and quality. To quantify the concepts of graph layout quality and stability, we define several metrics that measure graph layout quality and stability in Section 3.2. After that, in Section 3.3, we discuss the concept of graph layout understandability.

3.1 Mental Map

A lot of research into the effect of mental map preservation has been done. In general, the type of graph and tasks considered play a major role in whether mental map preservation benefits the user.

3.1.1 Undirected graphs

Most of the mental map research considers undirected graphs for which the layout is computed using a force-based algorithm. In particular, a large number of experiments [PS08, APP11, AP12a, AP12b] used Graphael [EHK+03]. Graphael implements a parameter which essentially allows the user to specify how well the mental map should be preserved. This makes Graphael ideal for mental map preservation experiments. Purchase et al. [PS08] examined the effect of mental map preservation for degree reading tasks. The experiment was done for high, medium, and low mental map preservation. Their results indicated that users performed the best with high or low mental map preservation. Performance was the worst under medium mental map preservation. The authors state that the good performance under high or low mental map preservation indicates that user preference also plays a role. Archambault et al. [APP11] tested the difference between no and high mental map preservation for both animation and small multiples presentation methods. Degree reading, graph appearance, graph growth, and path reading tasks were used in their experiment. For both animation and small multiples, no significant performance difference between

no and high mental map preservation was found. Archambault et al. [AP12a] considered mental map preservation in the context of memorability tasks. Their experiment consisted of two phases. In the first, the memorization phase, participants were requested to memorize different graph sequences. In the second, the recall phase, participants were shown sequences of graphs and had to indicate whether they were the same as the sequences presented in the memorization phase. The quantitative results of their experiment indicated that mental map preservation has no significant effect on response time or error rate. The qualitative results, however, suggested that users find memorability tasks easier when the mental map is preserved. Archambault et al. [AP12b] tested no versus high mental map preservation for revisitation and (long) path finding tasks. They found that for both tasks, mental map preservation significantly improves the response time and significantly reduces the number of errors. They argue that the stable drawings support the participants in orienting themselves in the data, making it easier to relocate certain areas of the drawing or to follow long paths. In contrast to the works mentioned above, Saffrey et al. [SP08] use their own dynamic force-based graph drawing algorithms. Their algorithms preserve the mental map by restricting node movement in different ways. In their experiment, participants had to do degree reading and path finding tasks. Their results contain no evidence in favor of preserving the mental map.

3.1.2 Directed graphs

In the context of dynamic hierarchically drawn directed acyclic graphs, Purchase et al. [PHG06] used the algorithm of Görg et al. [GBPD04] to test no versus high mental map preservation. They found that, when nodes need to be identifiable by name, mental map preservation was important. Tasks that focus on edges, or tasks that do not require nodes to be uniquely identifiable, however, did not benefit from mental map preservation. Zaman et al. [ZKS11] also investigated mental map preservation in the context of dynamic hierarchically drawn directed acyclic graphs. They did this for their DARLS system, which allows the user to compare two hierarchically drawn graphs. A low mental map preservation version of their algorithm was compared against a high mental map preservation version. For the task of finding the appearance of new nodes in the graph, no significant effect was found for response time or error rate.

3.1.3 Does mental map preservation help?

Intuitively, it seems reasonable to assume that mental map preservation has a positive effect on user performance. In general, however, research appears to suggest otherwise. We do note, however, that most of this research [PS08, SP08, APP11, AP12a, AP12b] is not really applicable to our work because it was done in the context of undirected graphs. The research [PHG06, ZKS11] in the context of dynamic hierarchically drawn directed acyclic graphs, which is more related to our work, does find some evidence in favor of mental map preservation. Finally, for tasks that involve revisitation [AP12b], path finding [AP12b], or nodes identifiable by name [PHG06], there is evidence in favor of preserving the mental map. These results are promising, because these tasks closely relate to our work (see Section 7.2.1).

3.1.4 Mental map definition

Another important aspect is that most of the aforementioned work [PS08, SP08, APP11, AP12a, AP12b] uses the most commonly used definition of the mental map [CP96]:

"The placement of existing nodes and edges should change as little as possible when a change is

made to the graph."

Note that this definition considers the change in a graph layout, and therefore, actually defines mental map preservation instead of the mental map itself. In the graph drawing literature, however, these two concepts are considered to be similar. Therefore, a mental map definition (often) defines the change that affects the mental map (preservation).

For the research [PHG06, ZKS11] done in the context of dynamic hierarchically drawn directed acyclic graphs, it not really clear which definition of the mental map is used. However, since the approach towards mental map preservation, in these works, focuses more on preserving the relative order of nodes, we believe that the mental map definition of Misue et al. [MELS95] is more appropriate here. Misue et al. [MELS95] define three mathematical models that represent the mental map:

- **Orthogonal ordering model:** a layout adjustment should preserve the direction of node v to node u for each pair of nodes u and v.
- **Proximity model:** nodes which are close together, should remain close together.
- **Topology model:** graphical objects in a region should stay in that region.

Essentially, we believe that the mental map definition of Coleman and Parker [CP96], which states that change in node/edge placement should be minimized, is too narrow. This is also one of the main conclusions of the work of Saffrey et al. [SP08]:

"A notion of the mental map as a restriction on node movement is too narrow. The mental map implementation should consider the overall relation between nodes and edges, rather than merely position.".

Especially in our context, the relationships between nodes and edges form semantic implications on the underlying data. For example, if a node is placed above another node, this implies that the activity represented by the upper node occurs before the activity represented by the lower node. Hence, we believe that the mathematical models of the mental map, as defined by Misue et al. [MELS95], provide a better mental map definition.

3.2 Metrics

In order to preserve the mental map, we want to create stable layouts that are still of high quality. To define the concepts of graph layout quality and stability more formally, we introduce quality and stability metrics. The quality and stability metric definitions (see Definitions 3.2.1 and 3.2.2, respectively) are similar to those of Diehl et al. [DG02]. Sections 3.2.1 and 3.2.2 describe the quality and stability metrics in more detail.

Definition 3.2.1 (Quality Metric) *Given a graph $G = (V, E)$ for which a layout has been computed. A quality metric is a function $QM_\lambda : G \to R_0^+$ that quantifies a certain graph layout aesthetic. λ is a unique name representing the metric. A higher value for a quality metric implies that G adheres less to the aesthetic criterion. For example, for a graph layout G that has no edge crossings, the number of edge crossings is denoted by $QM_{crossings}(G) = 0$.*

Definition 3.2.2 (Stability Metric) *Given two graphs $G = (V, E)$ and $G' = (V', E')$ for which layouts have been computed. A stability metric is a function $SM_\lambda : (G, G') \to R_0^+$ that quantifies,*

depending on what we measure, the amount of change between G and G'. λ is a unique name representing the metric. When $SM_\lambda = 0$, there is no difference between G and G', for metric λ. For example, the relative movement of nodes between the two graphs is denoted by $SM_{rel\_eucl}(G, G')$.

3.2.1 Quality Metrics

Graph layout quality is expressed in terms of aesthetic criteria [Pau93, Pur02, AEHK10, HEHL13]. Some commonly used aesthetic criteria include: minimize edge crossings, minimize edge bends, maximize symmetry, minimize the area, maximize consistent flow direction. Layout algorithms are often designed with such aesthetic criteria in mind, i.e., the layout is computed such that certain aesthetic criteria are optimized. Depending on the domain [PCA02] and type of graph, some aesthetic criteria are more relevant than others. For example, *dot*, which is based on the work of Gansner et al. [GKNV93], was designed with the following aesthetic criteria in mind [GKNV93]:

- ***A1:*** *Expose hierarchical structure in the graph. In particular, aim edges in the same general direction if possible. This aids finding directed paths and highlights source and sink nodes.*
- ***A2:*** *Avoid visual anomalies that do not convey information about the underlying graph. For example, avoid edge crossings and sharp bends.*
- ***A3:*** *Keep edges short. This makes it easier to find related nodes and contributes to A2.*
- ***A4:*** *Favor symmetry and balance. This aesthetic has a secondary role in a few places in our algorithm.*

Optimizing all of these aesthetic criteria at the same time is generally not possible, because some criteria can contradict each other. However, research [HEHL13] suggests that optimizing several aesthetic criteria at the same time provides better results than optimizing for only a single criterion. Nevertheless, some aesthetic criteria affect the readability of a graph layout more significantly than others. Purchase et al. [PCJ97, Pur00] find that the number of edge crossings affects graph readability more than the number of bends or the amount of symmetry. Additionally, Huang et al. [HEHL13] find that the angle of a crossing also affects readability.

Purchase et al. [PCA02] state that the semantic domain of the graph drawing affects which aesthetic criteria need to be taken into account. Since we are computing layouts for process graphs, we want to focus on aesthetics that relate to the process. In related fields (business process visualization [RBRB06, ESK09, AEHK10, KKRM11, GPZ+14, BS15], flowchart visualization [ST01], workflow visualization [DDK+02, YLS+04]), commonly consider aesthetics include: minimize edge crossings, minimize edge bends, minimize edge length, maximize consistent flow direction, minimize area, prevent node overlap. The aesthetic of preventing node overlap is not relevant because this can never occur in our setting. We deem all other aesthetics as relevant, however. Moreover, in a process, there is often some main structure which is relevant to understand the whole process [RBRB06, AEHK10]. Intuitively, this can be thought of as the 'skeleton' of the process graph. Since, in a process graph, the weight of an edge (u, v) represents how often activity u occurs before v, intuitively, the edges with highest weight represent the most prominent process behaviour. Consequently, we deem it especially important to optimize the aesthetic criteria for the edges with highest weight. Therefore, the quality metrics, as described in the sections below, (often) take into account edge weights.

Some of the quality metrics, as described in the sections below, consider the edges. Since the edges are drawn as splines, computing exact values (such as intersections or edge length) for the edges is quite expensive. Therefore, consider that an edge e is essentially drawn as a sequence of spline segments, which we denote as $segments(e)$. Every segment $s \in segments(e)$ contains

four points (coordinates): a start point p_s at which the spline segment starts, two control points p_{c1} and p_{c2} that define the shape of the spline segment, and an end point p_e at which the spline segment ends. In this work, we measure the length, shape, and number of crossings of edges. Since computing this exactly is expensive, we use an approximation of the edge e, which consists of approximations of the spline segments $s \in segments(e)$. An approximation of a spline segment is the straight line $line(s)$ between p_s and p_e. Moreover, the euclidean distance between p_s and p_e is an approximation of the actual length of s and is denoted as $length(s)$. Consequently, the approximate length of an edge e is defined as:

$$length(e) = \sum_{s \in segments(e)} length(s)$$

Running Time
The running time of the layout algorithm is not really an aesthetic criterion since it does not relate to the computed graph layout. Nevertheless, a faster algorithm is obviously more desirable to the user since this reduces waiting times. Therefore, the quality metric $QM_{time}(G)$ measures the time (in milliseconds) the layout algorithm required to compute the graph layout for G.

Edge Crossings
The number of edge crossings in a graph layout has a significant effect on the readability [PCJ97, Pur00]. Therefore, we define the following function that computes the 'cost' of the edge crossing(s) between two edges.

$$cross(e_1, e_2) = \sum_{(s_1,s_2) \in segments(e_1) \times segments(e_2)} intersect(e_1, e_2, s_1, s_2)$$

Where $intersect(e_1, e_2, s_1, s_2)$ is defind as follows:

$$intersect(e_1, e_2, s_1, s_2) = \begin{cases} 0 & line(s_1) \text{ and } line(s_2) \text{ do not intersect} \\ weight(e_1) \cdot weight(e_2) & line(s_1) \text{ and } line(s_2) \text{ intersect} \end{cases}$$

Since edges of higher weight are drawn more salient, edge crossings of high weight edges are also more salient. For example, in Figure 3.1, the crossings in red rectangle 1 are more salient than the edges crossing in red rectangle 2. Therefore, to quantify that we consider high weight edge crossings as worse, we multiply the weights of e_1 and e_2. Let $pairs(E)$ be the set of unique edge pairs in a graph G. The edge crossings quality metric is defined as follows:

$$QM_{crossings}(G) = \sum_{(e_1,e_2) \in pairs(E)} cross(e_1, e_2)$$

Average Edge Length
Short edges are easier to follow [GKNV93]. Moreover, we deem it important to have short high weight edges. Therefore, the average edge length quality metric, as defined below, takes the weight of an edge into account.

$$QM_{avg\_length}(G) = \frac{1}{|E|} \sum_{e \in E} length(e) \cdot weight(e)$$

Edge Bends
Edge bends negatively affect graph readability [PCJ97, Pur00]. Our approach towards computing

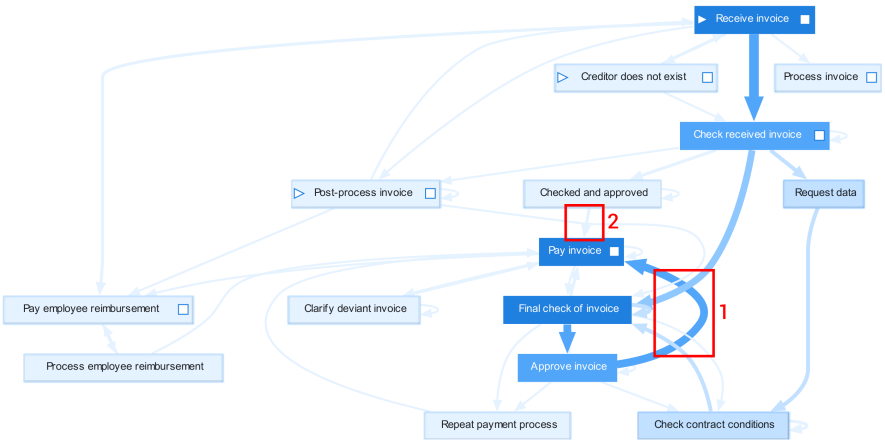

Figure 3.1: A layout computed by *dot* for the invoices data set. Clearly, the edge crossings in red rectangle 1 are more salient than those in red rectangle 2.

the number of edge bends is inspired by the work of Bridgeman et al. [BT98]. For an edge segment s, we define direction $dir(s) \in \{N, E, S, W\}$ (North, East, South, West) as the most prominent direction of $line(s)$. For example, in Figure 3.2 the direction of every edge segment is illustrated. As we can see, the shape of an edge can be described as a sequence of directions. The concatenation of these directions results in a shape string [BT98] $string(e)$. For example, in Figure 3.2, the shape string of the rightmost edge equals $string(e) = \text{ESSW}$. Consequently, the number of bends $bends(e)$ in an edge equals the number of times there is a change in segment direction. For example, in Figure 3.2, the bends are indicated by black circles. The edge bends quality metric is defined as follows:

$$QM_{bends}(G) = \sum_{e \in E} bends(e) \cdot weight(e)$$

Since we prefer high weight edges to be straight, we multiply by the edge weight.

Back Edges

Since we are drawing graphs vertically in a top-down manner, we want to have as many forward edges as possible. The amount of forward edges directly relates to the consistency of 'flow direction' in the graph layout. Therefore, let $backedges(G)$ be the set of back edges in graph layout G. Then, the back edge quality metric is defined as follows:

$$QM_{back\_edges}(G) = \sum_{e \in backedges(G)} weight(e)$$

Since we want high weight edges to be forward edges, we sum up the weights of the back edges instead of simply counting the number of back edges.

Flow Direction

The back edge quality metric relates to the consistency of the flow direction in a graph layout. However, since this metric does not take the actual shape of an edge into account, it does not properly describe the flow in terms of what the user sees on the screen. For example, a forward edge e, which, intuitively, is drawn downward, can have, for all segments $s \in segments(e)$, a direction $dir(s) = \text{E}$. Therefore, we introduce a quality metric, similar to the 'direction of a model' metric of Bernstein et al. [BS15], that measures the flow direction of the edges in a graph layout G. Let $north(G)$, $east(G)$, $south(G)$, $west(G)$, be the number of edge segments s in G that have $dir(s)$ value N, E, S, W respectively. Intuitively, since we are drawing graphs vertically

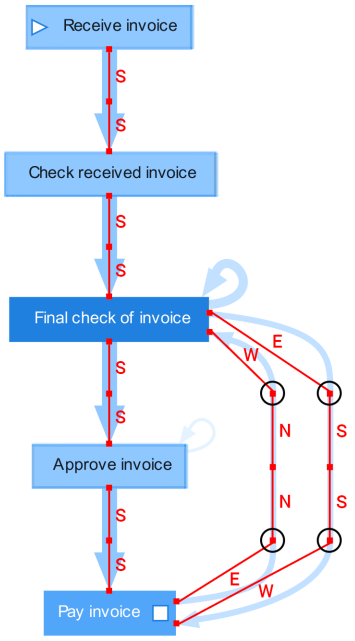

Figure 3.2: Illustration of the directions of every edge segment. Red squares indicate the start and end points of edge segments. Red lines illustrate the line $line(s)$ of an edge segment s. The direction $dir(s)$ value of every segment is indicated next to the respective red line. Moreover, the red squares encircled in black indicate edge bends. In total, there are 4 edge bends in this graph layout.

in a top-down manner, all $north(G)$ and $south(G)$ edge segments adhere to the vertical drawing direction while $east(G)$ and $west(G)$ edge segments 'break the flow'. Therefore, the flow direction quality metric is defined as follows:

$$QM_{flow}(G) = \frac{east(G) + west(G)}{north(G) + south(G)}$$

Note that the lower $QM_{flow}(G)$, the better the flow direction adheres to the (vertical) drawing direction of the graph layout. Also, we do not distinguish between $north(G)$ and $south(G)$ because this is already captured by the $QM_{back_edges}(G)$ quality metric.

Area
The quality metric $QM_{area}(G)$ computes the area required to draw the graph layout (including edge labels). Preferably, in order to fit a graph layout on the screen of the user, the area of a graph layout is a small as possible.

Discarded Metrics
Some quality aesthetics, as discussed below, found in the literature, are not used for different reasons.

Some layout algorithms aim to make a graph layout as symmetrical as possible. Since a process does not necessarily contain symmetries, we do not include this aesthetic.

Other layout algorithms, in order to fit the resulting layout on the screen, aim to compute a graph layout that has an aspect ratio of 1:1. Since this aesthetic can contradict quite some other aesthetic criteria (flow direction, back edges, edge bends), we do not consider this aesthetic.

Other alternative aesthetics related to the length of edges exist. We could, for example, compute the minimum/maximum edge length in a graph layout. However, since the average edge length relates to all edges, and not just to the shortest/longest, we decided to use the average of edge lengths.

3.2.2 Stability Metrics

Measuring graph stability essentially comes down to measuring the degree of change between two graphs. When the degree of change between graphs is small, there is a higher degree of stability. In the field of graph drawing, the problem of how to measure change (or similarity) has received quite some attention. Bridgeman et al. [BT98] introduce several metrics that measure the difference between two layouts of the same graph. Since we want to compute the difference between layouts of different graphs, these metrics are not directly usable. However, in a later work, Bridgeman et al. [BT00] extend their previous work by introducing additional metrics and refining some existing ones. Diehl et al. [DG02] provide a formal definition of a difference metric and consequently propose two difference metrics. In the field of layout adjustments, Lyons et al. [LMR98] introduce several similarity metrics. Since layout adjustments are considered, these metrics only work for different layouts of the same graph.

Considering the existing work [BT98, LMR98, BT00, DG02], there exist a significant number of difference metrics. However, based on our definition of the mental map [MELS95], some of these metrics are more applicable than others. Additionally, we found that some metrics essentially measure the same concept, but in a different way. Therefore, by considering, for every difference metric, how it relates to our mental map definition, we selected the difference metrics as described in the following sections. We denote the euclidean distance between two points (coordinates) by $dist(p_1, p_2)$. For a stability metric $SM_\lambda(G, G')$, the equivalent of a node $v_i \in G$ in G' is denoted by v_i'. For example, the distance between the old and new position of a node v equals $dist(pos(v), pos(v'))$. Additionally, difference metrics only measure based on nodes or edges that are in both layouts, i.e. the sets $V \cap V'$ and $E \cap E'$. Moreover, let $pairs(V' \cap V')$ and $pairs(E \cap E')$ be the sets of unique node/edge pairs in $V \cap V'$ and $E \cap E'$ respectively.

Relative Euclidean
The relative euclidean stability metric measures, for all node pairs present in both layouts, the change in relative distance. This metric is defined as follows:

$$SM_{rel\_eucl}(G, G') = \sum_{(v_i, v_j) \in pairs(V \cap V')} | \, dist(pos(v_i), pos(v_j)) - dist(pos(v_i'), pos(v_j')) \, |$$

By taking the absolute value, we make sure the metric is always positive.

Hausdorff
The hausdorff distance is a standard metric used to measure the match between two point sets [BT98]. A disadvantage, however, is the fact that the standard hausdorff metric does not take point labels into account. More specifically, every point in the point set is seen as a coordinate and not as a unique entity. Consequently, swapping the position of two points in the point set does not change the point set. Since we do have 'labeled' points (nodes), this standard metric does not work for us. Therefore, we consider an adapted version of the hausdorff distance metric [BT00] (that does take point labels into account) which is defined as follows:

$$SM_{hausdorff}(G, G') = \max_{v \in V \cap V'} dist(pos(v), pos(v'))$$

Orthogonal
The orthogonal ordering mental map model [MELS95] states that a layout adjustment should

preserve the direction of node v to node u for each pair of nodes u and v. In order to quantify this, we define the following stability metric:

$$SM_{orthogonal}(G, G') = \sum_{(v_i,v_j)\in pairs(V\cap V')} angle(pos(v_j) - pos(v_i), pos(v'_j) - pos(v'_i))$$

The *angle* function computes the smallest positive angle between the two provided vectors. As we can see, we provide the vectors for the old and new layout between the nodes in the node pair. Therefore, we essentially use the change in angle to express the change in direction between two nodes.

Epsilon-Cluster
The proximity mental map model [MELS95] states that nodes which are close together should remain close together after a change. An ϵ-cluster for a node v_i is the set of nodes v_j such that $dist(pos(v_i), pos(v_j)) \leq \epsilon$. In this work, ϵ is defined as the largest nearest neighbor distance, i.e. [ELMS91]:

$$\epsilon = \max_{v_i} \min_{v_j \neq v_i} dist(pos(v_i), pos(v_j))$$

The ϵ-cluster stability metric measures how the ϵ-cluster of node v_i compares to that of node v'_i. Note that ϵ can be different for G and G', denoted by ϵ and ϵ' respectively. Consequently, we define the following sets:

$$\epsilon(G) = \{(v_i, v_j) \in pairs(V \cap V') \,|\, dist(pos(v_i), pos(v_j)) \leq \epsilon\}$$
$$\epsilon(G') = \{(v_i, v_j) \in pairs(V \cap V') \,|\, dist(pos(v'_i), pos(v'_j)) \leq \epsilon'\}$$

The ϵ-cluster stability metric is then defined as:

$$SM_{cluster}(G, G') = 1 - \frac{|\, \epsilon(G) \cap \epsilon(G') \,|}{|\, \epsilon(G) \cup \epsilon(G') \,|}$$

Observe that when the ϵ-clusters of nodes do not change, this metric returns 0.

Edge Shape
While most stability metrics measure change in terms of the positions of nodes, Bridgeman et al. [BT00] propose a metric that measures the change in the shape of the edges. This metric is defined as follows:

$$SM_{edge\_shape}(G, G') = \sum_{e\in E\cap E'} edit(string(e), string(e'))$$

The *edit* function computes the levenshtein distance [Lev66] between the two edge shape strings. Intuitively, a large edit distance implies that the shape of an edge changed significantly.

Discarded Metrics
Some stability metrics found in the literature, as discussed below, are not used for different reasons.

The absolute euclidean distance metric is an intuitive and commonly used metric [BT98, LMR98, BT00, DG02]. However, because it is sensitive to global layout translations (which are visually not observable), we exclude this metric. For the same reason, the nearest neighbor between metric, as introduced by Bridgeman et al. [BT98], is also not considered.

An alternative orthogonal ordering metric is the one proposed by Diehl et al. [DG02]. In this metric, the amount of node pairs that change their relative vertical or horizontal direction is measured. Therefore, this metric is essentially a discrete variant of $SM_{orthogonal}(G, G')$. Consequently, it is less accurate.

Bridgeman et al. [BT98] introduce the nearest neighbor within metric to express the change in node proximity. We implemented this metric and found it to be too sensitive. Due to rounding issues, the coordinate of a node could change by 1 unit. Consequently, nearest neighbor information could change while no change was visually observable.

3.3 Layout Understandability

The metrics introduced in Section 3.2.1 capture the quality of (a) graph layout(s). These quality metrics, however, relate to the quality of the graph layout in terms of its readability. Another important aspect of a graph layout, is its understandability. Since we are visualizing processes, we want to compute graph layouts that allow the user to easily understand the underlying process. Therefore, in this section, we consider the understandability of graph layouts.

While the readability of a graph layout can be 'measured' using the quality metrics, the understandability of a graph layout is a more intuitive concept, i.e., it is more difficult to capture in a metric. In the sections below, we describe desirable visual properties of the graph layouts to be computed. By satisfying these visual properties, we aim to make the graph layouts as understandable (and readable) as possible. As we show in Section 3.3.1, the visual property of process semantics closely relates to the $QM_{back\_edges}(G)$ quality metric. Therefore, this quality metric is given higher priority in the design of our algorithms. The visual property of centrality & node alignment, as described in Section 3.3.2, is harder to express using quality metrics. Therefore, this visual property is explicitly considered in our algorithm design.

3.3.1 Process Semantics

The layout of a graph affects how the semantics of the underlying process are interpreted. For example, if some node A is positioned above another node B, then this implies that activity A occurs before activity B. Therefore, we want to compute graph layouts that properly represent the actual process, i.e., if, according to the event log, activity A occurs frequently before activity B, then A should be placed above activity B. This visual property closely relates to quality metric $QM_{back\_edges}(G)$. In order to see why, observe that the weight of an edge (u, v) represents how often activity u occurred before activity v. Consequently, properly showing the process semantics essentially comes down to maximizing the number of high weight forward edges.

In Figure 3.3, two layouts for the same graph are shown. The layout on the left is computed by *dot*, while the layout on the right is computed by our layout algorithm. To illustrate the process semantics visual property, consider the back edges in Figure 3.3. The layout on the left has 3 back edges, while the layout on the right only has 2 back edges. Moreover, the back edges in the layout on the left have a much higher weight. In particular, the path highlighted in green occurs between *Check received invoice* and *Final check of invoice*, which is nicely illustrated in the layout on the right. In the layout on the left, however, *Check contract conditions* is positioned below *Final check of invoice*, which may incorrectly suggest that *Check contract conditions* occurs after *Final check of invoice*. Hence, as we can see, by optimizing quality metric $QM_{back\_edges}(G)$, we obtain a better representation of the semantics of the process.

3.3.2 Centrality & Node Alignment

In a process, there is (often) some main structure or path which is relevant to understand the whole process [RBRB06, AEHK10]. For example, in Figure 3.3, this main path is the path highlighted

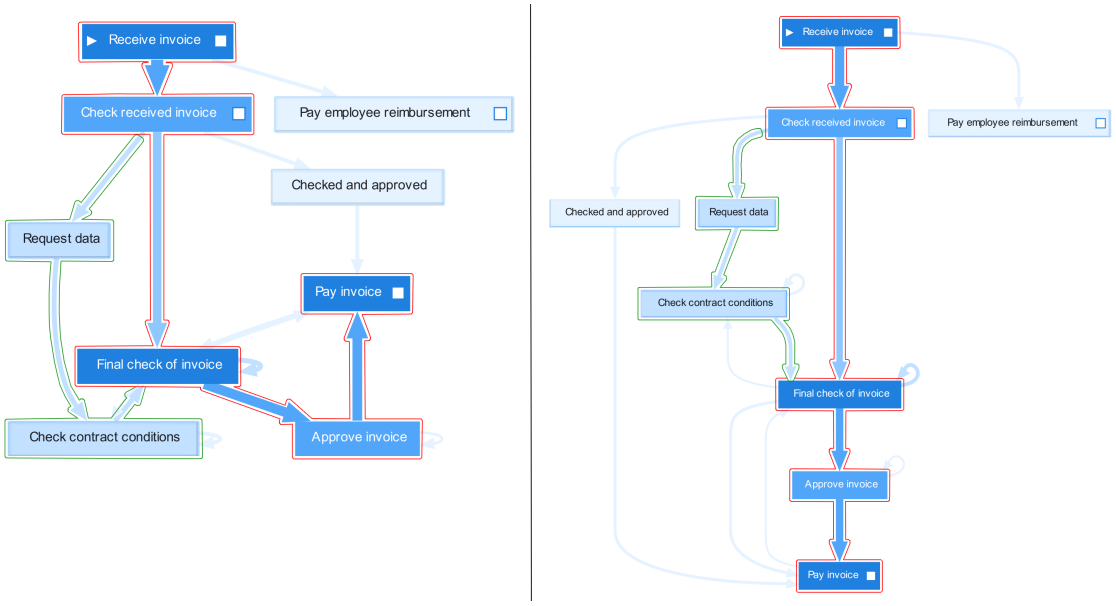

Figure 3.3: Illustration of two layouts computed for the same invoices graph. The layout on the left is computed by *dot*, while the layout on the right is computed by our layout algorithm.

in red. Since this path represents the most frequent behavior, we want to place this main path in the center of the graph layout and align the activities (nodes) on this path. By doing so, the main path is more prominently visible to the user. Consequently, the user can identify the main process structure more easily, which aids the user in understanding the whole process. As we can see in Figure 3.3 on the right, the centrality and alignment of the main path makes the graph much more understandable (and readable).

In terms of the Sugiyama framework, the centrality of nodes is essentially determined in the node order step. Therefore, our global order algorithm (as described in Section 4.2.3) explicitly addresses the centrality visual property by trying to centralize the main path. The alignment of nodes is done in the position step of the Sugiyama framework. Under the assumption that the main path has been centralized in the node order step, we (mostly) get the node alignment for free. Intuitively, the main path contains the edges of highest weight (see, for example, Figure 3.3). Since the position step implementation already tries to align nodes that are connected by high weight edges, the nodes on the main path are aligned.

Chapter 4

Approach

According to the orthogonal ordering model in our mental map definition [MELS95], a layout adjustment should preserve the direction of node v to node u for each pair of nodes u and v. In terms of the Sugiyama framework [STT81], this directly relates to the rank assignment and node ordering steps. The rank assignment step determines the vertical order of nodes while the node ordering step determines the horizontal order of nodes. Therefore, in order to preserve the mental map, we keep, for every graph layout we compute, the vertical and horizontal node order the same. We use a global ranking and global order to do this. These concepts are explained in more detail in Section 4.1 and Section 4.2, respectively. In Section 4.3 we discuss how the global ranking and global order fit into the Sugiyama framework.

4.1 Global Ranking

In order to ensure graph stability, we keep the vertical order of nodes the same for every graph layout we compute. More specifically, if a node $u \in V_i$ is on a lower rank than some other node $v \in V_i$ in some graph $G_i \subseteq \overline{G}$, then u should always be on a lower rank than node v for any graph $G_j \subseteq \overline{G}$. Therefore, as defined in Definition 4.1.1, a global ranking is essentially an ordered list of sets of nodes.

Definition 4.1.1 (Global Ranking) *The global ranking $GR = (\psi_0, ..., \psi_n)$ is an ordered list of sets ψ_i of nodes, where $i \in \mathbb{N}^0$. Intuitively, a set $\psi_i \in GR$ represents a global rank and therefore has an associated $rank(\psi_i)$ value representing its order i in GR. We denote the set of nodes $v \in \overline{V}$ on global rank ψ_i as $nodes(\psi_i)$. Moreover, we have the requirements that all node sets should be non-overlapping and that the union of node sets should equal $\overline{V}$. More specifically, $\forall_{\psi_i, \psi_j \in GR}$ such that $i \neq j$, we must have $nodes(\psi_i) \cap nodes(\psi_j) = \emptyset$ and $\bigcup_{\psi_i \in GR} nodes(\psi_i) = \overline{V}$. Consequently, every node $v \in \overline{V}$ is present on exactly one global rank, which we denote as $gr(v)$.*

So, essentially, the global ranking defines a ranking for $\overline{G}$. Additionally, recall that we disallow horizontal edges, and therefore, GR should be computed such that no edge (which is not a self-loop) $(u, v) \in \overline{E}$ is horizontal, i.e., $\forall_{(u,v) \in \overline{E},\, u \neq v}\, gr(u) \neq gr(v)$. When GR satisfies this requirement, we can directly use GR to obtain a valid ranking for any given graph $G_i \subseteq \overline{G}$. This is simply done by assigning every node $v \in V_i$ to rank $gr(v)$. Then, since not necessarily every node is present, we remove empty ranks to make sure the graph is as compact as possible. The global ranking

Figure 4.1: The global ranking computed for the invoices data set. The left column contains, from top to bottom, the ordered global ranks $\psi_i \in GR$. The nodes on each row represent the nodes present in $nodes(\psi_i)$.

computed for the invoices data set is shown in Figure 4.1. As we can see, all nodes are part of exactly one global rank ψ_i.

When using a global ranking, nodes cannot swap in the vertical direction. Consequently, the stability metrics: $SM_{rel\_eucl}(G, G')$, $SM_{hausdorff}(G, G')$ and $SM_{orthogonal}(G, G')$ are positively affected. Also, nodes that are close to each other in one layout are more likely to also be close to each other in another. Hence, the stability metric $SM_{cluster}(G, G')$ is also positively affected.

Computing a global ranking that adheres to the above-mentioned requirements is not too difficult. In fact, using any such global ranking will already ensure graph layout stability. However, we also want to have graph layouts of high quality, i.e., they should be readable and understandable. Therefore, for the graph layout readability, consider the following quality metrics, as defined in Section 3.2.1, that are mainly affected by the ranking of a graph layout:

- **Average Edge length:** the length of an edge mainly depends on the rank difference between the endpoints of the edge. Since we prefer short edges (i.e., $QM_{avg\_length}(G)$ is as low as possible), we want to compute a global ranking such that we obtain a low average edge length.
- **Back Edges:** the back edge quality metric, $QM_{back\_edges}(G)$, is directly affected by the global ranking. In fact, from a global ranking follows, for each edge, whether it is a forward or back edge.
- **Flow Direction:** the flow direction quality metric, $QM_{flow}(G)$, is also slightly affected by the global ranking. However, it is not completely clear to which extent. Intuitively, the

global ranking affects the length of every edge and thereby the flow direction is also affected.

- **Area:** when there are more global ranks, in general, graph layouts will also be 'taller'. Therefore, the area quality metric $QM_{area}(G)$ is also affected.

Obviously, computing a global ranking such that all above mentioned quality metrics are optimized is not always possible because contradictions may occur. For example, if we limit the number of ranks because we want to optimize the area, then it might not be possible to create forward edges. Therefore, we want to prioritize the above-mentioned quality metrics. To this end, consider the understandability of the graph layouts. In particular, the process semantics visual property, as described in Section 3.3.1, is directly affected by the global ranking. Since this visual property closely relates to the $QM_{back\_edges}(G)$ quality metric, we deem this metric as the most important. The other quality metrics ($QM_{avg\_length}(G)$, $QM_{flow}(G)$, $QM_{area}(G)$) are considered less important.

We aim to compute a global ranking that satisfies the above-mentioned quality criteria. To this end, we present a novel algorithm that brings process mining and graph drawing together. By using the variations, obtained from the event log, we compute a global ranking. The details of this algorithm are explained in Section 4.1.1.

4.1.1 Variation Based Ranking

As stated in Section 4.1, we mainly want to optimize the $QM_{back\_edges}(G)$ quality metric. More specifically, we want to compute a global ranking such that we obtain (readable) graph layouts that properly show the data semantics. Obviously, in order to show data semantics, we have to consider the actual data, which is contained in the event log. Therefore, our goal is to translate the event log into a global ranking. For this purpose, consider the previously mentioned statement that, in a process, there is (often) some main structure or path which is relevant to understand the whole process [RBRB06, AEHK10]. For example, in Figure 4.2, this main structure consists of the paths highlighted in red and green. The main observation here is that the main structure of a process (graph) can be seen as a set of paths, i.e., sequences of activities (cases). By using the cases in the event log, we already obtain some semantic information. However, by just considering the cases, we do not know yet which cases contribute to the main process structure. That is why we consider the variations. Intuitively, the variation $v \in \mathbb{V}$ of largest size contains all cases that describe the most frequent process behaviour. Subsequent variations of smaller size describe less frequent behavior. Therefore, our variation based ranking algorithm processes the variations $\mathbb{V}$, obtained from the event log, one by one from most important to least important. While doing this, we incrementally build a hierarchical graph structure such that all global ranking requirements remain satisfied. After processing all variations, we extract the global ranking from this graph structure. The variation based ranking algorithm is shown in Algorithm 1.

Algorithm 1 Variation Based Ranking

```
1: procedure VariationBasedRanking(𝕍)
2:     Sort(𝕍)
3:     for each v in 𝕍 do
4:         while s = NewSequence(v) do
5:             ProcessSequence(s)
6:         end while
7:     end for
8:     NormalizeRanks(V̄)
9: end procedure
```

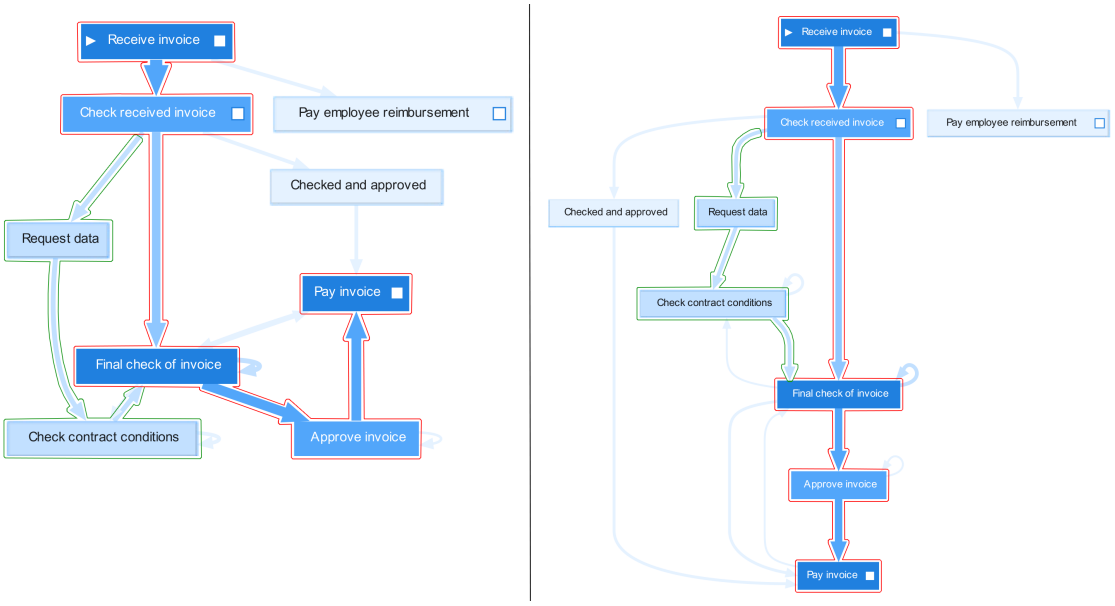

Figure 4.2: Illustration of two layouts computed for the same invoices graph. The layout on the left is computed by *dot* while the layout on the right is computed by our layout algorithm.

Before explaining algorithmic details, it is important to note that a variation can be translated into a sequence of nodes and edges. For example, let v be a variation with $\#_{seq}(v) = \langle A, B, C \rangle$. Then we have nodes A, B, C and edges (A, B), (B, C). More specifically, a variation can be translated into a sequence of interleaved nodes and edges as defined in Definition 4.1.2.

Definition 4.1.2 (Sequence) *A sequence s is an ordered list of interleaved nodes and edges. $nodes(s)$ and $edges(s)$ denote the ordered lists of nodes and edges in s respectively. The complete sequence of nodes and edges in s, is denoted similarly as a sequence of activities for a variation. For example, in the example above, the sequence for $\#_{seq}(v)$ is denoted as $s = \langle A, (A, B), B, (B, C), C \rangle$.*

In Algorithm 1, we start on line 2 by sorting the variations based on importance. There are several ways in which we can define importance of a variation and it is not immediately clear which definition provides the best results. The most naive definition simply considers the 'frequency' $|v|$ of a variation v, i.e., the number of cases in v. However, when two variations v have the same frequency, their order of importance is undefined. Therefore, we also consider $\#_{seq}(v)$. More specifically, let s be the sequence based on $\#_{seq}(v)$. Then, we also consider the $weight(e)$ of the edges $e \in edges(s)$. We denote the multiset of edge weights for some sequence as W. For every variation v, we compute an importance $imp(v)$ value which is used during the variation sorting. Table 4.1 lists the 12 sorting methods we consider.

As shown in Table 4.1, the first element on which sorting methods differ is a local versus global approach. In a local approach, variations are first sorted on their frequency $|v|$. After that, variations with the same frequency are locally sorted based on their $imp(v)$ values. On the other hand, the global approach simply sorts the variations on their $imp(v)$ values. Computation of the $imp(v)$ values is done based on the edge weights W and (depending on the sorting method) the variation frequency $|v|$. Column *Complete* in Table 4.1 lists how the $imp(v)$ value of a variation v is computed while the *Shorthand* column lists the notation we use to refer to the respective sorting method. There are some noteworthy aspects:

Type	**Complete,** $imp(v) =$	**Shorthand**
Local (L)	$\sum_{w \in W} w$	$\mathrm{L}\sum_{W}$
Local (L)	$\sum_{w \in W'} w$	$\mathrm{L}\sum_{W'}$
Local (L)	$\frac{\sum_{w \in W} w}{\lvert W \rvert}$	$\mathrm{L}\overline{W}$
Local (L)	$\frac{\sum_{w \in W'} w}{\lvert W' \rvert}$	$\mathrm{L}\overline{W}'$
Global (G)	$\sum_{w \in W} w + \lvert v \rvert$	$\mathrm{G}\sum_{W} + \lvert v \rvert$
Global (G)	$\sum_{w \in W} w \cdot \lvert v \rvert$	$\mathrm{G}\sum_{W} \cdot \lvert v \rvert$
Global (G)	$\frac{\sum_{w \in W'} w}{\lvert W' \rvert} + \lvert v \rvert$	$\mathrm{G}\overline{W} + \lvert v \rvert$
Global (G)	$\frac{\sum_{w \in W} w}{\lvert W \rvert} \cdot \lvert v \rvert$	$\mathrm{G}\overline{W} \cdot \lvert v \rvert$
Global (G)	$(\sum_{w \in W} w^2 + \lvert v \rvert^2)^2$	$\mathrm{G}(\sum_{W^2} + \lvert v \rvert^2)^2$
Global (G)	$(\sum_{w \in W} w^2 \cdot \lvert v \rvert^2)^2$	$\mathrm{G}(\sum_{W^2} \cdot \lvert v \rvert^2)^2$
Global (G)	$\sqrt{\sum_{w \in W} \sqrt{w} + \sqrt{\lvert v \rvert}}$	$\mathrm{G}\sqrt{\sum_{\sqrt{W}} + \sqrt{\lvert v \rvert}}$
Global (G)	$\sqrt{\sum_{w \in W} \sqrt{w} \cdot \sqrt{\lvert v \rvert}}$	$\mathrm{G}\sqrt{\sum_{\sqrt{W}} \cdot \sqrt{\lvert v \rvert}}$

Table 4.1: Every row represent a sorting method. The *Type* column lists whether sorting is done locally or globally. The *Shorthand* column lists the shorthand notation we use to refer to the respective sorting method. Finally, the *Complete* column lists how the $imp(v)$ value for a variation v is computed.

- The average of the edge weights is denoted by $\overline{W}$.
- $\sum_{W'}$ or $\overline{W}'$ also take the sum or average of edge weights, but only for a subset $W' \subseteq W$. Note that this is only used for the local sorting approaches. Basically, W' contains the weights of edges we have not encountered before. When we are locally sorting the variations, we keep track of the edges we have already encountered. Then, for every group of variations that we sort locally, when computing $imp(v)$, we only consider the edges not encountered thus far.

At this point, we do not know which sorting method(s) perform(s) the best. In Section 5.2, we provide an in-depth analysis of the different sorting methods.

After sorting, we process the variations in the for loop on line 3 from most important to least important. Intuitively, this means that we discover the most important process behaviour first. By doing this, we compute the global ranking such that the quality criteria, as discussed in Section 4.1, can be (mostly) satisfied for the main structure of the process (graph).

Based on Definition 4.1.2, we distinguish six different types of sequences, which are illustrated in Figure 4.3. As we can see, the sequences are distinguished based on whether the sequence starts or ends with a node or edge.

When we process a variation, procedure $NewSequence(v)$ on line 4 returns, starting from the start of the variation, all sequences s that contain nodes and/or edges we have not seen before. For example, let v_1, v_2, v_3 be three variations such that $\#_{seq}(v_1) = \langle A, B, C \rangle$, $\#_{seq}(v_2) = \langle A, D, C \rangle$ and $\#_{seq}(v_3) = \langle E, B, F \rangle$. Then, $NewSequence(v_1)$ returns type V sequence $s = \langle A, (A, B), B, (B, C), C \rangle$ because all of these nodes and edges have not been seen before. After that, $NewSequence(v_2)$ will return type VI sequence $s = \langle (A, D), D, (D, C) \rangle$ because nodes A and C have already been seen. Finally, $NewSequence(v_3)$ will first return type III sequence $s = \langle E, (E, B) \rangle$ and then type IV sequence $s = \langle (B, F), F \rangle$ because node B has already been seen before.

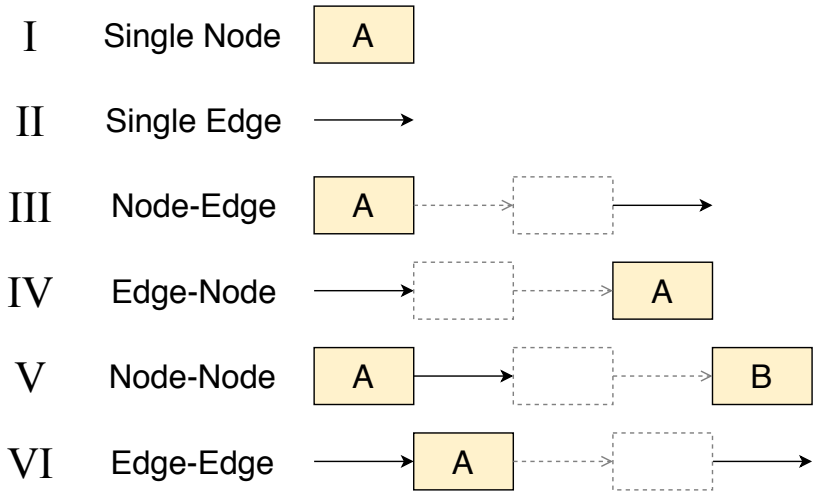

Figure 4.3: The different types of sequences we consider. Beige and black (non-dashed) elements represent the nodes and edges respectively which must be present in the sequence type. For example, sequence type VI must start and end with an edge, and therefore, must have at least one node in between. The dashed nodes and edges indicate that there can potentially be an arbitrary number of interleaved nodes and edges in between.

Procedure $ProcessSequence(s)$ on line 5 updates the global ranking structure. How this is done, depends on which sequence type is provided. In general, updates prevent horizontal edges and we adhere to the quality criteria discussed in Section 4.1. The update procedure for every sequence type is explained below. It is important to note that nodes $n \in \overline{V}$ are assigned $rank(n)$ values which can potentially become negative. Therefore, after processing all variations, procedure $NormalizeRanks(\overline{V})$ on line 8 normalizes all ranks such that the lowest rank has value 0. Consequently, in the following, when we refer to the lowest rank, we refer to the lowest $rank(n)$ value encountered so far. Additionally, while processing sequences, we keep track of connected components for the nodes/edges we have already seen. This information is (potentially) required when updating the global ranking, given some sequence. The details of using these connected components are given below for each sequence type separately (where required).

Type I: Single Node & Type V: Node-Node
When we have a sequence s that starts (and ends) with a node, we place the first node in s on the lowest rank and all subsequent nodes on subsequent ranks. Also, note that the node(s) in s form a new connected component.

Type II: Single Edge
When we have only a single edge (u, v), this implies that u and v have been seen before, and therefore, are already assigned to a global rank. Consequently, given the current global ranking, we can encounter three scenarios: forward edge, back edge, or horizontal edge.

When (u, v) is a forward edge, we are done because we want to create as many forward edges as possible.

In the case that (u, v) is a back edge and u and v are part of the same component, we do nothing. While in some cases, it would be possible to move nodes such that we transform (u, v) into a forward edge, this is generally not beneficial because this will always make some other (more important) edges longer. An example is shown in Figure 4.4. On the left, the back edge (u, v) is highlighted in red. On the right, by moving node v down, we can transform (u, v) into a forward edge. However, when doing this, edge (A, v) becomes significantly longer. Since (A, v) was already

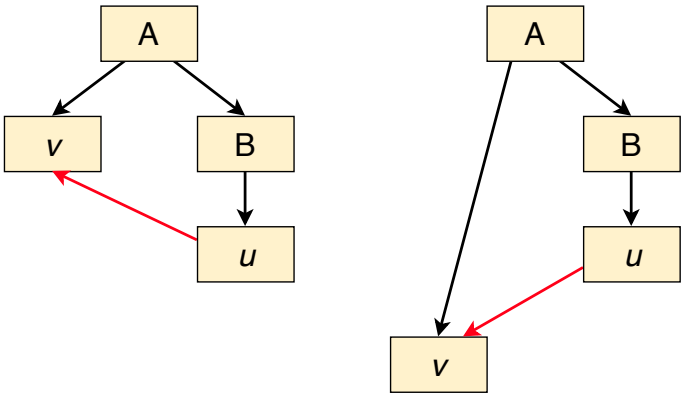

Figure 4.4: On the left, back edge (u, v) is highlighted in red. On the right, by moving node v down, we transform (u, v) into a forward edge. However, when doing this, edge (A, v) becomes significantly longer.

present, it is more important than edge (u, v). Therefore, we would rather have a back edge of low importance than (multiple) longer forward edges of higher importance.

When (u, v) is a back edge that connects two connected components, it is possible to create a forward edge by moving the component that v is part of. The details of this procedure are given in Algorithm 2. Note that in this case, u, v, and 1 are provided as parameters to Algorithm 2. *dist* is computed such that node v will be moved to *offset* ranks below $rank(u)$. By moving all nodes in the component of v over *dist* ranks, we obtain a forward edge of minimal length while all other requirements remain satisfied.

Algorithm 2 Merge Components

```
1: procedure MergeComponents(u, v, offset)
2:     dist = rank(u) − rank(v) + offset
3:     for each node w in comp(v) do
4:         rank(w) = rank(w) + dist
5:     end for
6: end procedure
```

In the case that (u, v) is a horizontal edge, we move v and all nodes reachable via a traversal, one rank down to create a forward edge. The traversal ignores the direction of edges and is started from v. Additionally, we only traverse edges of length 1 downward, i.e., when traversing an edge, we always reach a higher rank. The details of this procedure are given in Algorithm 3. Note that in this case, we run Algorithm 3 with parameters: u, v, and 1 respectively. An illustration of running Algorithm 3 with these parameters is shown in Figure 4.5. On the left and right, we have the global ranking before and after running Algorithm 3 respectively. As we can see, the horizontal edge is 'fixed' by moving v and all nodes reachable via a traversal down. Moreover, because we move the nodes we encountered during the traversal, we cannot create horizontal edges.

In Algorithm 3, *AddedNodes* refers to the set of nodes we have already added to the global ranking during the processing of the variations. $visited(n)$ is a value that indicates whether or not node n was visited during the graph traversal. Therefore, on line 2, we mark node u as visited to make sure we do not traverse u. Traverse refers to Algorithm 4, which recursively traverses the graph. In Algorithm 4, $OutEdges(node)$ and $InEdges(node)$ refer to the out and in edges of *node* respectively. Note that these are only the edges we have already seen while processing the variations. The if statements on lines 5 and 12 check whether traversing the current edge leads to a higher rank, i.e., if we move downwards. Additionally, we check if the length of the edge is

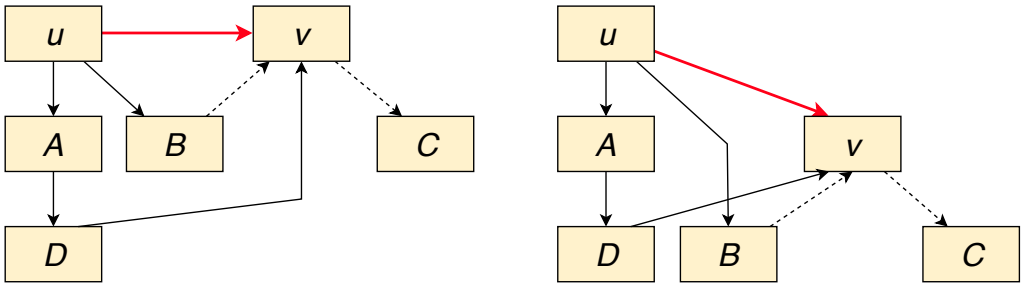

Figure 4.5: An illustration of running Algorithm 3 with parameters: u, v, and 1. On the left and right, the global ranking before and after running the algorithm respectively. The edge marked in red is the horizontal edge we are 'fixing'. Dashed edges indicate the edges that were traversed. Note that node D is not moved down because edge (D, v) has length longer than 1.

Algorithm 3 Shift Nodes

```
1: procedure ShiftNodes(u, v, numRanks)
2:     visited(u) = true
3:     Traverse(v, numRanks)
4:     for each node n in AddedNodes do
5:         if visited(n) and n ≠ u then
6:             rank(n) = rank(n) + numRanks
7:         end if
8:     end for
9: end procedure
```

smaller than or equal to $numRanks$. When the length of the edge is larger than $numRanks$, we do not have to move node v down (considering just this edge). For example, in Figure 4.5, node D was not moved down because the length of edge (D, v) was 2, which is larger than $numRanks = 1$.

Type III: Node-Edge & Type IV: Edge-Node
Let (u, x) and (y, v) be the first and last edge in a Edge-Node and Node-Edge sequence respectively. Then, in order to create forward edges, nodes in a Edge-Node sequence are placed below u and nodes in a Node-Edge sequence are placed above v. This concept is illustrated in Figure 4.6.

Type VI: Edge-Edge
Let s be the provided Edge-Edge sequence and let (u, x) and (y, v) be the first and last edge in $edges(s)$ respectively. Hence, nodes u and v are already present in the global ranking. Also, note that it is possible to have $x = y$ when $|nodes(s)| = 1$. The details of how s is processed are given in Algorithm 5. $First(edges(s))$ and $Last(edges(s))$ on lines 2 and 3, refer to the first and last edge in $edges(s)$ respectively. Essentially, there are three cases to consider.

The first, at line 4, is when u and v are part of different connected components. In this case, to create forward edges, we place the sequence of nodes on the ranks below u. Then, at line 10, we use Algorithm 2 to move v and the component v is part of. We move such that (y, v) becomes a forward edge of length 1.

The second case, at line 11, considers the scenario where $rank(v)$ is the same as or higher than $rank(u)$. When this happens, we place the sequence of nodes 'in between' u and v. Depending on the number of nodes, and number of ranks between u and v, there are two scenarios. If $freeRanks$, as shown on line 18, is large enough, we are done. However, if $freeRanks$ is too small, we have to make space. This is done by running Algorithm 3, as shown on line 19. An illustration of this procedure is shown in Figure 4.7 on the left.

Algorithm 4 Traverse

```
1: procedure TRAVERSE(node, numRanks)
2:     visited(node) = true
3:     for each (node, v) in OutEdges(node) do
4:         if not visited(v) then
5:             if rank(v) > rank(node) and rank(v) − rank(node) ≤ numRanks then
6:                 TRAVERSE(v, numRanks)
7:             end if
8:         end if
9:     end for
10:    for each (v, node) in InEdges(node) do
11:        if not visited(v) then
12:            if rank(v) > rank(node) and rank(v) − rank(node) ≤ numRanks then
13:                TRAVERSE(v, numRanks)
14:            end if
15:        end if
16:    end for
17: end procedure
```

Algorithm 5 Process Edge-Edge

```
1: procedure PROCESSEDGE-EDGE(s)
2:     (u, x) = First(edges(s))
3:     (y, v) = Last(edges(s))
4:     if comp(u) ≠ comp(v) then
5:         rank = rank(u) + 1
6:         for each node n in nodes(s) do
7:             rank(n) = rank
8:             rank = rank + 1
9:         end for
10:        MERGECOMPONENTS(u, v, |nodes(s)| + 1)
11:    else if rank(u) ≤ rank(v) then
12:        rank = rank(u) + 1
13:        for each node n in nodes(s) do
14:            rank(n) = rank
15:            rank = rank + 1
16:        end for
17:        freeRanks = rank(v) − rank(u) − 1
18:        if |nodes(s)| > freeRanks then
19:            SHIFTNODES(y, v, rank(y) − rank(v) + 1)
20:        end if
21:    else if rank(u) > rank(v) then
22:        rank = rank(u) − 1
23:        for each node n in nodes(s) do
24:            rank(n) = rank
25:            rank = rank − 1
26:        end for
27:        freeRanks = rank(u) − rank(v) − 1
28:        if |nodes(s)| > freeRanks then
29:            SHIFTNODES(v, y, rank(v) − rank(y) + 1)
30:        end if
31:    end if
32: end procedure
```

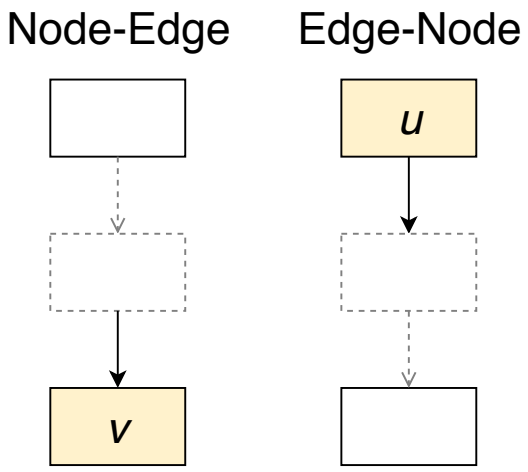

Figure 4.6: Visual representation of how the Type III Node-Edge and Type IV Edge-Node sequence types are processed. Nodes u and v in beige represent the nodes which were already inserted into the global ranking. For the Node-Edge case (on the left), we simply add the chain of nodes to the ranks above node v. For the Edge-Node case (on the right), we simply add the chain of nodes to the ranks below node u. Consequently, we create forward edges.

Finally, at line 21, we consider the scenario where $rank(v)$ is smaller than $rank(u)$. This is an interesting scenario, because there are quite some ways in which we can handle this. However, considering that v is on a lower rank than u, it is very likely that v occurs before u in the process. Hence, the sequence of nodes in s probably represents a sequence of activities that loop back to an earlier activity in the process. Therefore, to show that we go back in the process, we place $nodes(s)$ such that we obtain a sequence of back edges. By doing this, we make a trade off between flow consistency and data semantics. Again, similar to the previous case, we check on line 28 if there are enough free ranks for $nodes(s)$. If there are not enough ranks, we use Algorithm 3 on line 29 to make space. An illustration of this procedure is shown in Figure 4.7 on the right.

4.1.2 Unforeseen Edges

As stated before, the global ranking essentially computes a ranking for $\overline{G}$. Consequently, no edge $e \in \overline{E}$ is horizontal and thus, the global ranking can be used for any graph $G \subseteq \overline{G}$. One issue, however, occurs when a graph G contains edges which are not in $\overline{E}$. This can happen when activities are filtered out. For example, if we have a case c with sequence $\#_{seq}(c) = \langle A, B, C \rangle$ and activity B is filtered out. Then, in order to show that there is a path from A to C, an edge (A, C) is added to G. When A and C happen to be on the same global rank, this results in a horizontal edge. Note that, given the way our algorithms works and the fact that we have sequence $\#_{seq}(c) = \langle A, B, C \rangle$, this is a very rare scenario. Nevertheless, we still want to be able to compute a layout for such a graph.

The first solution we consider, is to first determine the set of edges $\overline{E}'$ that can potentially be created after filtering out activities. The global ranking is then computed using $\overline{E} \cup \overline{E}'$. The main disadvantage of this approach is that, for a single graph, only a subset of the edges in $\overline{E}'$ can be present at the same time. Moreover, it is very unlikely that one of those edges is actually horizontal. Additionally, the edges $\overline{E}'$ can significantly affect the global ranking and thereby also the quality of graph layouts. Hence, considering all possible edges $\overline{E}'$ is excessive and therefore we discard this solution.

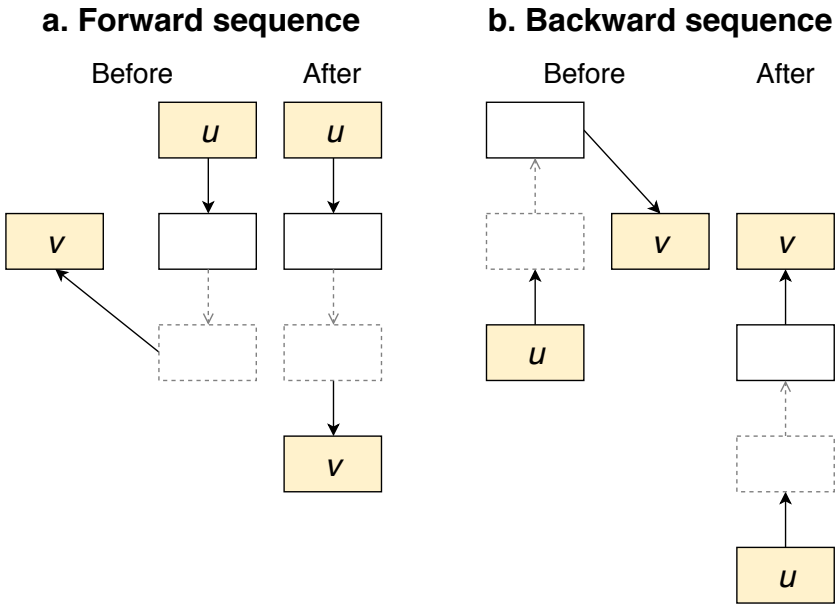

Figure 4.7: Illustration of how sequence type VI Edge-Edge, when there are not enough free ranks, is handled. Nodes u and v in beige represent the nodes already present in the global ranking while the other nodes represent the nodes in the sequence. Additionally, we assume that $comp(u) = comp(v)$. On the left and right we have the cases where we create sequences of forward and back edges respectively. We start with the setting under 'Before'. By running Algorithm 3, we obtain sequences of forward and back edges respectively. This is shown under 'After'.

Because of the rare occurrence in practice, we simply 'fix' the horizontal edges similar to how we fix horizontal edges for the type II Single Edge case in our variation based ranking algorithm. Let G be a graph for which we have computed ranks based on the global ranking. Note that every rank in G uniquely maps to a global rank ψ_i in GR. Therefore, let ψ_i be a rank in G that contains one (or more) horizontal edge(s). We denote the set of horizontal edges on ψ_i as $hor(\psi_i)$. To make sure that no node is moved to a different (global) rank (other than ranks that are created to 'fix' the horizontal edges), we run Algorithm 6 for every rank ψ_i that contains (a) horizontal edge(s).

Algorithm 6 Fix Horizontal Edges

```
1: procedure FixHorizontals(G, hor(ψ_i))
2:     Sort(hor(ψ_i))
3:     for each e in hor(ψ_i) do
4:         ProcessSequence(e)
5:     end for
6:     InsertRanks(G)
7: end procedure
```

In Algorithm 6, since there can be multiple horizontal edges, we start by sorting the edges based on weight (from large to small) on line 2. After that, every edge is processed as a Type II Single Edge sequence by $ProcessSequence(e)$ on line 4. Finally, since horizontal edges cause extra ranks to be created, $InsertRanks(G)$ on line 6 updates the ranks of G. An illustration of using Algorithm 6 on a rank ψ_i is shown in Figure 4.8. As we can see, an extra rank ψ_i' is created. By doing this, Algorithm 6 only affects ψ_i while all other (global) ranks in G remain unaffected. Consequently, we obtain a valid ranking after fixing all horizontal edges in G.

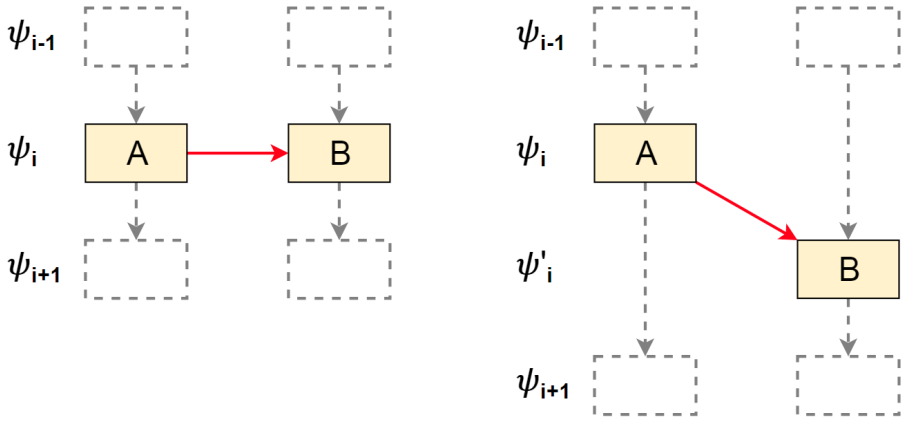

Figure 4.8: Illustration of how Algorithm 6 fixes a horizontal edge (highlighted in red) on a rank ψ_i. The adjacent ranks to ψ_i are represented by ψ_{i-1} and ψ_{i+1}. Dashed nodes and edges represent an arbitrary set of nodes and edges on the adjacent ranks. By moving node B down, we obtain a forward edge. Additionally, an extra rank ψ'_i is created.

4.1.3 Complexity

In this section, we consider the worst case running time of the variation based ranking algorithm. For readability purposes, Algorithm 7 is the same as Algorithm 1. Also, recall that $\mathbb{V}$ is the set of variations and that $\overline{V}$ and $\overline{E}$ are the sets of nodes and edges in $\overline{G}$, respectively.

For the sorting at Line 2, note that we do not know yet which sorting method performs the best. Therefore, in general, the sorting runs in $O(\mathbb{V} \log \mathbb{V})$ [CLRS09]. For the combination of the for loop at Line 3 and the while loop at Line 4, consider that every node and edge in $\overline{V}$ and $\overline{E}$, respectively, is handled exactly once. The worst case running times for each sequence type are listed in Table 4.2. Based on these times, a theoretical upper bound is $O(\overline{V}\overline{E})$. In theory, this happens when all edges are horizontal and in order to fix those edges, all nodes need to be traversed. In practice, however, this is a very unlikely scenario (if not impossible). The rank normalization at Line 8 runs in $O(\overline{V})$.

Unfortunately, it is not clear which term, $O(\mathbb{V} \log \mathbb{V})$ or $O(\overline{V}\overline{E})$, contributes more significantly to the running time. This depends on the sizes of $\mathbb{V}$, $\overline{V}$, and $\overline{E}$. In practice, however, the number of variations is (usually) much larger than the number of nodes/edges. Hence, we expect that the time required for sorting is the most significant. Nevertheless, we consider the worst case running time as $O(\mathbb{V} \log \mathbb{V} + \overline{V}\overline{E})$.

Algorithm 7 Variation Based Ranking

```
1: procedure VariationBasedRanking(𝕍)
2:     Sort(𝕍)
3:     for each v in 𝕍 do
4:         while s = NewSequence(v) do
5:             ProcessSequence(s)
6:         end while
7:     end for
8:     NormalizeRanks(V̄)
9: end procedure
```

Type	Complexity	Description
I Single Node	$O(1)$	We can always directly add this node
II Single Edge	$O(\overline{V})$	In order to fix a horizontal edge, we traverse all nodes in $\overline{V}$
III Node-Edge	$O(\overline{V})$	All nodes (except one) are in the given sequence
IV Edge-Node	$O(\overline{V})$	All nodes (except one) are in the given sequence
V Node-Node	$O(\overline{V})$	All nodes are in the given sequence
VI Edge-Edge	$O(\overline{V})$	We have to traverse all nodes in $\overline{V}$

Table 4.2: The table contains, for every sequence type, the worst case time required to handle such a sequence. The *Description* column describes the scenario in which this worst case time is required.

4.1.4 Edge Based Ranking

As we have seen, the main structure of a process (graph) is closely related to the weight of edges. Since the weight of an edge (u, v) represents how often u occurs before v, the main structure of a process (graph) (often) contains the edges of highest weight. Based on this idea, our first attempt at creating a global ranking that satisfies the quality criteria as specified in Section 4.1 directly considered the edges $\overline{E}$. A similar approach as in Algorithm 6 was used. First, the edges are sorted based on their weight. After that, the edges are processed one by one in a similar way as in Algorithm 1. Observe that in this algorithm, the only sequence type which we can not encounter is type VI Edge-Edge. All other sequence types (which all contain at most one edge), can be encountered. While this resulted in a simpler algorithm, we also found it to be too naive. The main problem lies in the fact that the edge weights are essentially an aggregation of the information in the event log. Because of this, it is not always possible to properly show the underlying process. Consider, for example, an event log with the following variations and their frequencies:

$$\begin{aligned}
\#_{seq}(v_1) &= \langle A, B, C, D\rangle, & |v_1| &= 97\\
\#_{seq}(v_2) &= \langle A, D\rangle, & |v_2| &= 50\\
\#_{seq}(v_3) &= \langle A, D, C\rangle, & |v_3| &= 48\\
\#_{seq}(v_4) &= \langle A, B, C\rangle, & |v_4| &= 2\\
\#_{seq}(v_5) &= \langle A, B\rangle, & |v_5| &= 1
\end{aligned}$$

Obviously, since v_1 occurs most frequently, we want activities A, B, C, D to be positioned in sequence. Our variation based ranking algorithm handles this properly, which is illustrated in Figure 4.9 on the left. The edge based ranking algorithm, however, fails to do this. Because of the aggregation, edge (A, D) has a higher weight than edge (C, D). Consequently, (A, D) is handled first, resulting in an undesirable global ranking (see Figure 4.9 on the right).

Another problem of the edge based ranking is that we do not obtain complete sequence information, i.e., we only obtain sequences with at most one edge. On the other hand, the variation based ranking does obtain this complete sequence information. As described in Section 4.2, this (complete) sequence information is required in our global order computation.

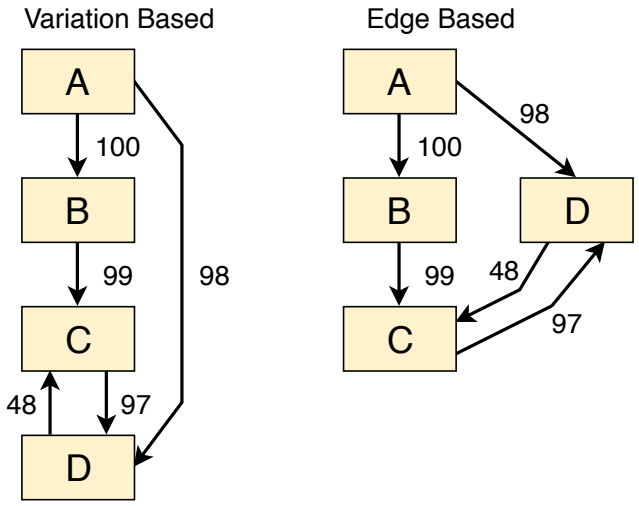

Figure 4.9: Illustration of global rankings computed, for the same event log, by the variation based algorithm (on the left), and the edge based algorithm (on the right). Because the edge based ranking algorithm only considers the (aggregated) edge weights, it fails to place node D below node C.

4.2 Global Order

By using a global ranking, we already obtain some graph stability. However, the global ranking only constrains the vertical movement of nodes. Horizontally, the nodes are unconstrained and can therefore move arbitrarily. For example, in Figure 4.10, we have two graph layouts which have been computed using the global ranking. Starting from the graph layout on the left, we obtain the graph layout on the right after filtering out the edge highlighted in red. Note that no significant layout change is required in order to handle the removal of this edge. Nevertheless, as we can see, *Checked and approved*, *Request data*, and *Check contract conditions* have been moved horizontally, while this was not necessary. Therefore, in order to obtain more graph stability, we also constrain the horizontal movement of nodes. In terms of our mental map definition [MELS95], this relates to all three mathematical models: restricted horizontal movement improves the preservation of the relative direction between nodes, nodes that are close together are more likely to remain close together, and graphical objects (nodes) in a region are more likely to stay in the same region.

In terms of the Sugiyama framework, horizontal positions are determined in a combination of two steps. In the node ordering step, the order of real and virtual nodes is determined for every rank. After that, in the node positioning step, the x-coordinate for every node is computed. These x-coordinates are computed such that the node order within each rank, as computed in the node ordering step, is adhered to. Therefore, since the node positioning should adhere to the 'constraints' (node order) of the node ordering step, we can not directly constrain node positions in the node positioning step because this could contradict the computed node order. Consequently, we constrain horizontal node movement in the node ordering step by first computing a global order that defines order constraints on node pairs (that are on the same rank). Then, by using a crossing minimization algorithm that adheres to the specified order constraints, we make sure the order constraints are satisfied. Note that, similar to the global ranking, the global order is global in the sense that it is computed based on $\overline{G}$, which implies it is applicable to any graph $G \subseteq \overline{G}$. Consequently, by always using the same order constraints for any graph G, we obtain graph stability.

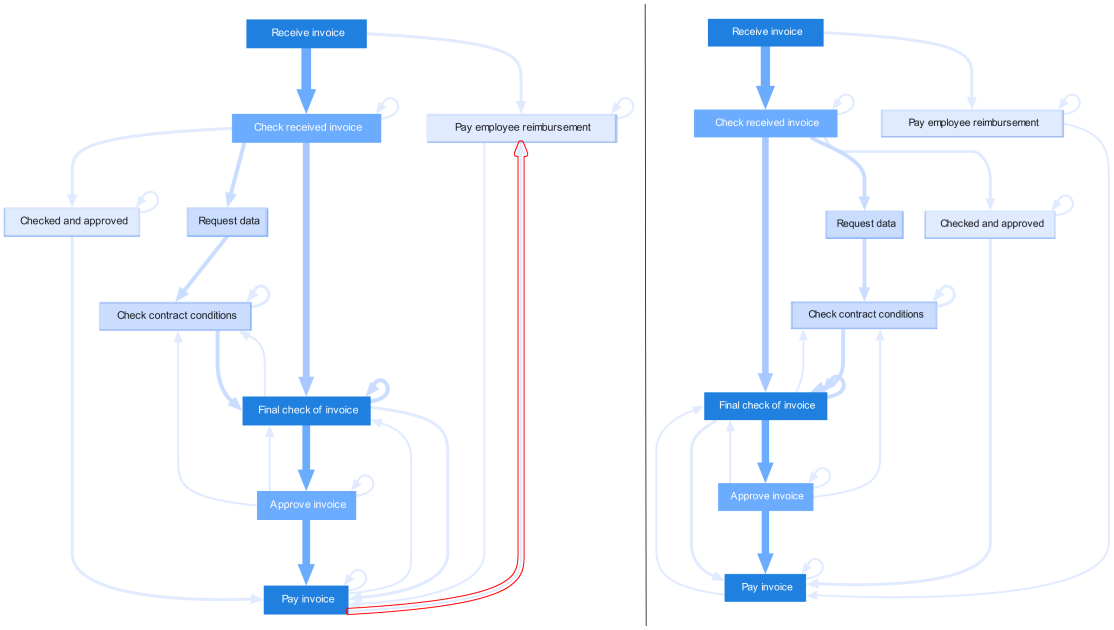

Figure 4.10: Illustrated are two graph layouts for the Invoices data set, computed by our layout algorithm which used the global ranking. Starting from the graph layout on the left, we obtain the graph layout on the right after filtering out the edge highlighted in red. Due to the removal, three nodes move horizontally.

While restricting horizontal node movement ensures graph stability, we want to do this such that we still obtain layouts of high quality, i.e, they should be readable and understandable. Therefore, consider the following quality metrics that are affected when we restrict horizontal node movement:

- **Crossings:** the edge crossings metric, $QM_{crossings}(G)$, is directly affected by the order of nodes. Therefore, extra edge crossings may occur when we restrict the horizontal node movement.

- **Average Edge Length:** the edge lengths cannot be optimized when movement is restricted. Therefore, the average edge length quality metric, $QM_{avg\_length}(G)$, is affected.

Since the number of edge crossings significantly affects graph readability [PCJ97, Pur00], we consider the $QM_{crossings}(G)$ quality metric to be more important.

While the above-mentioned quality metrics mainly relate to the readability of a graph layout, we also want understandable graph layouts. More specifically, we want to satisfy the centrality & node alignment visual property (as described in Section 3.3.2). Therefore, we explicitly consider this visual property in the design of our global order algorithm.

Constraining horizontal node movement is twofold, i.e., we have to compute node order constraints (a global order) and require a crossing minimization algorithm that can handle these constraints. Therefore, in Section 4.2.1, we describe the concept of a sequence based order. After that, in Section 4.2.2, we list requirements for the global order. Then, in Section 4.2.3, we describe a novel algorithm that computes a global order. Finally, in Section 4.2.4, we describe two crossing minimization algorithms that are able to handle node order constraints.

4.2.1 Sequence Based Order

For the global order, there are again two aspects to consider: layout quality and layout stability. For the stability, we want to define node order constraints such that the layouts actually remain stable. The main problem with node order constraints, however, is that real nodes on different ranks have no direct order relationship, i.e., the node order on one rank is independent of the node order on another rank (in terms of position). For example, in Figure 4.10, *Check received invoice* has order 0 because it is the leftmost node on its rank and *Request data* has order 1. Nevertheless, *Request data* is positioned left of *Check received invoice*. Therefore, in order to still be able to constrain the positions of real nodes, we also take the order of virtual nodes (edges) into account. In Figure 4.10, for example, *Request data* could be placed right of *Check received invoice* by adding the constraint that *Request data* should be placed right of any virtual node part of the edge (*Check received invoice*, *Final check of invoice*).

So, layout stability can be achieved by computing node order constraints for both real and virtual nodes. However, we want to do this such that we obtain layouts of high quality. In Section 4.2 we stated that this mainly implies that we want to reduce edge crossings (and possibly reduce edge length) and that we want to satisfy the centrality & node alignment visual property. In order to get an intuitive idea of how we should compute such node order constraints, consider again that, in a process, there is (often) some main structure or path which is relevant to understand the whole process [RBRB06, AEHK10]. More specifically, in Definition 4.2.1, we define this main structure as the skeleton of a process (graph). Since the skeleton of the process (graph) is relevant to understand the process, it is important to compute a graph layout such that the quality metrics and visual property, as described in Section 4.2, are satisfied for this skeleton. Therefore, a global order defines order constraints for the elements in the skeleton of the process (graph) such that the layout quality is optimized (as much as possible). Moreover, note that edges that are not part of the skeleton are unconstrained. By doing this, we lose some stability. On the other hand, however, the edge crossings of these edges can be optimized because the crossing minimization algorithm can freely position these edges, improving layout quality.

Definition 4.2.1 (Skeleton) *The skeleton of a process (graph) contains all nodes $\overline{V}$ and a subset of the edges $\overline{E}$. More specifically, this subset is the set of sequence edges (see Definition 4.2.3).*

In order to 'discover' the skeleton of a process (graph), consider that during the global ranking computation, sequences are discovered from most important to least important. Additionally, the skeleton contains all nodes $\overline{V}$. Therefore, intuitively, the set of sequences that contain at least one node, represents the skeleton of the process (graph). For example, Figure 4.11 illustrates the invoices data set for which the graph layout was computed using the global ranking. Additionally, the node sequences (see Definition 4.2.2), as discovered during the global ranking computation, are outlined in red and the numbers indicate the order of discovery. As we can see, the sequence of activities:

$\langle Receive\,invoice, Check\,received\,invoice, Final\,check\,of invoice, Approve\,invoice, Pay\,invoice\rangle$

occurs the most. And indeed, the first node sequence discovered by the global ranking algorithm contains this path. So, the node sequences provide us information about the process, and therefore, we consider the set of node sequences to be the skeleton of the process (graph). Consequently, the sequence edges (see Definition 4.2.3) are also part of the skeleton.

Definition 4.2.2 (Node sequence) *A sequence according to Definition 4.1.2, which contains at least one node.*

Definition 4.2.3 (Sequence edge) *An edge which is part of a node sequence.*

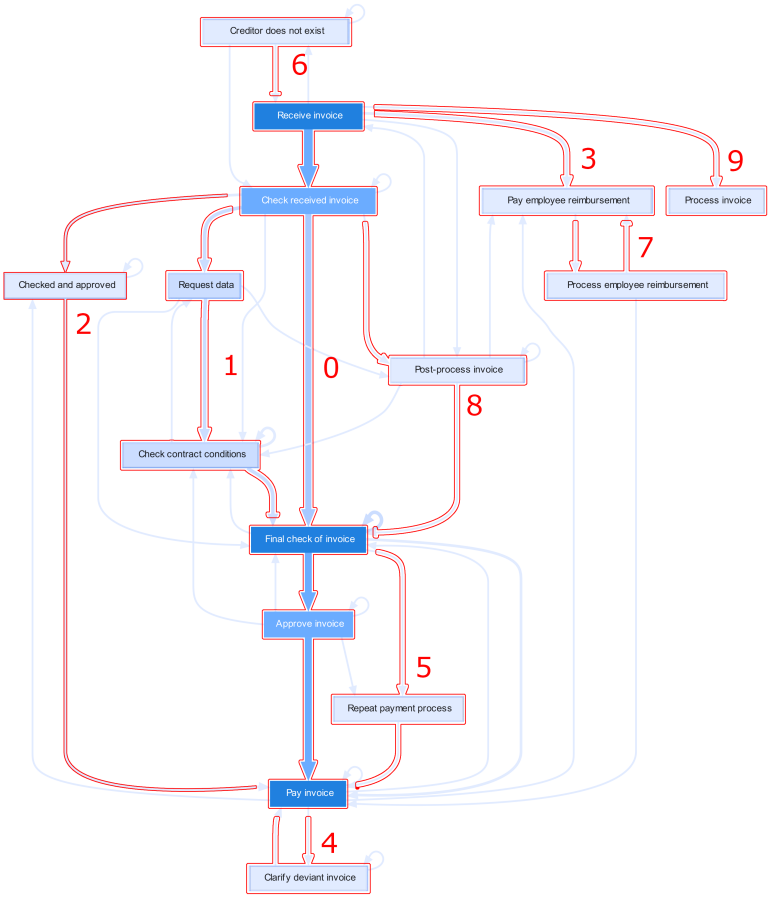

Figure 4.11: Illustration of the order in which the node sequences are discovered by the global ranking algorithm. Sequences are outlined in red. The number next to a red outline indicates the order of discovery.

The global order defines, for every global rank, the order in which the node sequences should be placed. To this end, based on the global ranking and the skeleton of the process (graph), we define the hierarchical undirected node sequence graph $NSG = (V_{nsg}, E_{nsg})$. The set of nodes V_{nsg} contains a number of *real sequence nodes*, representing the nodes in the skeleton (all nodes in $\overline{V}$). The set of edges E_{nsg} consists of the sequence edges which are split up into sequences of virtual nodes and (undirected) virtual edges. Therefore, V_{nsg} also contains a set of *virtual sequence nodes*, representing the sequence edges. Intuitively, since every rank in the NSG maps to a global rank, the real and virtual sequence nodes represent the presence of a node sequence on a certain global rank. Consequently, any order permutation of the sequence nodes results in a global order. The node sequence graph for the invoices data set is shown in Figure 4.12, where squares and circles represent real and virtual sequence nodes respectively. The number in each node corresponds to the sequence numbering in Figure 4.11. Note that the order on the ranks is the same as in Figure 4.11, and is therefore already the final global order. In Section 4.2.3, we describe the algorithm used to compute this global order.

Given the node sequence graph, any order permutation of the sequence nodes on the ranks results in a valid global order, i.e., for every real node, we can determine which sequences it should remain left/right of. Obviously, we aim to find a global order that satisfies the aforementioned quality criteria. To this end, we define some global order requirements in Section 4.2.2. After that, in Section 4.2.3 we describe our approach towards computing such a global order.

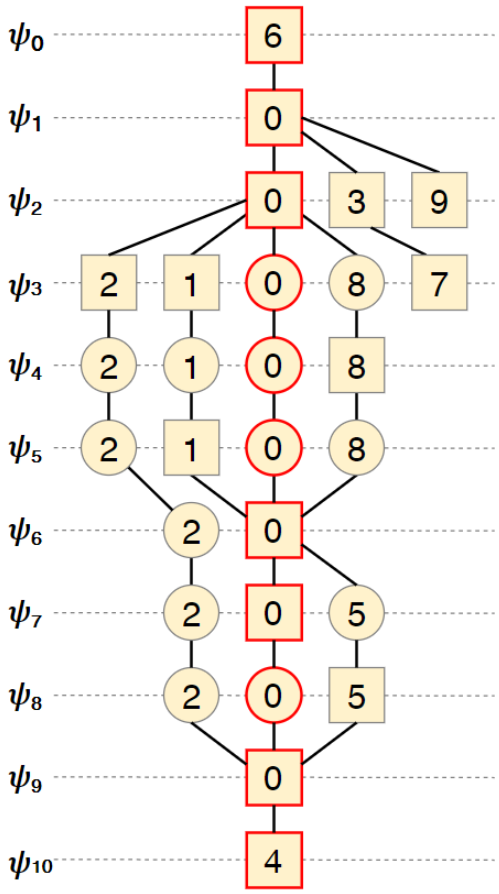

Figure 4.12: Visual representation of the node sequence graph of the invoices data set. We have the same ranks ψ_i as in the global ranking. Squares and circles represent real and virtual sequence nodes, respectively. Sequence nodes with a red border belong to the backbone (see Definition 4.2.5).

4.2.2 Global Order Requirements

Any order permutation of the sequence nodes on the ranks of the node sequence graph results in a valid global order. However, we want to compute a global order such that we obtain graphs of high quality. More specifically, we want a global order such that we reduce edge crossings (and possibly reduce edge length) and we want to satisfy the centrality & node alignment visual property. In this section, we describe how these quality criteria relate to the node sequence graph.

Let $\mathbb{S}$ be the set of node sequences. A node sequence $s \in \mathbb{S}$, has a unique identifier $id(s)$ which corresponds to the order of discovery during the global ranking algorithm, e.g., for the first s_1 and second s_2 sequence discovered, we have $id(s_1) = 0$ and $id(s_2) = 1$ respectively. Furthermore, we define sequence connectedness of a pair of sequences as follows:

Definition 4.2.4 (Sequence connectedness) *A value indicating how connected two node sequences are. More specifically, two node sequences $s_1, s_2 \in \mathbb{S}$ have a connectedness value $conn(s_1, s_2) = \sum_{e \in \gamma(s_1,s_2)} weight(e)$ where $\gamma(s_1, s_2)$ is the set of edges that start/end in $nodes(s_1)$ and start/end in $nodes(s_2)$.*

As stated before, the skeleton of the process (graph) already helps the user to understand the process. Next to some main structure, a process (often) contains a main path. Intuitively, this main path can be thought of as the most frequently occurring behavior. For example, in Figure 4.11, this main path is:

$\langle Receive\,invoice, Check\,received\,invoice, Final\,check\,of invoice, Approve\,invoice, Pay\,invoice\rangle$

Since the main path describes the most frequent behavior, we consider it to be the most important in the process. Consequently, we want to center it in the graph layout(s), i.e., by doing this we satisfy the centrality & node alignment visual property. We define this main path as the backbone of the graph (see Definition 4.2.5). The idea is that the backbone forms the center of the graph layout(s), and that all other nodes are positioned around this backbone. By taking, for each rank, the real or virtual sequence node with lowest id, the backbone contains, for every rank, the sequence which was discovered there first. Intuitively, this is the most important part of the graph. For example, in Figure 4.12, the nodes with a red border belong to the backbone. Note that a global rank always contains at least one node. Therefore, the backbone is defined for every global rank.

Definition 4.2.5 (Backbone) *A set of real or virtual sequence nodes, such that, for every rank r in the node sequence graph, the sequence node with lowest id belongs to the backbone. We denote this set of (virtual) sequence nodes as backbone(NSG).*

Given the node sequence graph, we wish to compute, for every rank, a sequence node permutation such that we obtain a 'good' global order. Therefore, we define the following three desirable properties. The first two mainly relate to the readability of graph layouts. The third relates to the understandability.

- **Connectedness:** node sequences that have a high connectedness value, should be placed next/close to each other. This especially holds for node sequences which have a high connectedness to the backbone. By doing this, edges between adjacent node sequences are shorter, i.e., $QM_{avg\_length}(G)$ is improved. Moreover, shorter edges are also less likely to cross, and therefore, this property also improves $QM_{crossings}(G)$.
- **Crossing minimization:** we do not want unnecessary sequence edge intersections. Since these sequence edges are part of the skeleton, they are (generally) more important and help the user to understand the process. Therefore, we deem it important to reduce the edge crossings of these edges. By doing this, $QM_{crossings}(G)$ is improved. For example, the node sequence graph as illustrated in Figure 4.12 satisfies this property.
- **Balance:** this property directly relates to the desirable visual property of centrality & node alignment (as described in Section 3.3.2). Given the backbone, we wish to, for every rank, balance the sequence nodes around the backbone. By doing this, the backbone ends up in the center of the node sequence graph. Consequently, in general, the backbone is also in the center of the graph layout(s) we compute.

In Section 4.2.3, we describe a novel algorithm, designed to compute a global order that satisfies the three properties above.

4.2.3 Global Order Computation

Given the node sequence graph $NSG = (V_{nsg}, E_{nsg})$, we want to compute, for every rank in the NSG, an order permutation of the sequence nodes such that we satisfy the quality criteria

specified in Section 4.2.2. To this end, observe that every (virtual) sequence node $v \in V_{nsg}$ belongs to a node sequence s, which we denote as $seq(v)$. Also, we denote the ranking of the NSG by $ranking(NSG)$. Note that a rank $r \in ranking(NSG)$ is essentially a subset of V_{nsg}. We define the backbone connectedness of a node as follows:

Definition 4.2.6 (Backbone connectedness) *A value indicating how connected a node sequences is to the backbone. Let $seq(backbone(NSG))$ be the set of sequences belonging to the backbone nodes, i.e, $seq(backbone(NSG)) = \{seq(v) | v \in backbone(NSG)\}$. Then, a node sequence s has backbone connectedness value $bconn(s) = \sum_{s' \in seq(backbone(NSG))} conn(s, s')$*

The global order algorithm is described in Algorithm 8.

Algorithm 8 Global Order

```
1: procedure GlobalOrder(NSG(V_nsg, E_nsg))
2:     for each rank r in ranking(NSG) do
3:         ConnectednessSort(r)
4:     end for
5:     comp = FindComponents()
6:     Balance(comp)
7: end procedure
```

The function ConnectednessSort(r) on Line 3 sorts the sequence nodes v on rank r based on backbone connectedness $bconn(seq(v))$. Moreover, sorting is done such that the backbone node on rank r always ends up at order 0 (leftmost position). By sorting like this, we satisfy the connectedness property (as specified in Section 4.2.2). Figure 4.13 shows the NSG of the invoices dataset after running ConnectednessSort(r). As we can see, on every rank, the backbone sequence nodes are placed at the leftmost position, while all other sequence nodes are sorted based on their backbone connectedness value.

As we can see in Figure 4.13, we still miss two of the desirable properties (see Section 4.2.2): crossing minimization and balance. Crossing minimization could be done by running a crossing minimization algorithm on the NSG. This could, however, break the backbone connectedness sort. Therefore, in order to fix sequence edge crossings and to balance the NSG, we move sequence nodes to the left of the backbone nodes (see Line 6). Obviously, we want to do this such that we do not introduce extra sequence edge crossings. Therefore, we first find connected components (see Line 5) that will be moved to the left of the backbone as a whole. $FindComponents()$ essentially considers the NSG without the backbone sequence nodes and then finds all connected components. Once all connected components have been found, $Balance(comp)$ considers the components from large to small (where the size of a component is defined as the number of sequence nodes in the component). For every component, $Balance(comp)$ checks whether moving that component to the left of the backbone improves the balance, i.e., if the ratio between the number of sequence nodes left/right of the backbone improves. Additionally, when moving a component to the left of the backbone, the connectedness sort order is preserved, i.e., sequence nodes that are closer to the backbone than other sequence nodes (on the same rank) will always remain closer. Figure 4.12 illustrates the NSG after balancing the sequence nodes. As we can see, the NSG is crossing free and balanced.

After running Algorithm 8, we can obtain the global order directly from the sequence node order permutation in the NSG. More specifically, every real sequence node $v \in V_{nsg}$ maps directly to a node $w \in \overline{V}$. Additionally, the virtual sequence nodes $x \in V_{nsg}$ belong to sequence edges, which directly map to an edge $e \in \overline{E}$. So, the global is essentially defined on the (virtual) sequence nodes V_{nsg}. Since these (virtual) sequence nodes map to $\overline{V}$ and $\overline{E}$, we can use the global order during the layout computation of a graph $G \subseteq \overline{G}$.

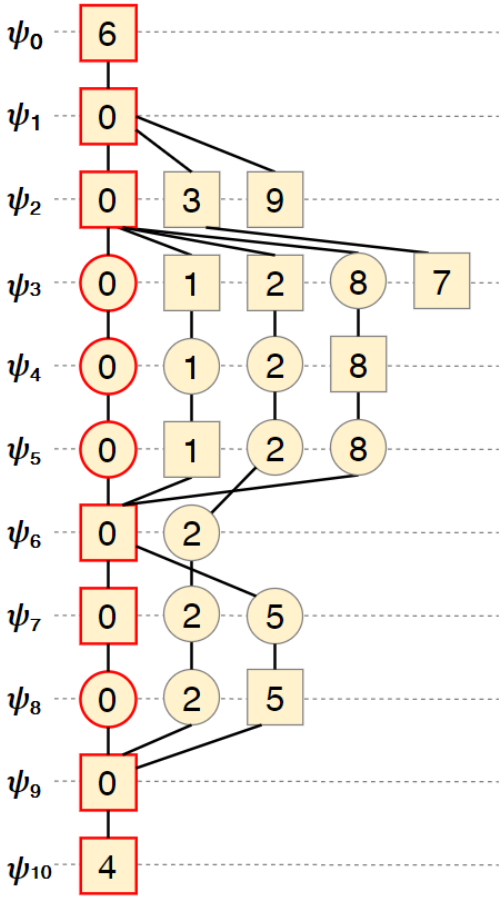

Figure 4.13: Visual representation of the node sequence graph after sorting each rank based on backbone connectedness. Squares and circles represent real and virtual sequence nodes respectively. Sequence nodes with a red border belong to the backbone.

4.2.3.1 Complexity

In this section, we consider the worst case running time of the global order algorithm. The ConnectednessSort(r) at Line 3 in Algorithm 8 takes $O(V_{nsg} \log V_{nsg})$ since we are essentially sorting all sequence nodes. Finding all components at Line 5 can be done by traversing the NSG, which takes $O(V_{nsg} + E_{nsg})$ time. Balancing at Line 6 takes $O(V_{nsg})$ time since we have at most $|V_{nsg}|$ components. Therefore, the worst case running time of the global order computation is $O(V_{nsg} \log V_{nsg} + E_{nsg})$.

4.2.4 Crossing Minimization

Computing a global order is not enough, we also have to enforce this global order when computing a layout for a graph $G \subseteq \overline{G}$. More specifically, during the node ordering step, we are given a set of ranks where every rank maps to exactly one global rank $\psi_i \in GR$. For every rank ψ_i, the global order defines order constraints for the (both real and virtual) sequence nodes on ψ_i. Ideally, we want to compute an order permutation for every rank such that we both satisfy the order constraints and minimize the number of edge crossings. Unfortunately, edge crossing minimization is NP-Complete [EMW86], and therefore, minimizing the number of edge crossings within a reason-

able (practical) running time is not always possible. Because of this, in practice, algorithms reduce edge crossing minimization for a graph G to a sequence of one-sided two-level [For04] (where level refers to a rank) crossing minimization problems. Given an order permutation for the first rank, an order permutation for the second rank is computed such that the number of crossings between the first two ranks is reduced. After that, the same is repeated for the second and third rank. This is repeated for all ranks, alternating between a top down and bottom up rank traversal, until some termination criterion is met. In this work, we present two crossing minimization algorithms based on this concept. The only difference is that we have a sequence of constrained one-sided two-level crossing reduction problems [For04]. We again have two levels (ranks) where the order permutation of the first rank is fixed. For the second rank, we want to compute an order permutation while adhering to the global order.

Gansner et al. [GKNV93] introduce an edge crossing minimization algorithm, which is used by *dot*. The two crossing minimization algorithms we present are based on the algorithm of Gansner et al. [GKNV93], which is shown in Algorithm 9. The first algorithm is the same as used by *dot* to handle order constraints. The second algorithm is a novel edge crossing minimization algorithm that uses the information of the global order and the backbone. In Section 5.2, we provide an in-depth analysis of both algorithms.

Algorithm 9 Crossing Minimization

```
1:  procedure Crossing Minimization(G)
2:      order = InitOrder()
3:      best = order
4:      for i = 0 to MaxIterations do
5:          wMedian(order, i)
6:          Transpose(order)
7:          if crossing(order) < crossing(best) then
8:              best = order
9:          end if
10:     end for
11:     return best
12: end procedure
```

Order constraints in *dot*
Dot implements functionality that can handle horizontal edges in a graph layout [GKN15]. Since *dot* aims to draw those edges from left to right, they can essentially be used as order constraints. For example, consider some rank with nodes u and v. If we want to enforce node u to be left of node v, we add the edge (u, v). Foster et al. [For04] mention two crossing minimization approaches (Sander et al. [San98] and Waddle et al. [Wad00]), which are similar to the approach of *dot*, that can handle order constraints. While these approaches differ slightly, the general idea is the same: given an order permutation that satisfies all order constraints, the wMedian(order, i) (on Line 5) and Transpose(order) (on Line 6) functions are modified such that updates are only allowed when they do not violate the order constraints.

In this work, we implement the same approach as *dot* in order to satisfy order constraints. wMedian(order, i) and Transpose(order) are modified such that updates only happen when they do not violate the global order. InitOrder() (on Line 2) is modified such that the initial order permutation satisfies the global order constraints. Basically, we first use the standard implementation of InitOrder(). After that, positions of (constrained) sequence nodes are reordered such that order constraints are satisfied. In the remainder of this work, we refer to this algorithm as Mincross.

Relative Order Computation
Observe that the global order only defines order constraints on the sequence nodes and sequence edges (and consequently all virtual sequence nodes belonging to those edges), i.e., all nodes $v \in$

$\overline{V}$ and a subset of the edges $\overline{E}$. Intuitively, because the global order is always the same, the position/order of these elements relative to each other is always the same. For example, if a node is left of the backbone in one graph, then it is always left of the backbone. Based on this idea, we introduce a novel crossing minimization algorithm called RelMincross that uses backbone and global order information in order to minimize edge crossings.

The main algorithmic structure of RelMincross is similar to Algorithm 9. The main differences include: InitOrder() is modified, wMedian(order, i) is not included anymore, and Transpose(order) is modified in the same way as in Mincross. The main idea of the algorithm is that (virtual) sequence nodes have a fixed order on each rank (which is based on the global order). Using this fixed order, we 'approximate' where a non-sequence edge should be placed in the overall order. This order initialization is done in InitOrder(). After that, Transpose(order) is used to 'fix' remaining edge crossings.

The fixed order value $fix(v)$ of a (virtual) sequence node v is a value between -1 and 1. Negative values represent nodes left of the backbone while positive values represent the inverse. Consequently, backbone nodes have a $fix(v)$ value of 0. $fix(v)$ value computation is done for each rank separately. First, the backbone node (which is the sequence node v with lowest $id(seq(v))$ value) on a rank is identified and assigned value 0. After that, based on the global order, the (virtual) sequence nodes left and right of the backbone node are uniformly assigned $fix(v)$ values in the ranges $(-1, 0)$ and $(0, 1)$, respectively. In Figure 4.14, a graph layout which was computed using RelMincross is shown. The red squares represent the virtual sequence nodes present in the graph layout and red numbers next to (virtual) sequence nodes indicate the $fix(v)$ values. As we can see, all backbone nodes have value 0. Nodes further to the right and left have lower and higher $fix(v)$ values respectively.

After computing the $fix(v)$ values for the (virtual) sequence nodes, InitOrder() computes an $order(v)$ value for all non-sequence virtual nodes. This is done for every edge separately. Let (x, y) be such a non-sequence edge. The $order(v)$ values of the virtual nodes belonging to (x, y) are then computed based on $fix(x)$ and $fix(y)$. More specifically, the sequence of virtual nodes is assigned $order(v)$ values that are uniformly distributed between $fix(x)$ and $fix(y)$. For example, in Figure 4.14, the virtual nodes of the edge (*Pay employee reimbursement, Pay invoice*) are indicated by green squares. The green numbers indicate the computed $order(v)$ values for the virtual nodes.

After computing all $fix(v)$ and $order(v)$ values, InitOrder() ends by sorting every rank based on the $fix(v)$ and $order(v)$ values. Note that, on a single rank, we can never have two nodes with the same $fix(v)$ value. It can occur, however, that there are two non-sequence virtual nodes (on the same rank) with the same $order(v)$ value. When this happens, we apply a sorting heuristic. First, we sort based on the length of the edges these non-sequence virtual nodes are part of, placing the shortest edge closer to the backbone to avoid edge crossings. If the edge lengths are the same, we sort on the weight of the edges, placing the heavier (more important) edge closer to the backbone. Observe that by sorting, we essentially position every non-sequence edge 'in between' its endpoints.

Finally, after running InitOrder(), the remainder of the algorithm uses Transpose(order) to 'fix' remaining edge crossings. As stated before, wMedian(order, i) is not used anymore because we do not compute any median values.

Complexity
In general, both Mincross and RelMincross have a similar running time as Algorithm 9. It is important to consider some differences, however. The first is that Gansner et al. [GKNV93] run Algorithm 9 twice to obtain the best results. The difference between the two runs is how the node order is initialized in InitOrder(). Since Mincross uses (the standard implementation

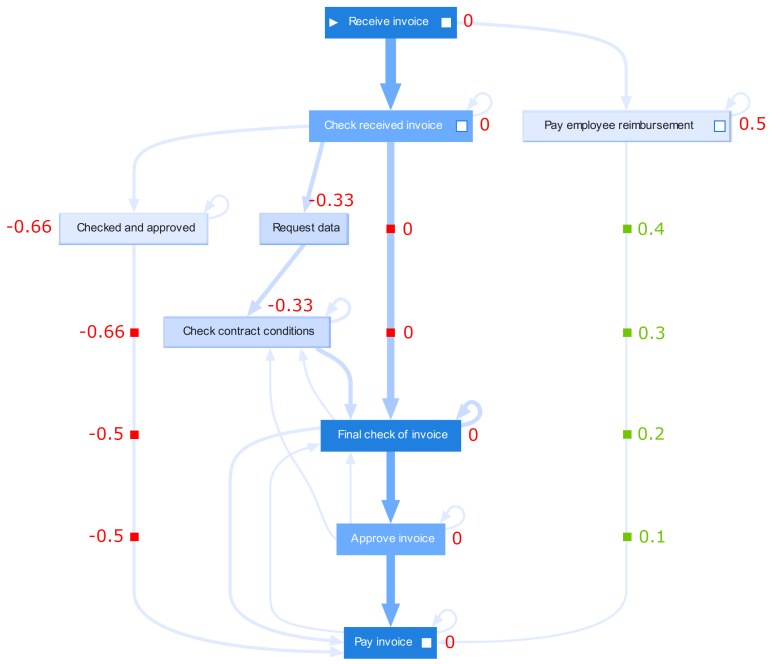

Figure 4.14: A graph layout computed using RelMincross. The red squares indicate all the virtual sequence nodes while the red numbers indicate the computed $fix(v)$ value for every (virtual) sequence node. The green squares indicate the virtual non-sequence nodes of the edge (*Pay employee reimbursement, Pay invoice*). Green numbers indicate the computed $order(v)$ value for the related non-sequence virtual nodes.

of) InitOrder(), we also run Mincross twice in order to obtain the best results. On the other hand, InitOrder() for RelMincross always computes the same initial node order. Therefore, RelMincross only runs once.

Another difference is that RelMincross does not use wMedian(order, i). Since wMedian(order, i) sorts all the nodes, we drop quite a significant part of the total (practical) running time.

Considering the two above-mentioned differences, in terms of (practical) running time, RelMincross runs faster. In Section 5.2, we provide an in-depth analysis of the two algorithms.

Duplicate global ranks

The issue discussed in this section is an implementation issue. However, since it is a direct consequence of the problem of unforeseen edges, as discussed in Section 4.1.2, we deem it important to discuss it here. Recall that (unforeseen) horizontal edges are fixed by adding extra ranks. Since we assume that every rank in a graph G uniquely maps to exactly one global rank, this causes issues. Consider, for example, Figure 4.15. On the left, we have a rank that maps to global rank ψ_i and we have a horizontal edge highlighted in red. After fixing this horizontal edge, we obtain an extra rank ψ_i' that also maps to global rank ψ_i. Consequently, the global order defines the same order constraints for these ranks. In principle, this is not a problem, unless fixing the horizontal edge causes a virtual sequence node to be created on ψ_i or ψ_i' when edges are split up in the node ordering step. In Figure 4.15, for example, assume that edge (A, C) is a sequence edge. After fixing the horizontal edge, a virtual sequence node is created on rank ψ_i' when edges are split up in the node ordering step. Since the global order does not define any order constraints for sequence edge (A, C) on ψ_i', this causes an issue in the crossing minimization algorithm. More specifically, the crossing minimization algorithm 'expects' order constraints for every virtual sequence node

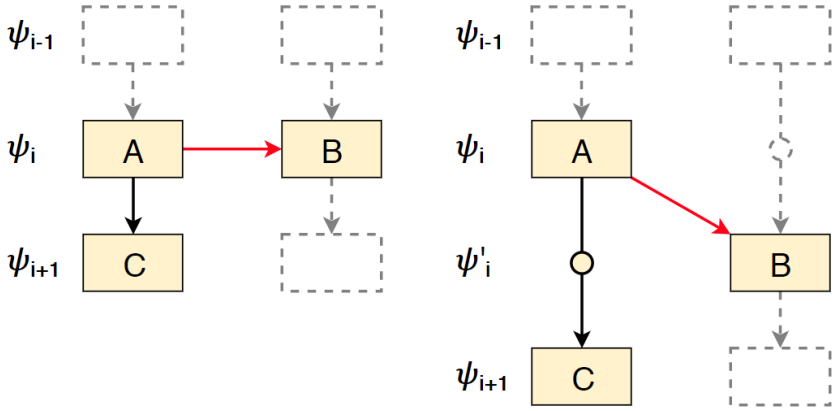

Figure 4.15: Illustration of how fixing a horizontal edge (highlighted in red) on a (global) rank ψ_i can cause issues. The adjacent ranks to ψ_i are represented by ψ_{i-1} and ψ_{i+1}. Dashed nodes and edges represent an arbitrary set of nodes and edges on the adjacent ranks. Circles represent virtual (sequence) nodes.

(because these are part of the global order). In this case, however, there are no order constraints for the virtual node belonging to edge (A, C) on ψ'_i. In order to solve this problem, we detect such virtual sequence nodes that are created due to fixed horizontal edges and change them to virtual nodes.

4.3 Approach Overview

In this section, we discuss how the global ranking and global order fit into the Sugiyama framework. Table 4.3 contains an overview of how our approach fulfills each of the Sugiyama framework steps. When we state "Same as *dot*", we refer to the algorithmic approach used by *dot* (as described in Section 2.3.1) for that specific step. As we can see, the "Cycle Removal" and "Rank Assignment" step are not required anymore because we can directly obtain the ranking from the global ranking. For "Node Ordering", it is not clear yet which crossing minimization algorithm performs the best. In Section 5.2, we provide an in-depth analysis of the two algorithms. "Node Positioning" is done using the same approach as *dot*. Finally, "Spline Drawing" is done using the approach of Mennens et al. [MSW17].

Sugiyama step	**Our Approach**
Cycle Removal	Follows from Global Ranking
Rank Assignment	
Node Ordering	MINCROSS or RELMINCROSS which both use the Global Order
Node Positioning	Same as *dot*
Spline Drawing	Same as Mennens et al. [MSW17]

Table 4.3: Every row represent a step in the Sugiyama framework. The *Our Approach* column lists how our approach fulfills each of the framework steps.

Chapter 5

Test Framework

In Chapter 3, we introduced several quality and stability metrics. In order to use these metrics, we implemented a test framework that automatically generates graphs and computes the quality and stability metrics. In Section 5.1, we shortly describe how the framework is set up. After that, we consider the different algorithmic configurations for our approach. More specifically, in Section 4.1.1, we proposed 12 variation sorting methods and in Section 4.2.4, we introduced two crossing minimization algorithms. For both the sorting methods and crossing minimization algorithms, we do not know which perform(s) the best. Therefore, in Section 5.2, we provide an in-depth analysis of the 24 (12 x 2) possible algorithmic configurations.

5.1 The Framework

The test framework allows us to automatically run several independent tests for multiple layout algorithms. More specifically, every layout algorithm runs the same tests, allowing us to compare the test results. A single test consists of a pair of randomly generated graphs $G_1, G_2 \subseteq \overline{G}$ which are obtained by randomly removing a subset of the edges $\overline{E}$ and then removing all nodes $v \in \overline{V}$ that become disconnected. Consequently, G_1 and G_2 are essentially random sub-graphs of $\overline{G}$. Note that in practice, users do not have just a pair of graphs. Rather, they have a sequence of graphs. Nevertheless, the tests are designed to be independent such that we can compare the results using statistical methods.

Running a test essentially comes down to computing the layouts for G_1 and G_2 with every layout algorithm. Consequently, the results of a test consist of the quality metrics $QM_\lambda(G_1)$, $QM_\lambda(G_2)$, and the stability metrics $SM_\lambda(G_1, G_2)$, for every layout algorithm. By running the same test with the different layout algorithms, we can statistically compare the test results in order to make claims about how well a certain algorithm performs.

In the test framework, we always use the same set of datasets (event logs). These are listed in Table 5.1. All of these datasets are obtained from a real process.

Name	Nodes	Edges	Cases	Variations	Description
BPI2012	36	188	23967	4366	Dataset of the second international business process intelligence challenge [2]
BPI2013	13	81	30239	1838	Dataset of the third international business process intelligence challenge [3]
BPI2014	39	772	88502	21000	Dataset of the fourth international business process intelligence challenge [4]
BPI2017	48	257	73364	16263	Dataset of the seventh international business process intelligence challenge [5]
BPI2017RCA	26	167	42995	2836	A simplified version of the BPI2017 dataset
BPI2018	41	594	466141	26104	Dataset of the eighth international business process intelligence challenge [6]
Electricity	22	300	144339	10874	Dataset containing process data of an electricity company
Hospital	18	144	100000	780	Dataset containing process data of a hospital
Insurance	13	87	99999	22149	Dataset containing process data of an insurance company
Invoices	15	48	25743	101	Our running example
P2P	50	615	80017	5807	Dataset containing process data of a purchase to pay process
Road	11	70	150370	171	Dataset of an information system managing road traffic fines
Sepsis	16	155	3383	806	Dataset containing process data of how sepsis (a potentially life-threatening complication of an infection) is handled

Table 5.1: The datasets used to test the algorithms.

5.2 Algorithm Configuration

In Section 4.1.1, we proposed 12 variation sorting methods and in Section 4.2.4, we introduced two crossing minimization algorithms. Since we do not know which sorting method and which crossing minimization algorithm works best, we use the test framework to run tests. In total, we consider 24 (12 x 2) different algorithms, which are listed in Table 5.2. As we can see, we use the same shorthand notation as introduced in Section 4.1.1 for the sorting methods. Additionally, M and R refer to MINCROSS and RELMINCROSS, respectively.

We have generated 500 tests for every dataset listed in Table 5.1, and every algorithm computed layouts for the graphs in each test. Consequently, due to test independence and a large sample size, we can assume, according to the Central Limit Theorem, that our metric results are normally distributed (per algorithm, per dataset). Given a normal distribution, we determine for every metric (per dataset) which algorithm(s) perform(s) significantly better. This is done by first running, for every metric, a one-way ANOVA test (significance level $\alpha = 0.05$). If the null-hypothesis of a one-way ANOVA test is rejected, this means there are at least two algorithms that have significantly different results for that metric. We do not know yet, however, which algorithms differ significantly. To determine this, we run a post-hoc Tukey Test (significance level $\alpha = 0.05$). Using the results of the Tukey Test, we can determine which algorithm(s) perform(s) significantly better for a certain metric.

To determine which algorithm performs the best (on average), we combine the statistical test results of all the datasets. An overview of these combined results is shown in Figure 5.1. All columns, except the first four, represent a metric (we refer to these columns as 'metric columns'), while each of the rows represents one of the algorithms. A value in a cell in one of the metric columns represents for how many of the datasets the related algorithm performs significantly better. For example, algorithm M:L$\sum_W$ has a significantly faster running time for two of the datasets. Next to the metric columns, we have the 'aggregation columns'. The *Sum* column contains, for every algorithm, the sum of the values in the metric columns. *qSum* and *sSum* also contain the sum of the values, but only for the quality and stability metrics, respectively. The *qSum (no Time)* column contains the sum of the quality metrics except for the time (see column *(QM) Time*) quality metric. Also, note that the rows have been sorted based on the *Sum* column values.

Considering Figure 5.1, we make the observation that G$(\sum_{W^2} \cdot |v|^2)^2$ is the best sorting method. When only considering the algorithms that use RELMINCROSS, then in both the *Sum* and *qSum* column it has the highest value. For the *sSum* column, there are only two algorithms that have one point higher. When only considering the algorithms that use MINCROSS, this sorting method performs the best in all aggregation columns. Therefore, given that G$(\sum_{W^2} \cdot |v|^2)^2$ is the sorting method of our choice, we have to decide which crossing minimization algorithm to use. More specifically, we have to choose between M:G$(\sum_{W^2} \cdot |v|^2)^2$ and R:G$(\sum_{W^2} \cdot |v|^2)^2$.

At first sight, considering the *Sum* column, RELMINCROSS appears to perform the best. There are some considerations, however. First, we deem the stability metric results as less relevant because both algorithms can only freely position the non-sequence edges. More specifically, the global ranking and global order constrain the nodes and sequence edges. Consequently, the difference in stability is mainly caused by the movement of the non-sequence edges. Since these non-sequence edges are generally of lower importance, we deem this difference in stability as less relevant.

Another consideration is the difference in quality between RELMINCROSS and MINCROSS. In column *qSum*, we observe that RELMINCROSS performs better with 43 points. Upon further inspection, we observe that there are two quality metrics for which the two algorithms perform significantly different (at least 3 points difference): *Time* and *Flow*. The flow quality metric

Algorithm Configurations	
M:L$\sum_W$	R:L$\sum_W$
M:L$\sum_{W'}$	R:L$\sum_{W'}$
M:L$\overline{W}$	R:L$\overline{W}$
M:L$\overline{W}'$	R:L$\overline{W}'$
M:G$\sum_W + \lvert v\rvert$	R:G$\sum_W + \lvert v\rvert$
M:G$\sum_W \cdot \lvert v\rvert$	R:G$\sum_W \cdot \lvert v\rvert$
M:G$\overline{W} + \lvert v\rvert$	R:G$\overline{W} + \lvert v\rvert$
M:G$\overline{W} \cdot \lvert v\rvert$	R:G$\overline{W} \cdot \lvert v\rvert$
M:G$(\sum_{W^2} + \lvert v\rvert^2)^2$	R:G$(\sum_{W^2} + \lvert v\rvert^2)^2$
M:G$(\sum_{W^2} \cdot \lvert v\rvert^2)^2$	R:G$(\sum_{W^2} \cdot \lvert v\rvert^2)^2$
M:G$\sqrt{\sum_{\sqrt{W}} + \sqrt{\lvert v\rvert}}$	R:G$\sqrt{\sum_{\sqrt{W}} + \sqrt{\lvert v\rvert}}$
M:G$\sqrt{\sum_{\sqrt{W}} \cdot \sqrt{\lvert v\rvert}}$	R:G$\sqrt{\sum_{\sqrt{W}} \cdot \sqrt{\lvert v\rvert}}$

Table 5.2: The different algorithms we consider.

essentially expresses the general flow direction of the edges. So, basically, MINCROSS computes layouts in which the edges are more vertical (and thereby more readable). For the time quality metric (see column *(QM) Time*), RELMINCROSS performs significantly better than MINCROSS (which we already suspected in Section 4.2.4). In practice, however, this difference is barely noticeable, especially for small graphs. Therefore, we would rather have a slower (but still fast enough) algorithm that computes layouts of high quality than a faster algorithm that computes layouts of poor quality. Therefore, we exclude the time quality metric (see column *qSum (no Time)*). As we can see, M:G$(\sum_{W^2} \cdot |v|^2)^2$ has the highest value in the *qSum (no Time)* column.

All in all, M:G$(\sum_{W^2} \cdot |v|^2)^2$ is slower and less stable but produces layouts of higher quality while R:G$(\sum_{W^2} \cdot |v|^2)^2$ is faster and more stable but produces layouts of lower quality. Given the considerations mentioned above, M:G$(\sum_{W^2} \cdot |v|^2)^2$ is the algorithmic configuration of our choice.

5.2.1 Variation Sorting

The main reason for testing this many different variation sorting methods, is that it is difficult to intuitively grasp which method works best. We can only speculate about which sorting method actually provides the best results. With the results in Figure 5.1, however, we can make some general observations:

- The global sorting methods that multiply the variation frequency ($|v|$) generally perform better than the global methods that add the variation frequency, indicating that variation frequency is an important indicator for how 'important' a variation is.
- There appears to be no clear significant difference between global methods that multiply the variation frequency (see the observation above) and local sorting methods. Recall that local sorting methods first sort based on variation frequency and then locally sort groups of variations with the same frequency. Therefore, similarly to global methods that multiply the variation frequency, local sorting methods essentially regard variations with a higher frequency as more important, again indicating that variation frequency is an important indicator for how 'important' a variation is.

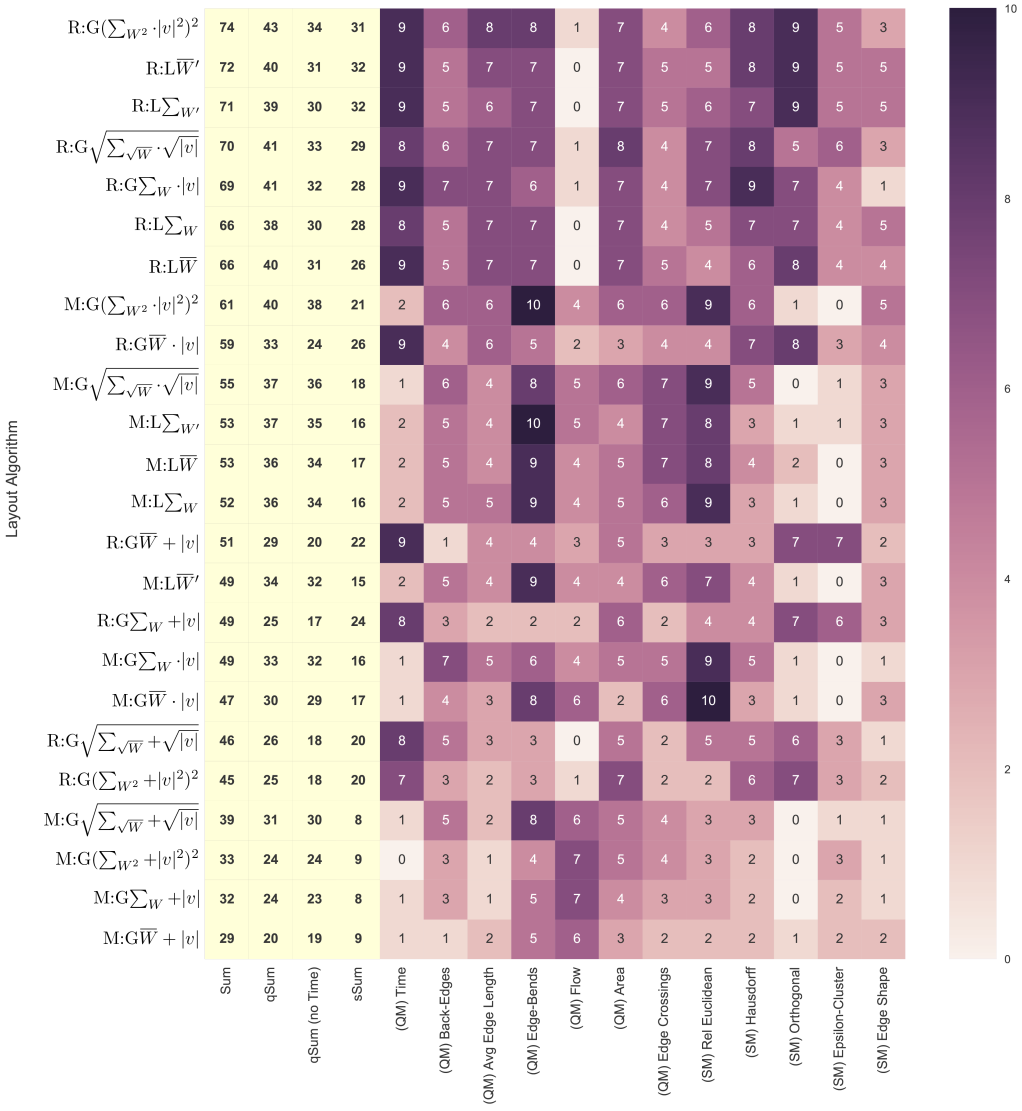

Figure 5.1: The combined statistical test results.

Chapter 6

Animation

One of the approaches towards mental map preservation, is animation [Bra01] (sometimes also called morphing). The idea is that by animating the transition from one graph layout to another, the user can 'see' which changes occur. A lot of research [BB99, SIG07, SI08, ZKS11, BPF14, AP16] finds that animation aids mental map preservation and therefore improves the performance of users on (certain) tasks. There are some issues to consider, however. For example, a user can only follow up to 5 moving elements at the same time [PS88]. In order to (partially) overcome this problem, similar to other works [FE00, FH01, FE02, FT08, LD08, RPD09, ZKS11, BPF14], we use phased animation, which is described in Section 6.1. Additionally, we consider the issue of change blindness [SL97]. Also, our edges are drawn as splines and, to the best of our knowledge, all other works that implement animation only consider straight-line edges. Therefore, we discuss spline morphing in Section 6.2. Finally, videos of our animation can be found online[1].

6.1 Phased animation

Animating in phases is done to reduce the cognitive effort of the user. More specifically, by splitting the animation into different phases, the user only has to follow a 'subset' of the changes at the same time. Our phased animation starts with fading out removed graph elements, then, graph elements are moved to their new position while color/shape properties are updated, and finally, new graph elements are faded in. A major drawback of phased animation is that it takes longer to animate than an animation in which all elements are animated at once. Therefore, it is important to determine an animation time that is neither too long nor too short. Research [BB99] suggests animation times of about 500-1000 milliseconds. The animation times of each phase can be found in Table 6.1, which we determined after exploring different animation times for each phase and fine tuning them through pilot tests. The animation phases are described in more detail in the sections below.

6.1.1 Fade Out

In the fade out phase, removed graph elements are faded out by decreasing the elements' opacity. An issue to consider, is change blindness [SL97]. By only decreasing the opacity, users may not notice all elements that fade out. Therefore, since research [APP10] has shown that using color

[1] https://robinmennens.github.io/StableGraphLayouts/

Phase	Time (milliseconds)
Fade Out	750
Move	500
Fade In	750

Table 6.1: The duration of every animation phase.

to highlight changes between two graphs increases user performance, we highlight these elements by temporarily changing their color to red. Additionally, since edges are not as salient as nodes, we temporarily increase the width of edges that fade out.

6.1.2 Move

In the move phase, nodes are moved to their new position while color and shape properties are updated. This is done in a slow-in/slow-out fashion. Additionally, edges are morphed to their new 'shape' and position, which is described in Section 6.2. Note that the duration of this phase is shorter than the other two. The main reason for this is that our stable layouts reduce node movement. Therefore, animation time can be shorter.

6.1.3 Fade In

This phase is similar to the fade out phase, except now, new graph elements are faded in. Again, in order to avert the effect of change blindness, we highlight elements that fade in by temporarily changing their color to green. And, again, since edges are not as salient as nodes, we temporarily increase the width of edges that fade in.

6.1.4 Considerations

As stated above, the animation duration is one of the drawbacks of phased animation. In order to reduce the total animation time, we skip animation phases when nothing occurs in them. For example, if no graph elements are removed, we start the animation with the move phase.

Another issue we consider is color blindness. As stated above, we use red and green for the fade out and fade in phases, respectively. The problem with these colors is that people who suffer from color blindness can not distinguish them [Won11]. Therefore, we could change red and green to colors that are distinguishable by color blind people [Won11]. However, observe that the color itself is only one of the encodings we use to emphasize elements that fade in/out. The change in color (and temporary edge width increase) also emphasize elements that fade in/out. Additionally, since fade out and fade in happen in different phases, it is still clear whether elements fade in or out. Therefore, we do not change the colors to color blind safe colors.

6.1.5 Viewport Animation

After animation, we obtain a new graph which may be of a different size than the original graph. Consequently, the new graph may not fit on the screen. To solve this problem, we also animate

the viewport of the user. More specifically, zooming and panning is done automatically such that, at the end of the animation, the new graph is centered and fits on the screen. We consider several options related to when we should animate the viewport. We could, for example, animate the viewport before fade out or after fade in. This would, however, increase the total animation time even more. Another option is to animate the viewport while all other phases are animating. We found this to be unsatisfactory, however, because in some scenarios, the graph would temporarily (partially) leave the viewport. Finally, after experimentation, we found that animating the viewport during the move phase resulted in the best behavior: the total animation time is not increased and the graph does not temporarily (partially) leave the viewport.

6.2 Spline Morphing

To the best of our knowledge, all existing work that implements graph animation considers straight line edges (which are easy to animate). Since our edges are drawn as (bezier) splines, we have to do some more work. Recall that an edge $e \in E$ consists of segments $segments(e)$, and that every segment $s \in segments(e)$ contains four points (two endpoints and two control points) that define the position and shape of the spline segment. Consequently, the spline belonging to e is defined by a sequence of points: $spline(e)$.

Given some edge e, let $spline(e)$ be the current spline and let $spline'(e)$ be the spline to which e has to be morphed. Then, there are two scenarios to consider. When $spline(e)$ and $spline'(e)$ are of the same size, we can simply linearly interpolate every point between its old and new position. When $spline(e)$ and $spline'(e)$ have a different size, we have to do some extra work. More specifically, we add extra points (that do not affect the position or shape of the spline) to $spline(e)$ or $spline'(e)$ (depending on which one is of smaller size) to make $spline(e)$ and $spline'(e)$ equal size. We can then linearly interpolate the points from their old to their new positions.

Adding new points to $spline(e)$ or $spline'(e)$ should be done such that the position and shape of the spline is not affected. A simple approach, is to duplicate segment endpoints (not their control points) in the spline. A more complex approach, such as knot-insertion [RT92], could also be used. We found, however, that our more simple approach already provides satisfying results: it is fast and only in rare cases do artifacts occur.

Chapter 7

Evaluation

In order to evaluate our algorithm, we compare it against *dot* (which is the current industry standard for visualizing processes). In Section 7.1, we describe a quantitative evaluation that was done using our test framework (see Section 5.1). After that, in Section 7.2, we describe a user study which was done with process mining experts. Finally, in Section 7.3, we shortly summarize and discuss our evaluation results.

7.1 Quantitative Evaluation

By using our test framework, we are able to quantitatively compare our algorithm and *dot*. Similar to the algorithm configuration test described in Section 5.2, we have generate 500 tests for the same datasets (see Section 5.1). Both algorithms compute layouts for the graphs in every test. Again, due to test independence and a large sample size, we can assume, according to the Central Limit Theorem, that our metric results are normally distributed (per algorithm, per dataset). Therefore, we determine, for every metric (per dataset), which algorithm performs significantly better. This is done by using a one-way t-test (significance level $\alpha = 0.05$). After running all statistical tests, we combine the test results of the different datasets. These results are shown and discussed in Section 7.1.1.

7.1.1 Results

The combined statistical test results are shown in Figure 7.1. All columns, except the first three, represent a metric (we refer to these columns as 'metric columns') while the rows represent the tested algorithms. A value in a cell in one of the metric columns represents for how many of the datasets the related algorithm performs significantly better. For example, our algorithm is significantly faster (see column *(QM) Time*) for all 13 datasets. Next to the metric columns, we have the 'aggregation columns'. The *Sum* column contains, for both algorithms, the sum of the values in the metric columns. *qSum* and *sSum* also contain the sum of the values, but only for the quality and stability metrics, respectively.

As we can see, our algorithm outperforms *dot* significantly. In terms of stability, our algorithm performs, for all 13 datasets, significantly better for every stability metric. This makes sense, because we specifically designed our algorithm to compute stable layouts. Considering this stability, we would expect that our graph layouts have lower quality than those computed by *dot*. As we can

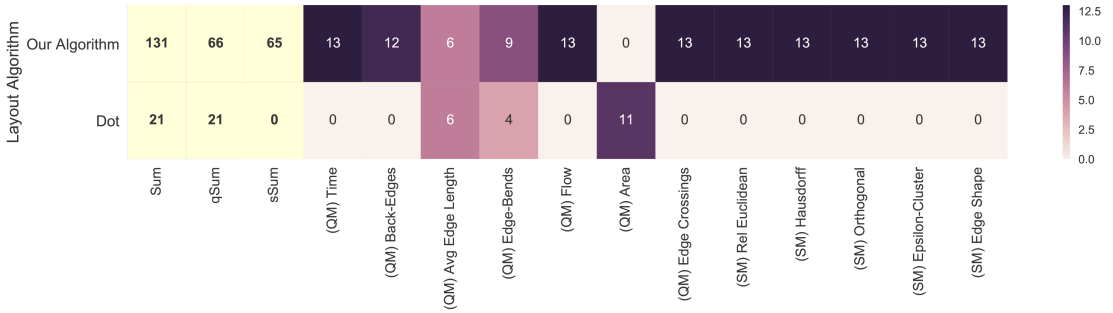

Figure 7.1: The combined statistical test results.

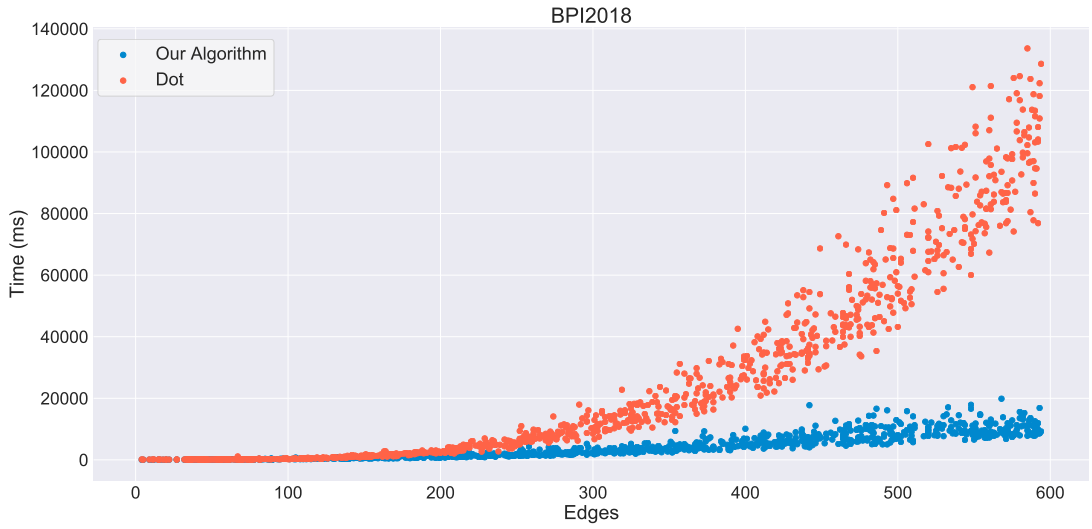

Figure 7.2: The running time results for the BPI2018 dataset.

see, however, our algorithm outperforms *dot* for most of the quality metrics. The only two metrics where *dot* performs better (or equally good) are *Average Edge Length* and *Area*. This makes sense, because *dot* is designed to keep edges short [GKNV93], and thereby compute compact layouts. Another quality metric of interest, on which our algorithm performs better, is *Edge Crossings*. Observe that, in general, our algorithm computes layouts with more edge crossings because it limits node movement. Note, however, that the *Edge Crossings* metric takes the weight of edges into account. Therefore, while *dot* generally computes layouts with fewer edge crossings, those edge crossings are between high weight (more salient and more important) edges. Our algorithm, on the other hand, generally computes layouts with more edge crossings, but these edge crossings are between low weight (less salient and less important) edges. Overall, we can state that, compared to *dot*, our algorithm computes graph layouts of high quality and high stability.

The running time results for the BPI2018 dataset are shown in Figure 7.2. As we can see, our algorithm is significantly faster than *dot*, especially for graphs with a high number (more than 250) of edges. Additionally, the increase in running time for *dot* is much more significant, suggesting a difference in running time complexity. The running time results of the other 12 tested datasets can be found in Appendix A.

7.2 Qualitative Evaluation

The results of our quantitative analysis indicate that our algorithm performs significantly better than *dot*. The quantitative analysis, however, was not done with users and is therefore a poor indication of which algorithm is preferred by actual users. Therefore, in this section, we present a user study in which 14 process mining experts participated.

The main idea of the user study is to let participants work with the process graph dashboard by asking them process-related questions about two datasets. In total, we consider three algorithms that visualize the processes in the process graph dashboard: our algorithm (named A), *dot* with animation (named B), and *dot* without animation (named C). After answering the dataset questions three times (once per algorithm), we ask the participants evaluation questions about their experience with the algorithms. The answers to these evaluation questions are then analyzed in order to determine which algorithm is preferred. Note that we consider the animated version of *dot* such that we can also test the effect of (only adding) animation.

7.2.1 Tasks

In order to define meaningful and useful dataset questions, to be answered by the participants, we first investigated tasks that are (often) carried out using the process graph dashboard. After interviewing three process mining experts, we constructed the following list of tasks:

- Filtering: filters are used to remove/add nodes/edges such that only data of interest is shown.
- Lookup: the user wants to find a node/edge.
- Revisitation: after a graph change, some node/edge needs to be found again.
- Path finding: a path from one node to another needs to be found.
- Relation seeking (similar to path finding): the user wants to discover a relation between two nodes (activities). For example, the user wants to see if activity A occurs before activity B.
- Zooming: the user zooms to focus on (a) specific graph element(s).
- Panning: the user 'moves' the graph layout to focus on (a) specific graph elements(s).
- Hover: by hovering over a graph element, the user obtains more information about that element.

In Section 7.2.3, we use this list of tasks to define the dataset questions.

7.2.2 Data Sets

In the user study, we used two datasets (which we already introduced in Chapter 5): Road and Insurance (see Table 7.1). Both datasets have been obtained from a real-life process. Also, the datasets are neither too small or too large, making them ideal for the user study.

7.2.3 Experiment Setup

In the user study, participants have to work with the process graph dashboard in order to answer 10 process-related questions about the two datasets. Participants have to answer these dataset

Name	Nodes	Edges	Cases	Variations	Description
Road	11	70	150370	171	Dataset of an information system managing road traffic fines
Insurance	13	87	99999	22149	Dataset containing process data of an insurance company

Table 7.1: The datasets used in the user study.

questions three times, each time using a different algorithm (A, B, or C). The order of the algorithms is randomized among the participants. Note that, after answering the dataset questions the first time, the participant already (thinks he/she) knows the answer to every question. Nevertheless, we ask the participants to reperform the steps taken to answer the questions. The goal is to let the participant work with every algorithm.

The dataset questions for the two datasets are listed below. All of these questions are designed with the tasks, as listed in Section 7.2.1, in mind. Table 7.2 lists, for every question, which tasks are required to answer the question. Note that zooming, panning, and hover are not included here because we deem these as 'general' tasks. In the lists below, *italic* text is used to indicate filter values while underlined text is used to indicate activity names.

Road:

- **R1:** which sequence of activities appears to be the most frequent/important?
- **R2:** are all of these most frequent/important activities executed for *Jurisdiction related* activity types?
- **R3:** are all of these most frequent/important activities executed for *Fine related* activity types?
- **R4:** for the years after *2005* (not including 2005). In which year is Payment directly followed by Payment the most?
- **R5:** which activity is executed in *2007* but not in *2008*?

Insurance:

- **I1:** which sequence of activities appears to be the most frequent/important?
- **I2:** which activities occur in *2017* but not in *2018*?
- **I3:** for which age group is Re-assessment directly followed by Payment the most?
- **I4:** at which office is Correct payment directly followed by Correct payment the most?
- **I5:** at this office, for which age group is Correct payment directly followed by Correct payment the most?

After answering all 10 dataset questions three times, the participants are asked to answer a number of evaluation questions, which are listed in Table 7.3. The *Answer* column lists the answer options provided to the participant. "Open" means that the participant can provide any answer. During the experiment, the experiment description, dataset questions, and evaluation questions are given to the participant on a form, which can be found in Appendix B.

Question	Task(s)
R1	Path finding/Relation seeking
R2	Filtering, Revisitation/Path finding
R3	Filtering, Revisitation/Path finding
R4	Lookup, Filtering, Revisitation
R5	Lookup, Filtering, Revisitation
I1	Path finding/Relation seeking
I2	Lookup, Filtering, Revisitation
I3	Relation seeking, Filtering, Revisitation
I4	Lookup, Filtering, Revisitation
I5	Lookup, Filtering, Revisitation

Table 7.2: For every user study question, we list the required tasks.

	Question	Answer
Q1	In general, which algorithm do you prefer the most?	A / B / C / No preference
Q2	Why?	Open
Q3	For which algorithm did you find the processes to be the most readable/easiest to follow/understandable?	A / B / C / No preference
Q4	Why?	Open
Q5	Your interaction with the process graph dashboard changed the process being visualized. For which algorithm was this change the easiest to follow?	A / B / C / No preference
Q6	Why?	Open
Q7	Algorithms A and B use animation. Did the animation help you answer the questions?	Yes / No / No preference
Q8	If yes, for which algorithm did the animation help the most?	A / B / No preference

Table 7.3: The user study evaluation questions. The *Answer* column lists, for every question, the possible answers a participant can give.

7.2.4 Results

In total, 14 process mining experts (13 male and 1 female) participated in the user study. All participants were already familiar with the process graph dashboard. Two of the participating males suffer from some form of color blindness. However, as discussed in Section 6.1.4, we believe color blindness does not strongly affect the understandability of our animation.

In our user test, the dataset questions are only used as a means to make the participants work with the different algorithms. Also, not all dataset questions have a single correct answer (R1, R2, and I1, for example). Therefore, we deem the (correctness of the) answers to these dataset questions as less relevant. We are more interested in the answers to the evaluation questions. Nevertheless, Appendix C contains an overview of the answers provided to the dataset questions. In general, all participants performed well with all three algorithms and managed to answer most dataset questions correctly.

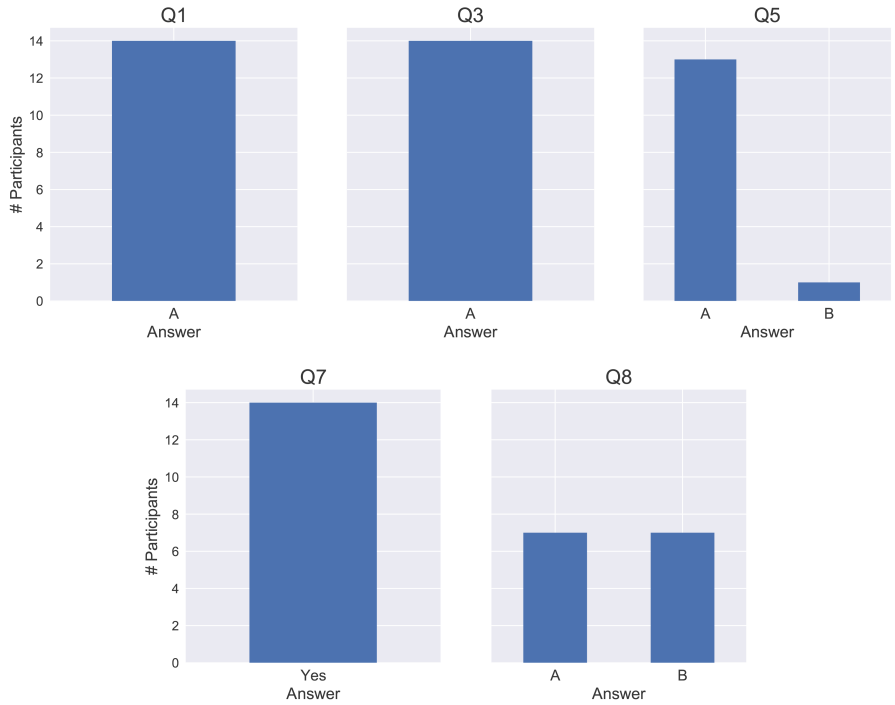

Figure 7.3: An overview of the answers given to the (non-open) user study evaluation questions.

An overview of the answers given to the evaluation questions, is provided in Figure 7.3. As we can see, our algorithm (A) performs significantly better than *dot* (B and C). For Q1 and Q3, all participants prefer our algorithm. To verify these results, we run a Chi-Squared test, which returns a p-value of 0.0001828, indicating that with a significance value of $\alpha = 0.05$, the null-hypothesis can be rejected, i.e., there is significant evidence that these results are not obtained by chance. For Q5, there is only one participant who prefers algorithm B. Again, we run a Chi-Squared test and obtain a p-value of 0.0013406, which again results in the rejection of the null-hypothesis (with a significance value of $\alpha = 0.05$). After investigation, we find that this single participant prefers algorithm B mainly due to question I2. For this question, the participants have to identify a set of activities that is present in 2017, but not in 2018. It happened (by chance), that in the layout computed by *dot* (algorithm B), this set of activities is nicely clustered together, and thereby easily identifiable.

Related to animation, all participants prefer animation over no animation (Q7) (and thus the Chi-Squared results for Q1 and Q3 also apply to Q7), which is in line with previous animation research [BB99, SIG07, ZKS11, BPF14, AP16]. As a side note, however, a single participant indicated that he/she only prefers animation for our algorithm (A). For *dot* (B), he/she found the animation to be distracting because a lot of nodes/edges are modified, i.e., the lack of stability results in distracting animations. Considering question Q8, we realized during the execution of the experiment that this question is flawed. More specifically, participants never used our stable algorithm without animation. Therefore, we do not consider the results of Q8 in more detail.

Questions Q2, Q4, and Q6 are more difficult to analyze because they are open questions. Nevertheless, answers to these questions indicate that, in general, participants prefer our algorithm because it computes graph layouts where the 'main path' is positioned in the center and because edges are more straight and therefore easier to follow. Additionally, participants state that the stability and animation combination makes it easy to follow changes. All in all, given these results, we can conclude that users prefer our algorithm over *dot*.

7.3 Conclusion & Discussion

In terms of layout quality, the evaluation results indicate that the graph layouts computed by our algorithm are more readable and understandable than the graph layouts computed by *dot*. In the quantitative evaluation, our algorithm performs better than *dot* for all quality metrics except for the *Area* and *Average Edge Length* quality metrics. Hence, intuitively, *dot* computes layouts that are more compact. This compactness, however, does not benefit the proper representation of the underlying process. The qualitative evaluation results indicate the same: all participants indicate that they find the layouts computed by our algorithm more readable/understandable.

Considering graph layout stability, which we use to improve mental map preservation, the evaluation results indicate that the graph layouts computed by our algorithm ensure more graph layout stability. In the quantitative evaluation, our algorithm performs better than *dot* for all stability metrics. Similarly, in the qualitative evaluation, 13 out of the 14 participants indicate that the changes between graph layouts are easier to follow when using our algorithm. Also related to mental map preservation, we find that animation on its own already appears to aid mental map preservation significantly. All participants in the user study indicate that they prefer using animation over not using animation.

In terms of performance, our quantitative evaluation results indicate that our algorithm is significantly faster than *dot*, especially for graphs with more than (about) 250 edges. Consequently, our algorithm is more usable in the sense that it provides a more responsive interaction, i.e., a user has to wait significantly shorter when he/she wants to visualize a graph with more than 250 edges.

All in all, both the quantitative and qualitative evaluation results show that, in terms of layout quality, mental map preservation, and running time, our algorithm performs significantly better than *dot*.

Chapter 8

Discussion

In this chapter, we discuss the pros and cons of our approach. In Section 8.1, we discuss our approach related to layout quality. After that, in Section 8.2, we discuss our approach related to preserving the mental map of the user, i.e., we discuss graph layout stability and animation.

8.1 Layout Quality

In terms of layout quality, we want graph layouts to be readable and understandable. In our approach, we achieve this goal by computing a global ranking and global order based on the process data. By using these in the graph layout computation, we make sure the computed graph layouts better represent the actual process. In Figure 8.1, for example, we have two identical graphs. The graph layout on the left is computed by *dot* and the graph layout on the right is computed by our algorithm. In terms of readability, we deem our graph layout to be much more readable: edges are more straight and there are fewer edge crossings. Also, we deem our graph layout to be more understandable: the 'main path' (most frequent behaviour) is nicely centered and aligned. Additionally, the taller layout makes sure there is (mostly) a (more understandable) downward 'flow' of the edges/process. On the other hand, the layout computed by *dot* is much wider and has a (less understandable) 'flow' that goes both up and down. The main reason for this difference in layout is that one of the goals of *dot* is to compute layouts that are as compact as possible. This compactness, however, often results in layouts that have poor readability and understandability (with respect to the process). In Appendix D, more graph layout comparisons can be found.

Using a global ranking and global order also has its downsides. Mainly because the global ranking and global order are computed based on the complete process data, and not based on the filtered data, some issues can occur. For example, the global ranking essentially enforces nodes to be on a certain rank, which is not always optimal for every graph. Similarly, the global order enforces the nodes (and some of the edges) to be in a specific order. Obviously, this order is not always optimal for every graph layout. Consider, for example, the two graph layouts in Figure 8.2. We have the same two graphs where the graph layout on the left is computed by *dot* and the graph layout on the right is computed by our algorithm. As we can see, in our layout, node *Post-process invoice* is not placed optimally because we use the global ranking. More specifically, due to the global ranking, *Post-process invoice* needs to be placed below the rank of *Checked and approved* and *Process employee reimbursement*, and above the rank of *Final check of invoice*. Consequently, an unnecessary extra rank is created. Additionally, the global order enforces *Post-process invoice*

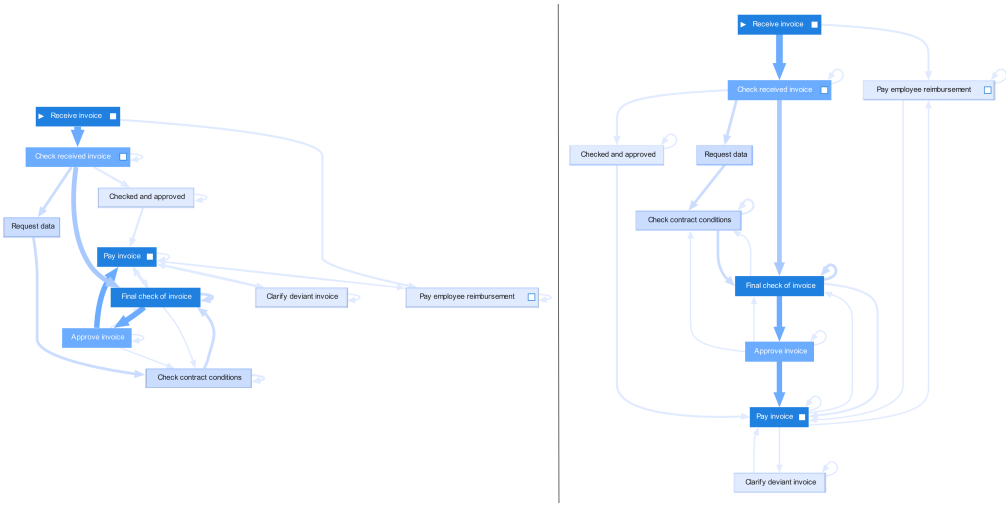

Figure 8.1: Illustrated are two identical graphs. The left graph layout is computed by *dot* while the right graph layout is computed by our algorithm. We prefer the right layout, because it is much more readable and understandable.

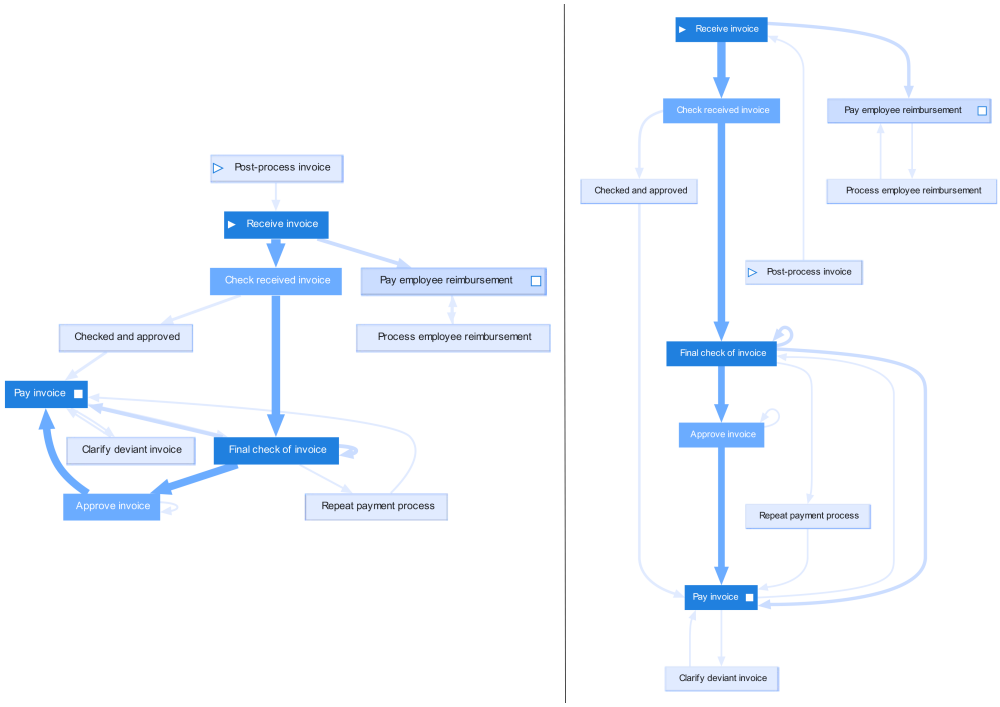

Figure 8.2: Illustrated are two identical graphs. The left graph layout has been computed by *dot* while the right graph layout has been computed by our algorithm. As we can see, the graph layout computed by our algorithm is taller than strictly necessary. See text for the explanation.

to be right of the edge (*Check received invoice*, *Final check of invoice*). Placing it left of this edge, would have resulted in a more balanced layout. Nevertheless, considering both graph layouts, we still deem our layout to be better than the one computed by *dot*. Note that in this case, since the graph is not too large, this problem is not disturbing. For larger graphs, however, this problem can be quite significant. In future work, it would be interesting to investigate this problem.

8.2 The Mental Map

One of the approaches we take towards preserving the mental map, is to ensure graph layout stability. In this work, we aim to satisfy our mental map definition (see Chapter 3 and [MELS95]), i.e., create graph layout stability, by using the global ranking and global order. The global ranking constrains the vertical order of the nodes, and thereby reduces the possible change between two layouts. Also, since the global ranking is computed based on the process data, it makes sure computed graph layouts properly show the actual process. A disadvantage, which we already discussed in Section 8.1, however, is that using the global ranking can result in unnecessary extra ranks. The global order, on the other hand, constrains the horizontal movement of nodes, further reducing possible layout change. An obvious disadvantage, is that the constraints can result in extra edge crossings. We partially solve this problem by only constraining nodes and sequence edges. Non-sequence edges can be positioned freely, allowing the graph layout algorithm to reduce edge crossings. Another potential issue, relates to our assumption that a process always has a single main/most frequent path, i.e., has only one backbone. In practice, this is often true, but it may not hold for every process. In future work, it would be interesting to investigate whether or not this is actually a problem.

The second approach we take towards preserving the mental map is to transition between two consecutive graph layouts with animation. An obvious disadvantage of animation is that it takes time. On the other hand, the animation helps the user follow the changes between two graph layouts. In our user study, for example, all participants indicate that they prefer animation over no animation. Since illustrating animation and graph layout stability by using images is not practical, we provide videos, which can be found online[1]. In these videos, to illustrate the effect of animation, we show *dot* both with and without animation.

[1] `https://robinmennens.github.io/StableGraphLayouts/`

Chapter 9

Conclusions

With process mining [Aal16], organizations are moving towards a fact-based approach where they can make data-driven decisions based on facts. Visualization of the process, which allows organizations to gain insight in their internal/external process(es), plays an important role in process mining. The current industry standard used for process visualization, is *dot* [GKN15], a Sugiyama [STT81] based hierarchical graph layout algorithm which is part of the open source, free to use, Graphviz [1] software package. *Dot*, however, often fails to satisfy two important aspects of process visualization. Firstly, graph layouts should be of high quality: the layouts should be readable and understandable, and thereby properly show the underlying process. Secondly, the mental map [MELS95, CP96] of the user should be preserved if the graph changes due to filtering. In general, there are two approaches towards preserving the mental map of the user [Bra01]. The first is to minimize layout change between two consecutive layouts, i.e., to ensure graph layout stability [AP13]. The second is to transition between two consecutive graph layouts, which allows the user to follow the layout changes.

In order to compute graph layouts that preserve the mental map and are of high quality, we bring the areas of process mining and graph drawing together. More specifically, we provide three main contributions. The first is a novel ranking algorithm that computes a global ranking based on the process data. By using the global ranking during graph layout computation, we obtain layouts that properly represent the actual process. Also, since the global ranking essentially constrains the vertical order of nodes, we minimize the change between layouts. The second contribution is a constraint computation algorithm that computes a global order based on the process data. The global order constrains the horizontal movement of nodes and is used in every graph layout computation by our third contribution: a crossing minimization algorithm that makes sure the global order constraints are satisfied. Additionally, to make it easier for a user to follow the changes between two graph layouts, we use a phased animation technique to transition from one graph layout to another, further improving mental map preservation.

In this work, we evaluate our contributions using a quantitative and qualitative evaluation. The quantitative evaluation shows that our approach, compared to *dot*, produces graph layouts of high quality and high stability. Additionally, our algorithm is significantly faster than *dot*, especially for graphs with a high number of edges. In the qualitative evaluation, a user study done with 14 process mining experts, we find similar results. All 14 participants indicate that they prefer our approach over *dot*. Additionally, the participants indicate that they prefer animation over no animation. Given these results, it can be concluded that our approach provides a fast, usable, and effective method to visualize process data.

9.1 Future work

In this section, we list possible directions for future work.

Constrained Crossing Minimization
In Chapter 4, we introduced two constrained crossing minimization techniques: an existing technique, to which we refer as MINCROSS, and a novel technique, called RELMINCROSS. Next to these techniques, other techniques [Fin01, For04] exist as well. It would be interesting to see how these perform. Additionally, in Chapter 5, we essentially concluded that MINCROSS generally produces layouts of high quality, but of lower stability. RELMINCROSS, on the other hand, generally produces layouts of lower quality, but of high stability. It would be interesting to investigate a combination of both algorithms. We could, for example, use the (more stable) order initialization of RELMINCROSS and then use the median sorting and transpose techniques of MINCROSS to improve layout quality.

Backbone
In this work, we essentially assume that there is only a single backbone, i.e., there is one main/most frequent path. In practice, this is often true, but it does not hold for every process. Therefore, it would be interesting to investigate the detection of multiple backbones, i.e., to detect other frequent/important behaviour in a process. This information could then be used to further improve the computed graph layouts.

Global Ranking Relaxation
Another issue, as discussed in Chapter 8, relates to unnecessary ranks being created because we enforce the global ranking. It would be interesting to investigate a similar technique as used by Görg et al. [GBPD04], in which node constraints (in our case the global ranking) are 'relaxed' in some scenarios, i.e., we do not enforce all nodes to be on their global rank. By doing so, we could prevent the creation of these unnecessary ranks, resulting in a layout quality improvement. On the other hand, such a technique would reduce layout stability.

Dynamic Animation Times
Research [SIG07] finds evidence for a correlation between preferred animation time and task complexity. More specifically, when tasks are more complex, users prefer a longer animation time. Given this research, it would be interesting to look into dynamic animation times. We could, for example, try to determine task complexity (or the amount of change) and base the animation time on this complexity.

Bibliography

[1] Graphviz - graph visualization software. http://www.graphviz.org/. Accessed: 2018-06-04. 16, 79

[2] BPI2012 - second international business process intelligence challenge. http://www.win.tue.nl/bpi/doku.php?id=2012:challenge. Accessed: 2018-08-09. 58

[3] BPI2013 - third international business process intelligence challenge. http://www.win.tue.nl/bpi/doku.php?id=2013:challenge. Accessed: 2018-08-09. 58

[4] BPI2014 - fourth international business process intelligence challenge. http://www.win.tue.nl/bpi/doku.php?id=2014:challenge. Accessed: 2018-08-09. 58

[5] BPI2017 - seventh international business process intelligence challenge. https://www.win.tue.nl/bpi/doku.php?id=2017:challenge. Accessed: 2018-08-09. 58

[6] BPI2018 - eighth international business process intelligence challenge. https://www.win.tue.nl/bpi/doku.php?id=2018:challenge. Accessed: 2018-08-09. 58

[AAA+07] A. Alves, A. Arkin, S. Askary, C. Baretto, B. Bloch, F. Cubera, M. Ford, Y. Goland, A. Guizar, N. Kartha, C. Liu, R. Khalaf, D. Konig, M. Marin, V. Mehta, S. Thatte, D. van der Rijn, P. Yendluri, and A. Yiu. Web Services Business Process Execution Language, April 2007. 11

[Aal16] Wil M. P. van der Aalst. *Process Mining: Data Science in Action*. Springer, April 2016. Google-Books-ID: hUEGDAAAQBAJ. 1, 2, 79

[AEHK10] Benjamin Albrecht, Philip Effinger, Markus Held, and Michael Kaufmann. An Automatic Layout Algorithm for BPEL Processes. In *Proceedings of the 5th International Symposium on Software Visualization*, SOFTVIS '10, pages 173–182, New York, NY, USA, 2010. ACM. 22, 28, 33, 46

[AP12a] D. Archambault and H. C. Purchase. The mental map and memorability in dynamic graphs. In *2012 IEEE Pacific Visualization Symposium*, pages 89–96, February 2012. 19, 20

[AP12b] Daniel Archambault and Helen C. Purchase. Mental Map Preservation Helps User Orientation in Dynamic Graphs. In *Graph Drawing*, Lecture Notes in Computer Science, pages 475–486. Springer, Berlin, Heidelberg, September 2012. 7, 19, 20

[AP13] Daniel Archambault and Helen C. Purchase. The Map in the mental map: Experimental results in dynamic graph drawing. *International Journal of Human-Computer Studies*, 71(11):1044–1055, November 2013. 1, 4, 79

[AP16] Daniel Archambault and Helen C. Purchase. Can animation support the visualisation of dynamic graphs? *Information Sciences*, 330:495–509, February 2016. 7, 63, 72

[APP10] Daniel Archambault, Helen C. Purchase, and Bruno Pinaud. Difference Map Readability for Dynamic Graphs. In *Graph Drawing*, Lecture Notes in Computer Science, pages 50–61. Springer, Berlin, Heidelberg, September 2010. 6, 63

[APP11] D. Archambault, H. Purchase, and B. Pinaud. Animation, Small Multiples, and the Effect of Mental Map Preservation in Dynamic Graphs. *IEEE Transactions on Visualization and Computer Graphics*, 17(4):539–552, April 2011. 6, 7, 19, 20

[BB99] B. B. Bederson and A. Boltman. Does animation help users build mental maps of spatial information? In *1999 IEEE Symposium on Information Visualization, 1999. (Info Vis '99) Proceedings*, pages 28–35, 1999. 6, 63, 72

[BBDW17] Fabian Beck, Michael Burch, Stephan Diehl, and Daniel Weiskopf. A Taxonomy and Survey of Dynamic Graph Visualization. *Computer Graphics Forum*, 36(1):133–159, January 2017. 7, 15

[BP90] Karl-Friedrich Bhringer and Frances Newbery Paulisch. Using Constraints to Achieve Stability in Automatic Graph Layout Algorithms. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '90, pages 43–51, New York, NY, USA, 1990. ACM. 8

[BPF14] B. Bach, E. Pietriga, and J. D. Fekete. GraphDiaries: Animated Transitions andTemporal Navigation for Dynamic Networks. *IEEE Transactions on Visualization and Computer Graphics*, 20(5):740–754, May 2014. 11, 63, 72

[Bra01] Jürgen Branke. Dynamic Graph Drawing. In *Drawing Graphs*, Lecture Notes in Computer Science, pages 228–246. Springer, Berlin, Heidelberg, 2001. 1, 4, 6, 7, 11, 63, 79

[BS08] Michael Baur and Thomas Schank. Dynamic Graph Drawing in Visone. Technical report, Karlsruhe : Universitat Karlsruhe, Fakultat fur Informatik, 2008, 2008. 10

[BS15] Vered Bernstein and Pnina Soffer. Identifying and Quantifying Visual Layout Features of Business Process Models. In *Enterprise, Business-Process and Information Systems Modeling*, Lecture Notes in Business Information Processing, pages 200–213. Springer, Cham, June 2015. 22, 24

[BT98] Stina Bridgeman and Roberto Tamassia. Difference Metrics for Interactive Orthogonal Graph Drawing Algorithms. In *Graph Drawing*, Lecture Notes in Computer Science, pages 57–71. Springer, Berlin, Heidelberg, August 1998. 24, 26, 27, 28

[BT00] Stina Bridgeman and Roberto Tamassia. A User Study in Similarity Measures for Graph Drawing. In *Graph Drawing*, Lecture Notes in Computer Science, pages 19–30. Springer, Berlin, Heidelberg, September 2000. 26, 27

[BW97] Ulrik Brandes and Dorothea Wagner. A bayesian paradigm for dynamic graph layout. In *Graph Drawing*, Lecture Notes in Computer Science, pages 236–247. Springer, Berlin, Heidelberg, September 1997. 10, 12

[CDBTT95] R. Cohen, G. Di Battista, R. Tamassia, and I. Tollis. Dynamic Graph Drawings: Trees, Series-Parallel Digraphs, and Planar ST-Digraphs. *SIAM Journal on Computing*, 24(5):970–1001, October 1995. 9

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. The MIT Press, Cambridge, Mass, 3rd edition edition, July 2009. 42

[CP96] Michael K. Coleman and D. Stott Parker. Aesthetics-based Graph Layout for Human Consumption. *Softw. Pract. Exper.*, 26(12):1415–1438, December 1996. 1, 4, 20, 21, 79

[CT12] Michele Chinosi and Alberto Trombetta. BPMN: An introduction to the standard. *Computer Standards & Interfaces*, 34(1):124–134, January 2012. 11

[DDK+02] Gilles Diguglielmo, Eric Durocher, Philippe Kaplan, Georg Sander, and Adrian Vasiliu. Graph Layout for Workflow Applications with ILOG JViews. In *Graph Drawing*, Lecture Notes in Computer Science, pages 362–363. Springer, Berlin, Heidelberg, August 2002. 12, 22

[DG02] Stephan Diehl and Carsten Grg. Graphs, They Are Changing. In *Graph Drawing*, Lecture Notes in Computer Science, pages 23–31. Springer, Berlin, Heidelberg, August 2002. 9, 10, 21, 26, 27

[DGK01] Stephan Diehl, Carsten Grg, and Andreas Kerren. Preserving the Mental Map using Foresighted Layout. In *In Proceedings of Joint Eurographics Ieee Tcvg Symposium on Visualization Vissym01*, pages 175–184. Springer Verlag, 2001. 9, 10

[EHK+03] Cesim Erten, Philip J. Harding, Stephen G. Kobourov, Kevin Wampler, and Gary Yee. GraphAEL: Graph Animations with Evolving Layouts. In *Graph Drawing*, Lecture Notes in Computer Science, pages 98–110. Springer, Berlin, Heidelberg, September 2003. 9, 10, 19

[ELMS91] Peter Eades, Wei Lai, Kazuo Misue, and Kozo Sugiyama. Preserving the mental map of a diagram. Technical report, International Institute for Advanced Study of Social Information Science, 1991. 27

[EMW86] P. Eades, B.D. McKay, and N. C. Wormald. On an edge crossing problem. In *Proc. 9th Australian Computer Science Conf.*, pages 327–334, 1986. 51

[ESK09] P. Effinger, M. Siebenhaller, and M. Kaufmann. An Interactive Layout Tool for BPMN. In *2009 IEEE Conference on Commerce and Enterprise Computing*, pages 399–406, July 2009. 12, 22

[FE00] Carsten Friedrich and Peter Eades. The Marey Graph Animation Tool Demo. In *Graph Drawing*, Lecture Notes in Computer Science, pages 396–406. Springer, Berlin, Heidelberg, September 2000. 11, 63

[FE02] Carsten Friedrich and Peter Eades. Graph drawing in motion. *Journal of Graph Algorithms and Applications, Volume 6*, 2002. 11, 63

[FH01] Carsten Friedrich and Michael E. Houle. Graph Drawing in Motion II. In *Graph Drawing*, Lecture Notes in Computer Science, pages 220–231. Springer, Berlin, Heidelberg, September 2001. 11, 63

[Fin01] Irene Finocchi. Layered Drawings of Graphs with Crossing Constraints. In Jie Wang, editor, *Computing and Combinatorics*, Lecture Notes in Computer Science, pages 357–367. Springer Berlin Heidelberg, 2001. 80

[For04] Michael Forster. A Fast and Simple Heuristic for Constrained Two-Level Crossing Reduction. In *Graph Drawing*, Lecture Notes in Computer Science, pages 206–216. Springer, Berlin, Heidelberg, September 2004. 52, 80

[FQ11] Michael Farrugia and Aaron Quigley. Effective Temporal Graph Layout: A Comparative Study of Animation Versus Static Display Methods. *Information Visualization*, 10(1):47–64, January 2011. 6, 7

[FR91] Fruchterman Thomas M. J. and Reingold Edward M. Graph drawing by forcedirected placement. *Software: Practice and Experience*, 21(11):1129–1164, 1991. 9

[FT08] Y. Frishman and A. Tal. Online Dynamic Graph Drawing. *IEEE Transactions on Visualization and Computer Graphics*, 14(4):727–740, July 2008. 9, 11, 63

[FWSL12] K. C. Feng, C. Wang, H. W. Shen, and T. Y. Lee. Coherent Time-Varying Graph Drawing with Multifocus+Context Interaction. *IEEE Transactions on Visualization and Computer Graphics*, 18(8):1330–1342, August 2012. 10

[GBPD04] Carsten Grg, Peter Birke, Mathias Pohl, and Stephan Diehl. Dynamic Graph Drawing of Sequences of Orthogonal and Hierarchical Graphs. In *Graph Drawing*, Lecture Notes in Computer Science, pages 228–238. Springer, Berlin, Heidelberg, September 2004. 9, 10, 20, 80

[GKN15] Emden R. Gansner, Eleftherios Koutsofios, and Stephen North. Drawing graphs with dot. Technical report, Graphviz, January 2015. 16, 52, 79

[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, March 1993. 16, 17, 22, 23, 52, 53, 68

[GNV88] E. R. Gansner, S. C. North, and K. P. Vo. DAGa program that draws directed graphs. *Software: Practice and Experience*, 18(11):1047–1062, November 1988. 16

[GPZ+14] Thomas Gschwind, Jakob Pinggera, Stefan Zugal, Hajo A. Reijers, and Barbara Weber. A linear time layout algorithm for business process models. *Journal of Visual Languages & Computing*, 25(2):117–132, April 2014. 22

[HEHL13] Weidong Huang, Peter Eades, Seok-Hee Hong, and Chun-Cheng Lin. Improving multiple aesthetics produces better graph drawings. *Journal of Visual Languages & Computing*, 24(4):262–272, August 2013. 22

[HM98] Weiqing He and Kim Marriott. Constrained Graph Layout. *Constraints*, 3(4):289–314, October 1998. 8

[Kit94] Robert M. Kitchin. Cognitive maps: What are they and why study them? *Journal of Environmental Psychology*, 14(1):1–19, March 1994. 4

[KK89] Tomihisa Kamada and Satoru Kawai. An algorithm for drawing general undirected graphs. *Information Processing Letters*, 31(1):7–15, April 1989. 10

[KKRM11] Sonja Kabicher, Simone Kriglstein, and Stefanie Rinderle-Ma. Visual Change Tracking for Business Process Models. In *Conceptual Modeling ER 2011*, Lecture Notes in Computer Science, pages 504–513. Springer, Berlin, Heidelberg, October 2011. 12, 22

[LD08] Elose Loubier and Bernard Dousset. Temporal and relational data representation by graph morphing. In *Safety and Reliability for Managing Risk*, volume 14. CRC Press, January 2008. 11, 63

[Lev66] V. I. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707, February 1966. 27

[LLY06] Yi-Yi Lee, Chun-Cheng Lin, and Hsu-Chun Yen. Mental Map Preserving Graph Drawing Using Simulated Annealing. In *Proceedings of the 2006 Asia-Pacific Symposium on Information Visualisation - Volume 60*, APVis '06, pages 179–188, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc. 10

[LMR98] Kelly Lyons, Henk Meijer, and David Rappaport. Algorithms for Cluster Busting in Anchored Graph Drawing. *Journal of Graph Algorithms and Applications*, 2(1):1–24, 1998. 26, 27

[MELS95] Kazuo Misue, Peter Eades, Wei Lai, and Kozo Sugiyama. Layout Adjustment and the Mental Map. *Journal of Visual Languages & Computing*, 6(2):183–210, June 1995. 1, 4, 21, 26, 27, 31, 44, 77, 79

[MHT93] K. Miriyala, S. W. Hornick, and R. Tamassia. An incremental approach to aesthetic graph layout. In *Proceedings of 6th International Workshop on Computer-Aided Software Engineering*, pages 297–308, July 1993. 8, 9

[MSW17] R.J.P. Mennens, R. Scheepens, and M.A. Westenberg. The implementation of a sugiyama layout algorithm. Unpublished, 2017. 55

[Nor95] Stephen C. North. Incremental layout in DynaDAG. In *Graph Drawing*, Lecture Notes in Computer Science, pages 409–418. Springer, Berlin, Heidelberg, September 1995. 8, 9

[NW01] Stephen C. North and Gordon Woodhull. Online Hierarchical Graph Drawing. In *Graph Drawing*, Lecture Notes in Computer Science, pages 232–246. Springer, Berlin, Heidelberg, September 2001. 8, 9

[Pau93] Frances Newbery Paulisch. *The Design of an Extendible Graph Editor*. Springer-Verlag, Berlin, Heidelberg, 1993. 8, 22

[PB08] Mathias Pohl and Peter Birke. Interactive Exploration of Large Dynamic Networks. In *Visual Information Systems. Web-Based Visual Information Search and Management*, Lecture Notes in Computer Science, pages 56–67. Springer, Berlin, Heidelberg, September 2008. 9, 10

[PCA02] Helen C. Purchase, David Carrington, and Jo-Anne Allder. Empirical Evaluation of Aesthetics-based Graph Layout. *Empirical Software Engineering*, 7(3):233–255, September 2002. 22

[PCJ97] H. C. Purchase, R. F. Cohen, and M. I. James. An Experimental Study of the Basis for Graph Drawing Algorithms. *J. Exp. Algorithmics*, 2, January 1997. 22, 23, 45

[PHG06] Helen C. Purchase, Eve Hoggan, and Carsten Grg. How Important Is the Mental Map? An Empirical Investigation of a Dynamic Graph Layout Algorithm. In *Graph Drawing*, Lecture Notes in Computer Science, pages 184–195. Springer, Berlin, Heidelberg, September 2006. 20, 21

[PKL04] Bruno Pinaud, Pascale Kuntz, and Rmi Lehn. Dynamic Graph Drawing with a Hybridized Genetic Algorithm. In *Adaptive Computing in Design and Manufacture VI*, pages 365–375. Springer, London, 2004. 10

[PS88] Zenon W. Pylyshyn and Ron W. Storm. Tracking multiple independent targets: Evidence for a parallel tracking mechanism*. *Spatial Vision*, 3(3):179–197, January 1988. 4, 63

[PS08] Helen C. Purchase and Amanjit Samra. Extremes Are Better: Investigating Mental Map Preservation in Dynamic Graphs. In *Diagrammatic Representation and Inference*, Lecture Notes in Computer Science, pages 60–73. Springer, Berlin, Heidelberg, September 2008. 19, 20

[PT90] F. Newberry Paulisch and W. F. Tichy. EDGE: An Extendable Graph Editor. *Softw. Pract. Exper.*, 20(S1):63–88, June 1990. 8

[Pur00] H. C. Purchase. Effective information visualisation: a study of graph drawing aesthetics and algorithms. *Interacting with Computers*, 13(2):147–162, December 2000. 22, 23, 45

[Pur02] HELEN C. Purchase. Metrics for Graph Drawing Aesthetics. *Journal of Visual Languages & Computing*, 13(5):501–516, October 2002. 22

[RBRB06] S. B. Rinderle, R. Bobrik, M. U. Reichert, and T. Bauer. Business Process Visualization - Use Cases, Challenges, Solutions. In *Proceedings of the Eighth International Conference on Enterprise Information Systems (ICEIS'06): Information System Analysis and Specification*. INSTICC PRESS, May 2006. 22, 28, 33, 46

[RM13] S. Rufiange and M. J. McGuffin. DiffAni: Visualizing Dynamic Graphs with a Hybrid of Difference Maps and Animation. *IEEE Transactions on Visualization and Computer Graphics*, 19(12):2556–2565, December 2013. 7

[RPD09] F. Reitz, M. Pohl, and S. Diehl. Focused Animation of Dynamic Compound Graphs. In *2009 13th International Conference Information Visualisation*, pages 679–684, July 2009. 9, 10, 11, 63

[RT92] Ronald N. Goldman and Tom Lyche, editors. *Knot Insertion and Deletion Algorithms for B-Spline Curves and Surfaces*. Other Titles in Applied Mathematics. Society for Industrial and Applied Mathematics, January 1992. 65

[San98] Georg Sander. Visualisierungstechniken für den Compilerbau. In *Ausgezeichnete Informatikdissertationen 1996*, GI-Dissertationspreis, pages 114–127. Vieweg+Teubner Verlag, Wiesbaden, 1998. 52

[SI08] Maruthappan Shanmugasundaram and Pourang Irani. The Effect of Animated Transitions in Zooming Interfaces. In *Proceedings of the Working Conference on Advanced Visual Interfaces*, AVI '08, pages 396–399, New York, NY, USA, 2008. ACM. 63

[SIG07] Maruthappan Shanmugasundaram, Pourang Irani, and Carl Gutwin. Can Smooth View Transitions Facilitate Perceptual Constancy in Node-link Diagrams? In *Proceedings of Graphics Interface 2007*, GI '07, pages 71–78, New York, NY, USA, 2007. ACM. 63, 72, 80

[SL97] Daniel J. Simons and Daniel T. Levin. Change blindness. *Trends in Cognitive Sciences*, 1(7):261–267, October 1997. 63

[SP08] Peter Saffrey and Helen Purchase. The "Mental Map" Versus "Static Aesthetic" Compromise in Dynamic Graphs: A User Study. In *Proceedings of the Ninth Conference on Australasian User Interface - Volume 76*, AUIC '08, pages 85–93, Darlinghurst, Australia, Australia, 2008. Australian Computer Society, Inc. 9, 20, 21

[ST01] Janet M. Six and Ioannis G. Tollis. Automated Visualization of Process Diagrams. In *Graph Drawing*, Lecture Notes in Computer Science, pages 45–59. Springer, Berlin, Heidelberg, September 2001. 22

[STT81] K. Sugiyama, S. Tagawa, and M. Toda. Methods for Visual Understanding of Hierarchical System Structures. *IEEE Transactions on Systems, Man, and Cybernetics*, 11(2):109–125, February 1981. 1, 5, 15, 31, 79

[SV01] Georg Sander and Adrian Vasiliu. The ILOG JViews Graph Layout Module. In *Graph Drawing*, Lecture Notes in Computer Science, pages 438–439. Springer, Berlin, Heidelberg, September 2001. 12

[Wad00] Vance Waddle. Graph Layout for Displaying Data Structures. In *Graph Drawing*, Lecture Notes in Computer Science, pages 241–252. Springer, Berlin, Heidelberg, September 2000. 8, 52

[Won11] Bang Wong. Color blindness. *Nature Methods*, 8(6):441, June 2011. 64

[YLS+04] Yun Yang, Wei Lai, Jun Shen, Xiaodi Huang, Jun Yan, and Lukman Setiawan. Effective Visualisation of Workflow Enactment. In *Advanced Web Technologies and Applications*, Lecture Notes in Computer Science, pages 794–803. Springer, Berlin, Heidelberg, April 2004. 11, 12, 22

[ZKS11] Loutfouz Zaman, Ashish Kalra, and Wolfgang Stuerzlinger. The Effect of Animation, Dual View, Difference Layers, and Relative Re-layout in Hierarchical Diagram Differencing. In *Proceedings of Graphics Interface 2011*, GI '11, pages 183–190, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 2011. Canadian Human-Computer Communications Society. 7, 11, 20, 21, 63, 72

Appendix A

Running Time Results

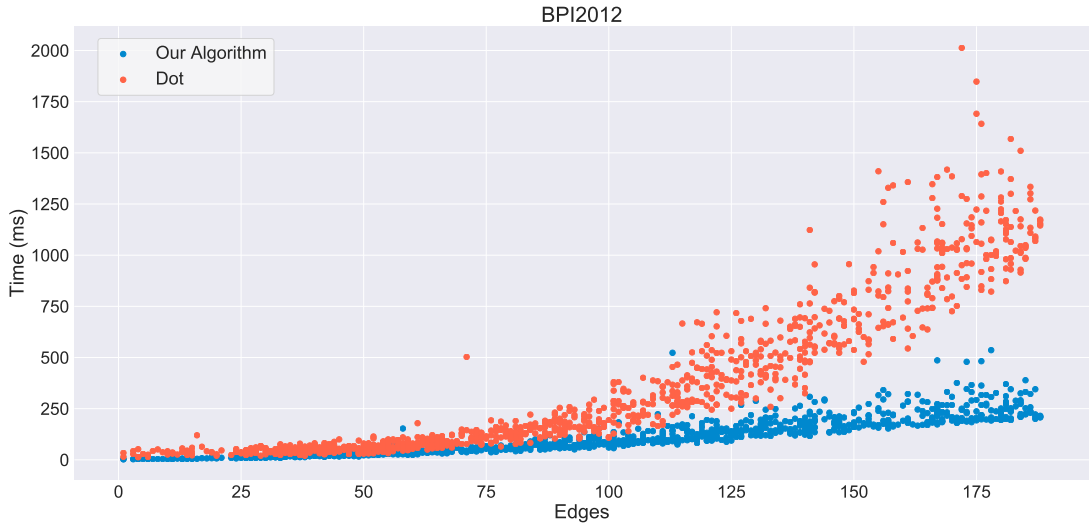

Figure A.1: The running time results for the BPI2012 dataset.

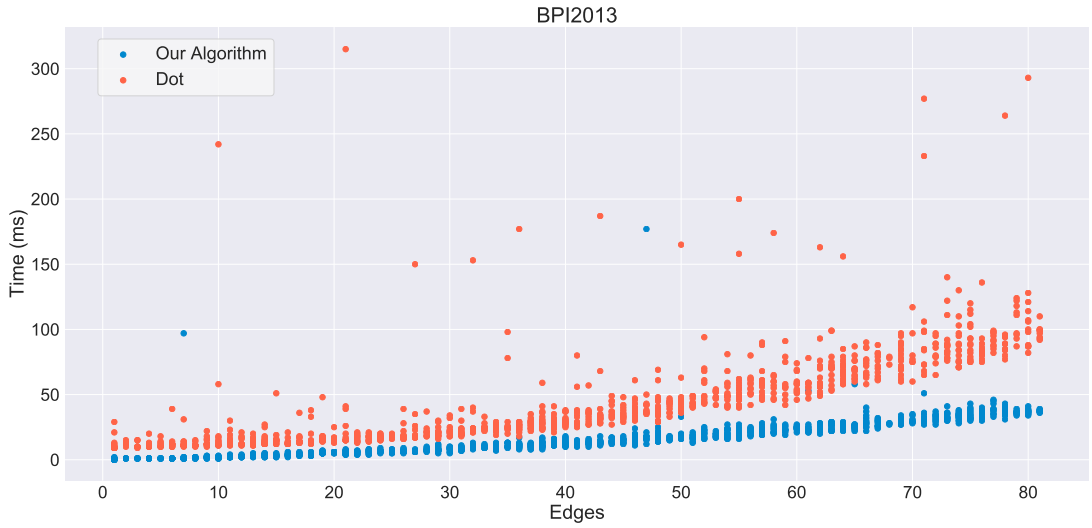

Figure A.2: The running time results for the BPI2013 dataset.

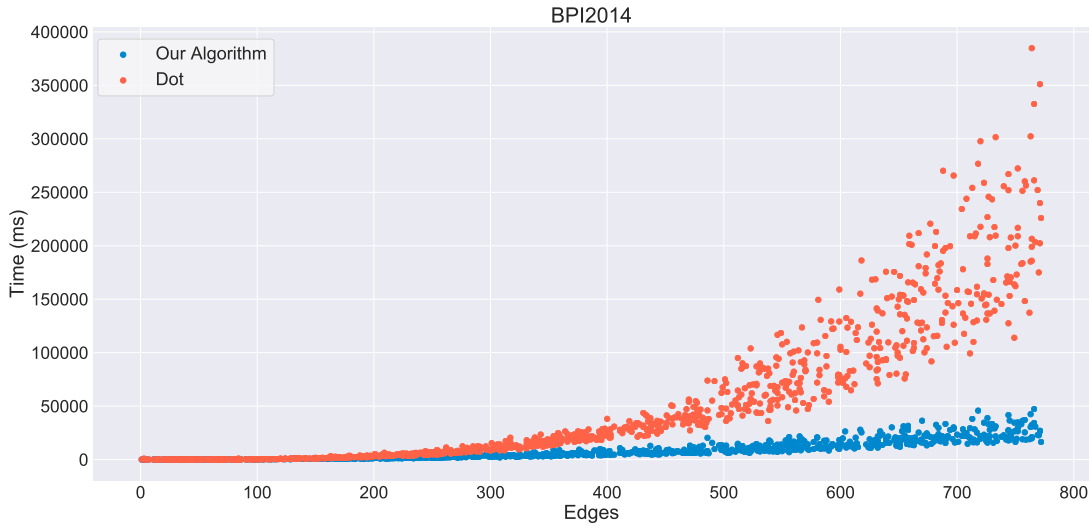

Figure A.3: The running time results for the BPI2014 dataset.

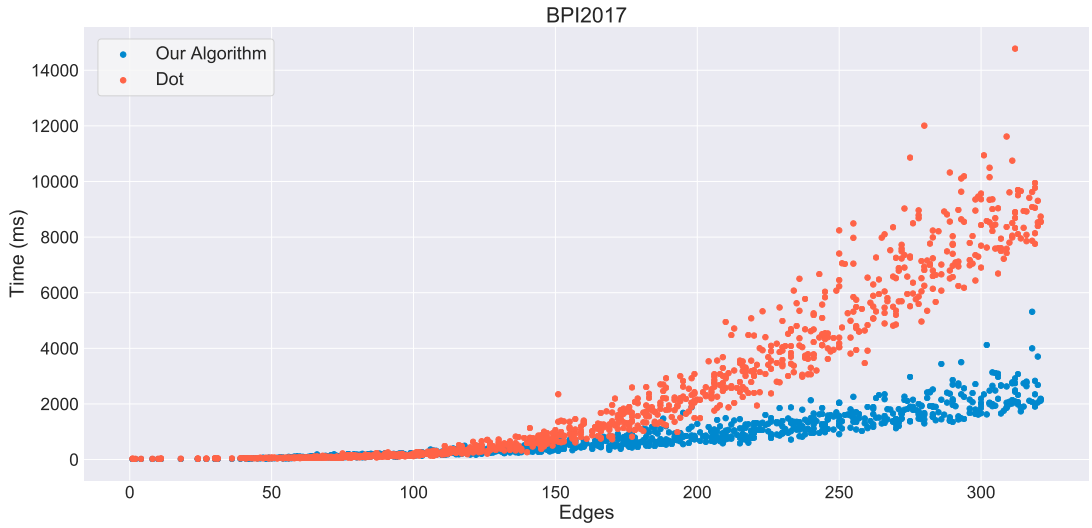

Figure A.4: The running time results for the BPI2017 dataset.

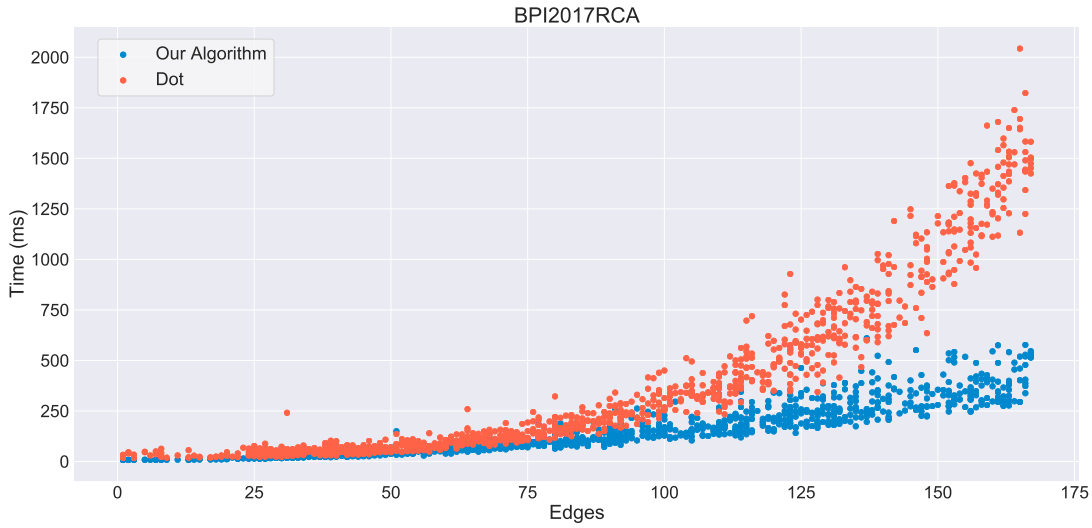

Figure A.5: The running time results for the BPI2017RCA dataset.

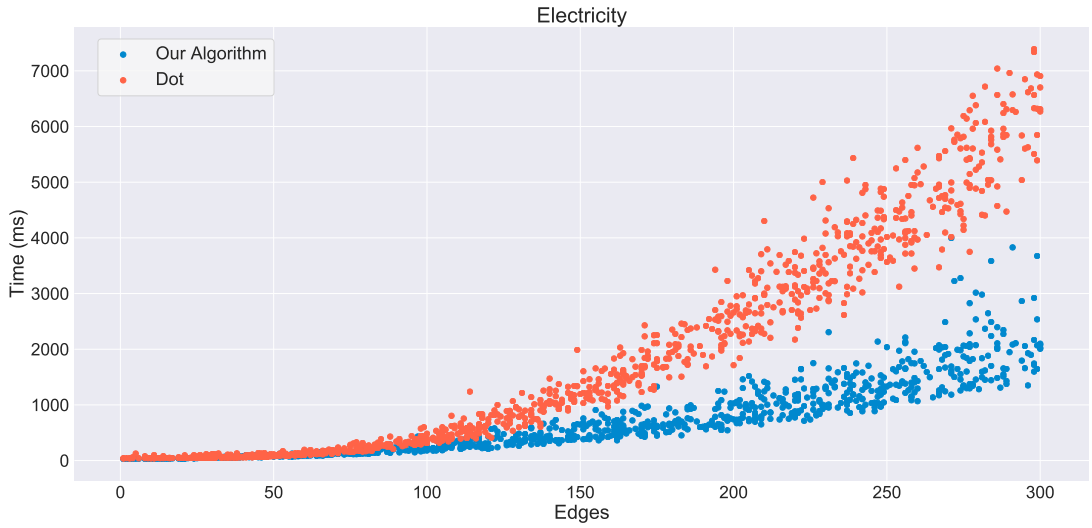

Figure A.6: The running time results for the Electricity dataset.

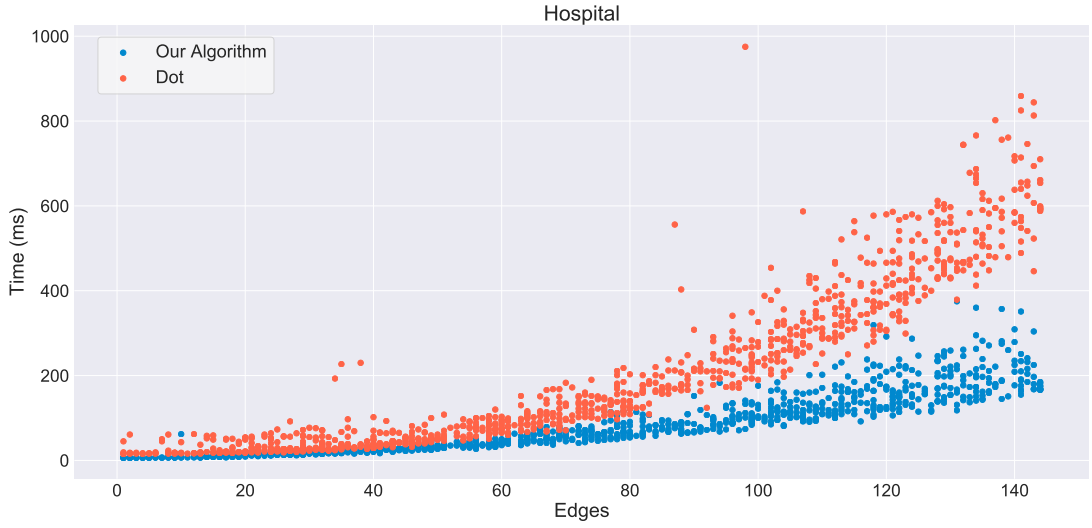

Figure A.7: The running time results for the Hospital dataset.

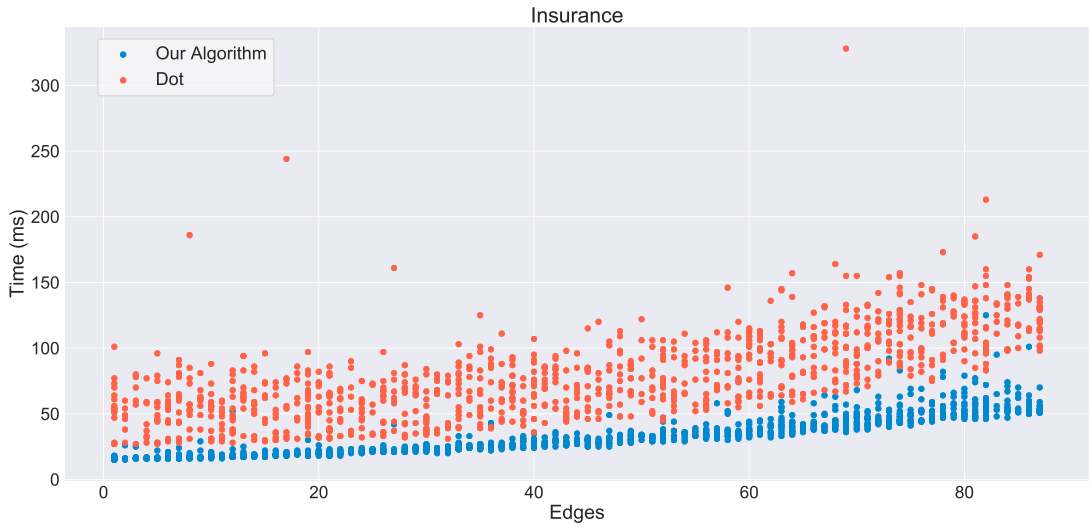

Figure A.8: The running time results for the Insurance dataset.

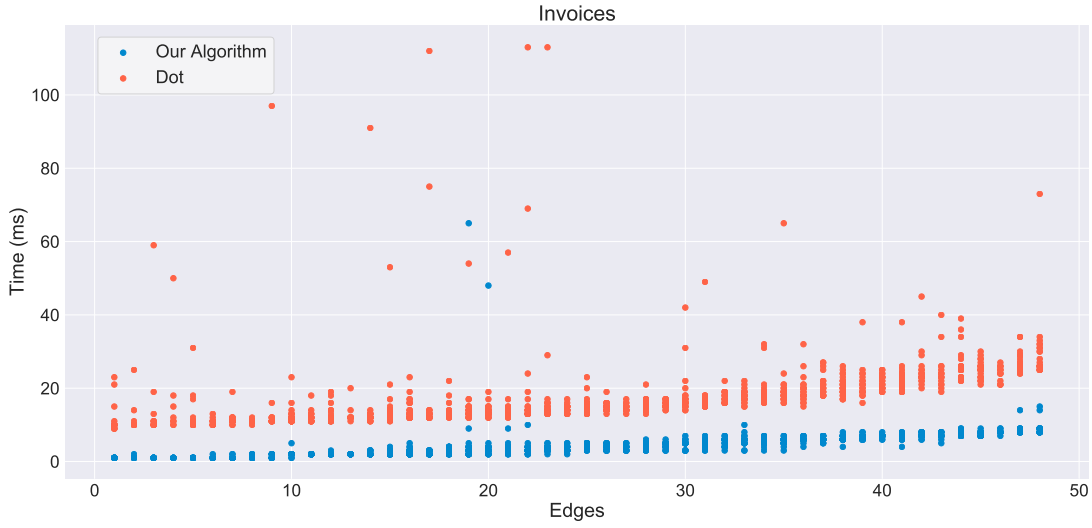

Figure A.9: The running time results for the Invoices dataset.

Figure A.10: The running time results for the P2P dataset.

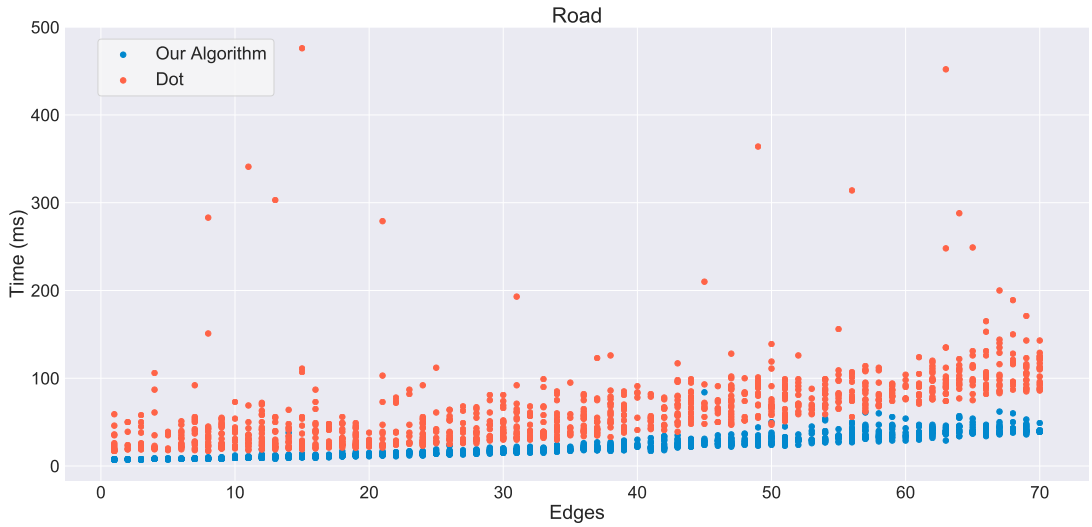

Figure A.11: The running time results for the Road dataset.

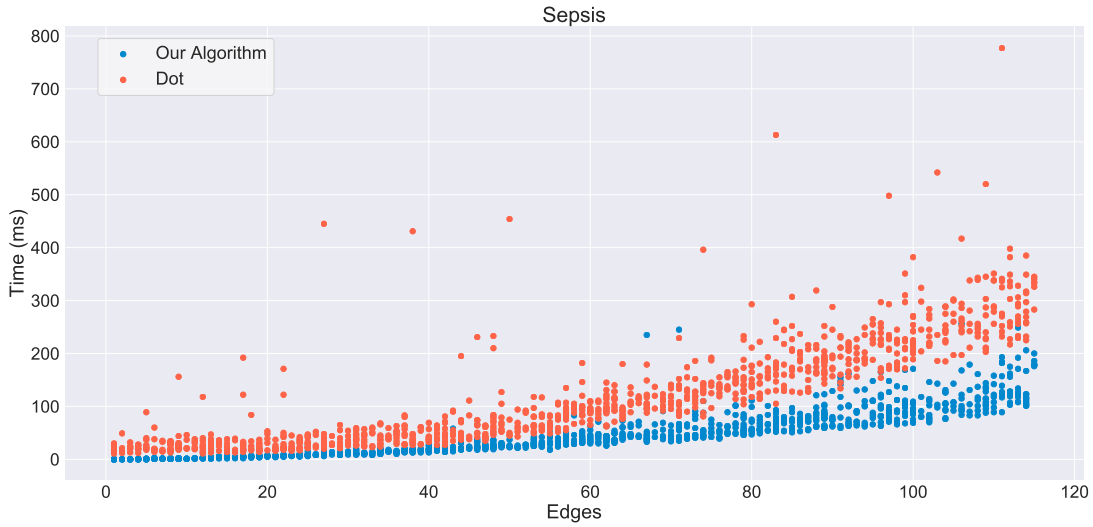

Figure A.12: The running time results for the Sepsis dataset.

Appendix B

Qualitative Evaluation Documents

Hi there!

In this user study you will be working with the process graph dashboard in order to answer questions about two different datasets: a dataset (named ROAD) of an information system managing road traffic fines and a dataset (named INSURANCE) containing process data of an insurance company. In total, there are three different 'algorithms' (named A, B, and C) that 'run' the process graph dashboard. In order to identify which algorithm you prefer the most, you will answer the questions three times, each time using a different algorithm. Note that, after answering the questions the first time, you already (think you) know the answer to every question. Nevertheless, we want you to reperform the steps you took to answer the questions. The goal is to let you work with every algorithm. After answering all the questions using each of the algorithms, there will be some questions about your experience.

Note, you are only allowed to use the process graph dashboard. No other dashboards may be used. Additionally, you may not add any extra filters. All required filters are already present.

The questions for the two datasets are listed below. *Italic* text is used to indicate filter values and underlined text is used to indicate activity names.

ROAD:

- **R1:** which sequence of activities appears to be the most frequent/important?
- **R2:** are all of these most frequent/important activities executed for *Jurisdiction related* activity types?
- **R3:** are all of these most frequent/important activities executed for *Fine related* activity types?
- **R4:** for the years after *2005* (not including 2005). In which year is Payment directly followed by Payment the most?
- **R5:** which activity is executed in *2007* but not in *2008*?

INSURANCE:

- **I1:** which sequence of activities appears to be the most frequent/important?

- **I2:** which activities occur in *2017* but not in *2018*?
- **I3:** for which age group is Re-assessment directly followed by Payment the most?
- **I4:** at which office is Correct payment directly followed by Correct payment the most?
- **I5:** at this office, for which age group is Correct payment directly followed by Correct payment the most?

In the opened ProcessGold application, you should see the following favorites in arbitrary order:

- ROAD A
- ROAD B
- ROAD C
- INSURANCE A
- INSURANCE B
- INSURANCE C

Each of these favorites links to a process graph dashboard that visualizes one of the datasets (ROAD or INSURANCE) using one of the algorithms (A, B, or C). You should open the favorites one by one from top to bottom. For every favorite that you open, you answer the questions for the respective dataset. Answers to the questions should be written in the provided answer tables (make sure you use the table that belongs to the algorithm you are using).

After answering the questions using all three algorithms, please answer the questions in the "User study evaluation table". After answering these questions, you are done with the experiment.

Thank you for your participation!

Answers to questions for algorithm A	
Question	**Answer**
R1	
R2	
R3	
R4	
R5	
I1	
I2	
I3	
I4	
I5	

Table B.1: User study questions for algorithm A. Fill in your answer.

Answers to questions for algorithm B	
Question	**Answer**
R1	
R2	
R3	
R4	
R5	
I1	
I2	
I3	
I4	
I5	

Table B.2: User study questions for algorithm B. Fill in your answer.

Answers to questions for algorithm C	
Question	**Answer**
R1	
R2	
R3	
R4	
R5	
I1	
I2	
I3	
I4	
I5	

Table B.3: User study questions for algorithm C. Fill in your answer.

User study evaluation table	
Question	**Answer**
In general, which algorithm do you prefer the most?	A / B / C / No preference
Why?	
For which algorithm did you find the processes to be the most readable/easiest to follow/understandable?	A / B / C / No preference
Why?	
Your interaction with the process graph dashboard changed the process being visualized. For which algorithm was this change the easiest to follow?	A / B / C / No preference
Why?	
Algorithms A and B use animation. Did the animation help you answer the questions?	Yes / No / No preference
If yes, for which algorithm did the animation help the most?	A / B / No preference

Table B.4: User study evaluation questions. Encircle/fill in your answer.

Appendix C

Qualitative Evaluation Answers

Figure C.1 contains an overview of the answers given to the questions about the road dataset (except R1). As we can see, most answers are correct. During the user test, we noticed that most incorrect answers were given due to misunderstanding the question, or because the participant simply wrote down something different than what he/she actually intended. For example, we observed one participant who concluded that 2007 is the correct answer for R4. Nevertheless, he/she wrote down 2008 instead.

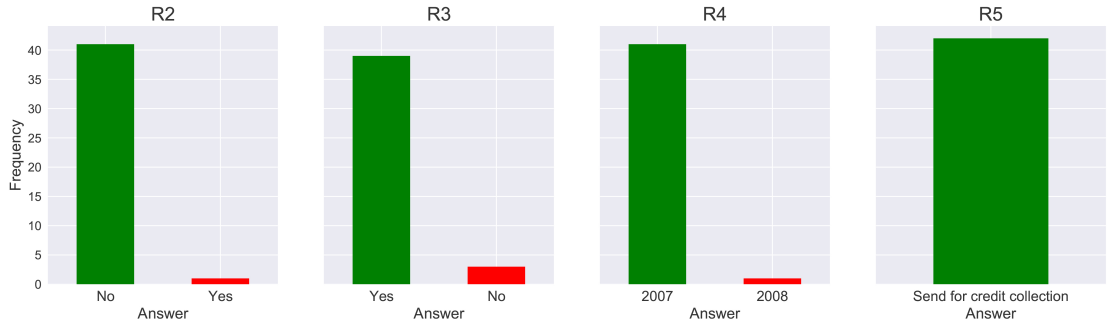

Figure C.1: An overview of the answers given to the questions about the road dataset (except for R1, see Figure C.3). Green bars indicate correct answers while red bars indicate incorrect answers.

Figure C.2 contains an overview of the answers given to the questions about the insurance dataset (except I1 and I2). As we can see, similar to the road questions, most answers are correct. Finally, Figure C.3 contains the answers to R1, I1, and I2. For R1 and I1, the bars are colored blue because R1 and I1 do not necessarily have a single correct answer. For R1, we can see that most participants identified the same sequence of activities. For I1, on the other hand, there is a lot of variation in which sequence is identified. This is mainly because in the insurance dataset, there is no clear 'main path'. Finally, all participants correctly answered question I2.

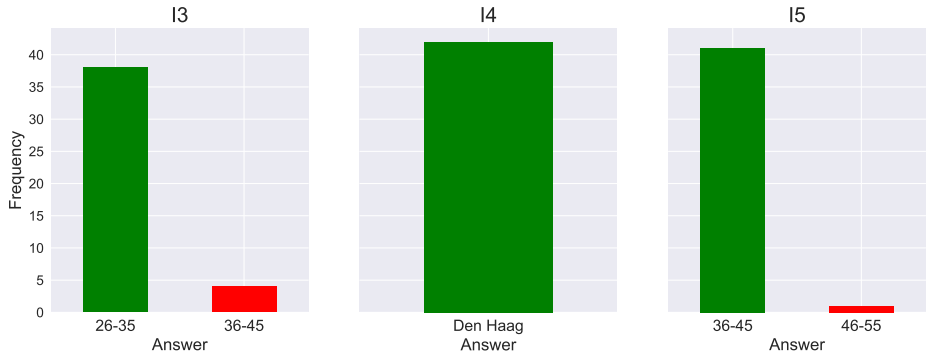

Figure C.2: An overview of the answers given to the questions about the insurance dataset (except for I1 and I2, see Figure C.3). Green bars indicate correct answers while red bars indicate incorrect answers.

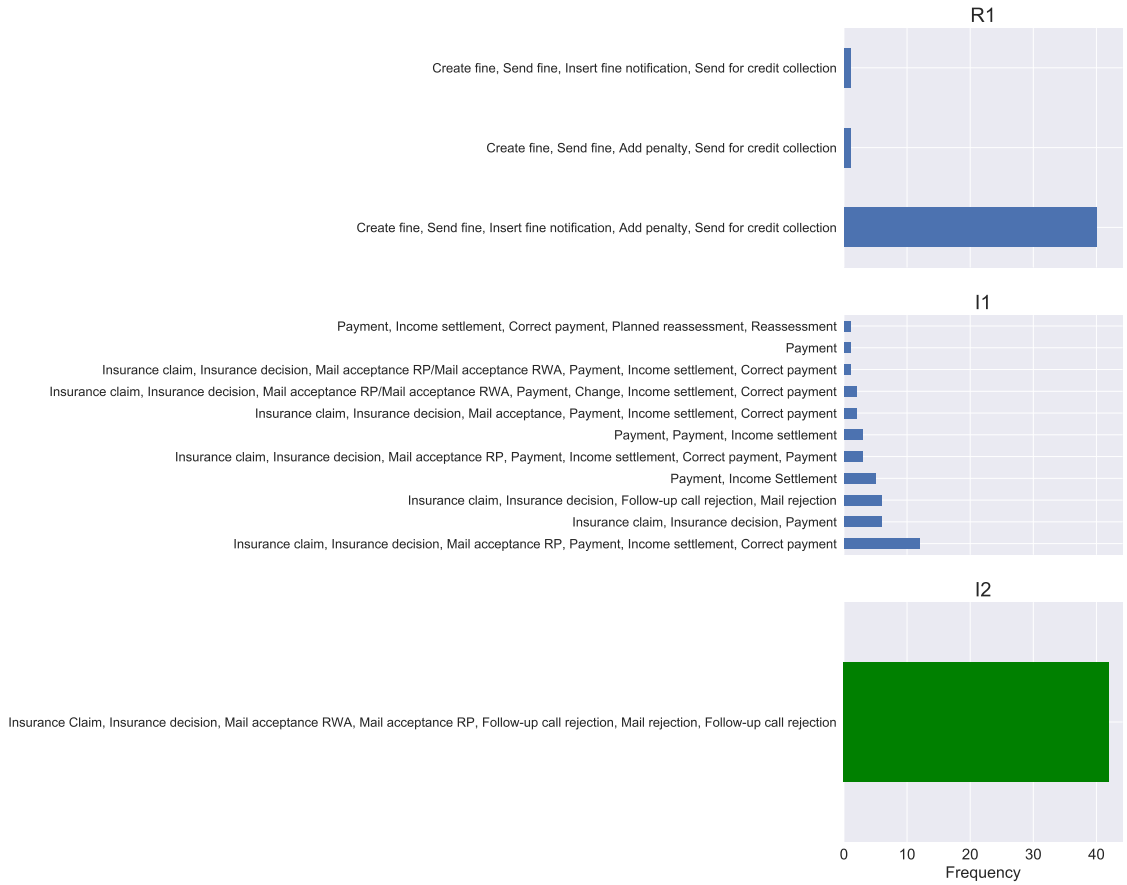

Figure C.3: An overview of the answers given to R1, I1, and I2. Green bars indicate correct answers while red bars indicate incorrect answers. Blue bars are neutral in the sense that there is no single correct answer for the respective question.

Appendix D

Layout Examples

This appendix contains figures in which two graph layouts of the same graph are shown. The graph layout on the left is always computed by *dot* while the graph layout on the right is computed by our algorithm.

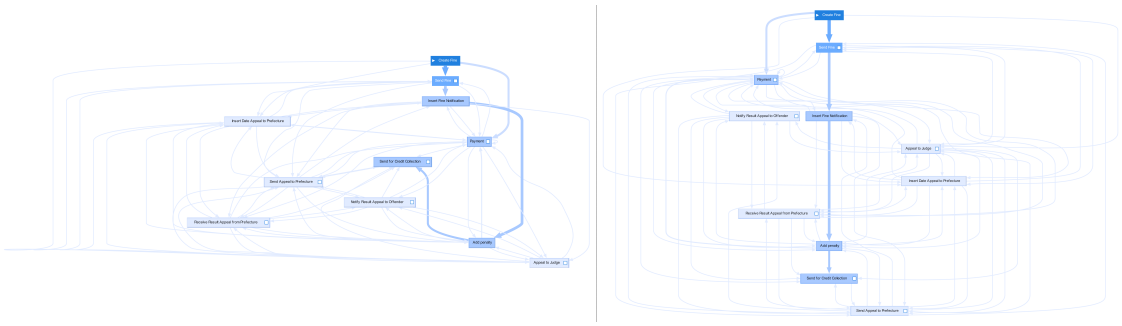

Figure D.1: Graph layouts for the complete Road dataset (which was used during the user study). In our computed layout, the 'main path' is in the center and aligned, making it easier to understand the process. In the layout computed by *dot*, this is not the case. Also, in our layout, the edges are more straight, making them easier to follow.

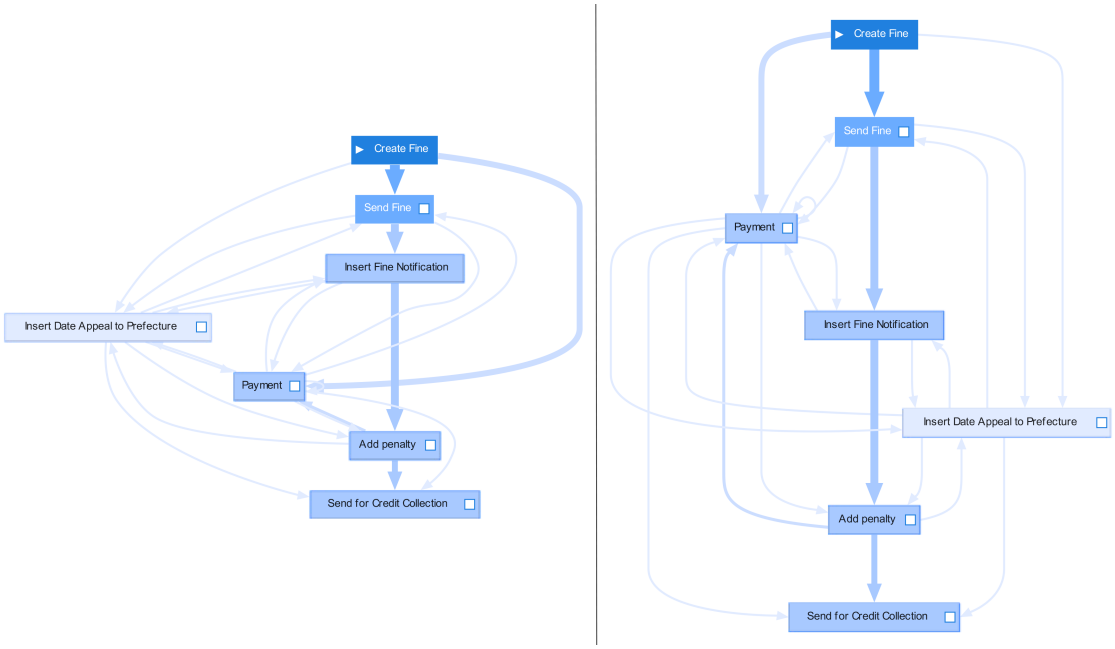

Figure D.2: Graph layouts for a subset of the Road dataset (which was used during the user study). In both layouts, the 'main path' is nicely aligned. In the layout computed by *dot*, however, the (salient) edge (*Create fine*, *Payment*) crosses the main path, which is quite distracting.

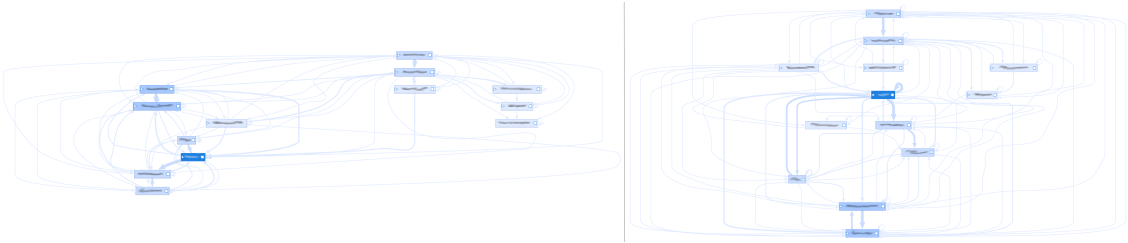

Figure D.3: Graph layouts for the complete Insurance dataset (which was used during the user study). Due to data privacy, node names have been blurred. In this dataset, there is no clear 'main path'. Nevertheless, in our graph layout, the edges are more straight, making them easier to follow. Also, the layout computed by *dot* is unnecessarily wide.

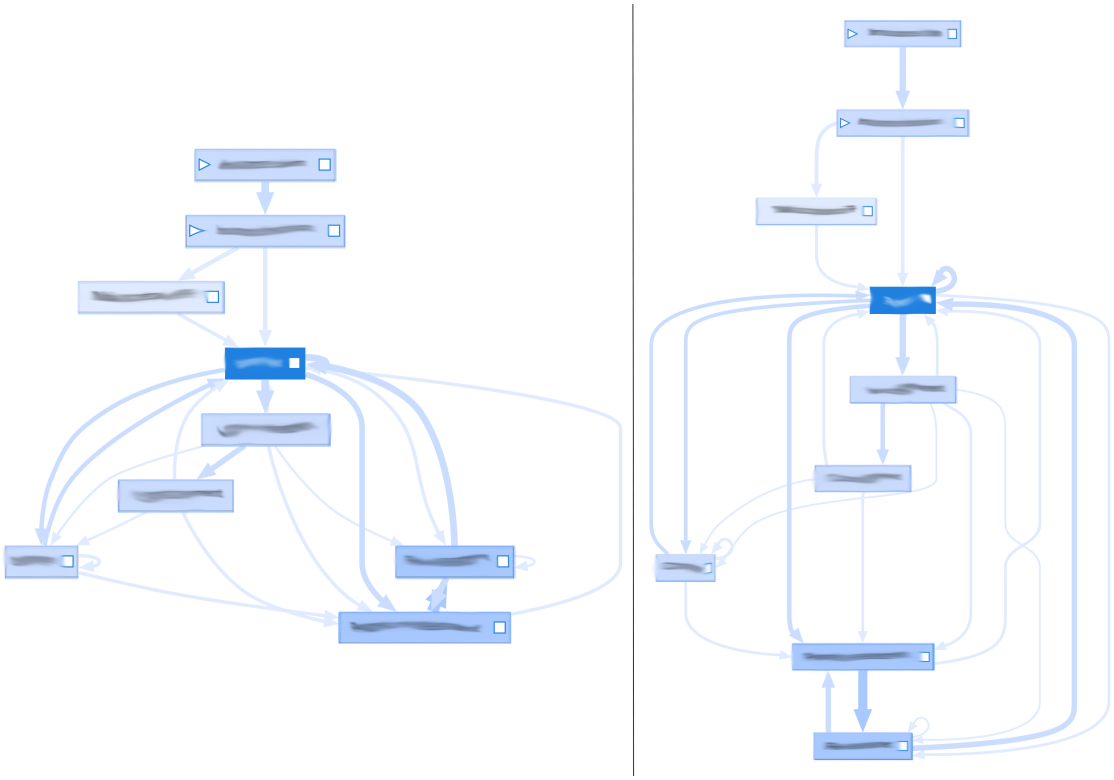

Figure D.4: Graph layouts for a subset of the Insurance dataset (which was used during the user study). Due to data privacy, node names have been blurred. Since this graph is not too large, both layouts are (quite) readable and understandable.

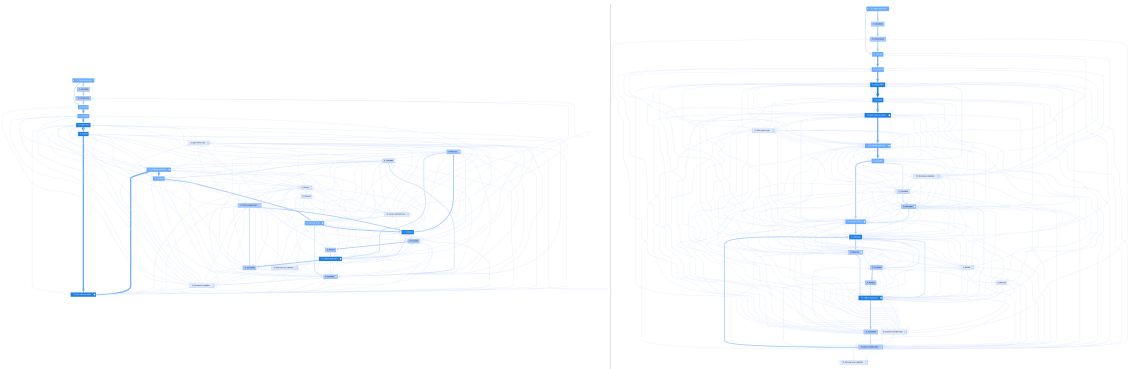

Figure D.5: Graph layouts for the complete BPI2017RCA dataset. In our computed layout, the 'main path' is nicely in the center and aligned, making it easier to understand the process. In the layout computed by *dot*, the beginning of the 'main path' is nicely aligned. After that, however, it is difficult to follow the main path.

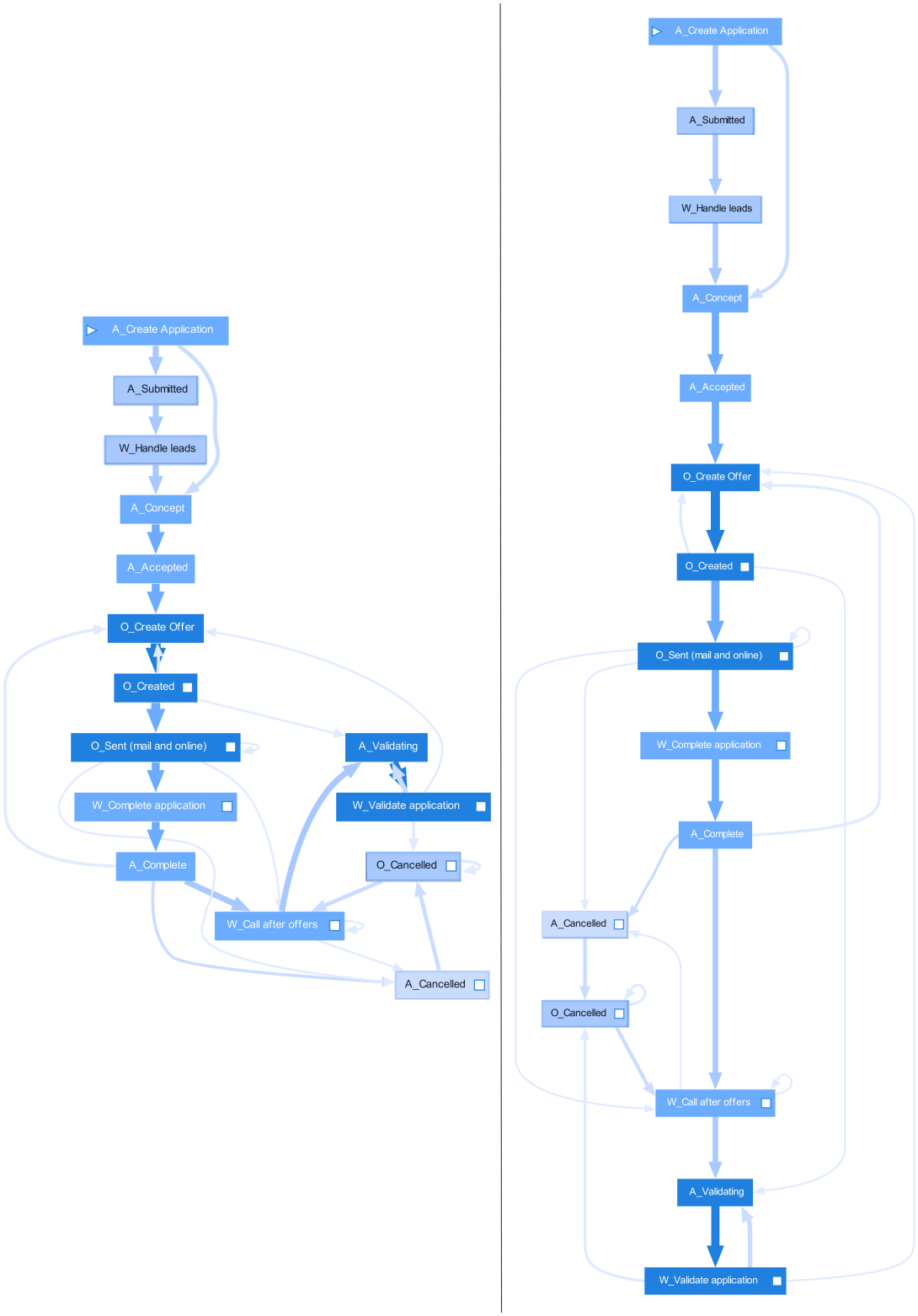

Figure D.6: Graph layouts for a subset of the BPI2017RCA dataset. In our computed layout, the 'main path' is nicely in the center and aligned, making it easier to understand the process. In the layout computed by *dot*, the beginning of the 'main path' is nicely aligned. After that, however, it is difficult to follow the main path.

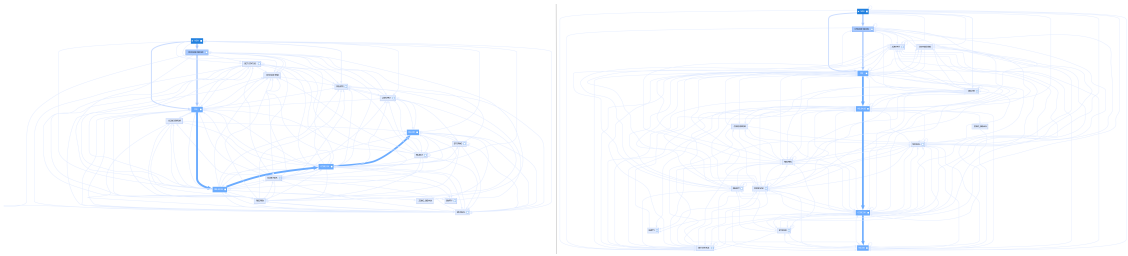

Figure D.7: Graph layouts for the complete Hospital dataset. In our computed layout, the 'main path' is nicely in the center and aligned, making it easier to understand the process. In the layout computed by *dot*, the 'main path' is not centralized and aligned.

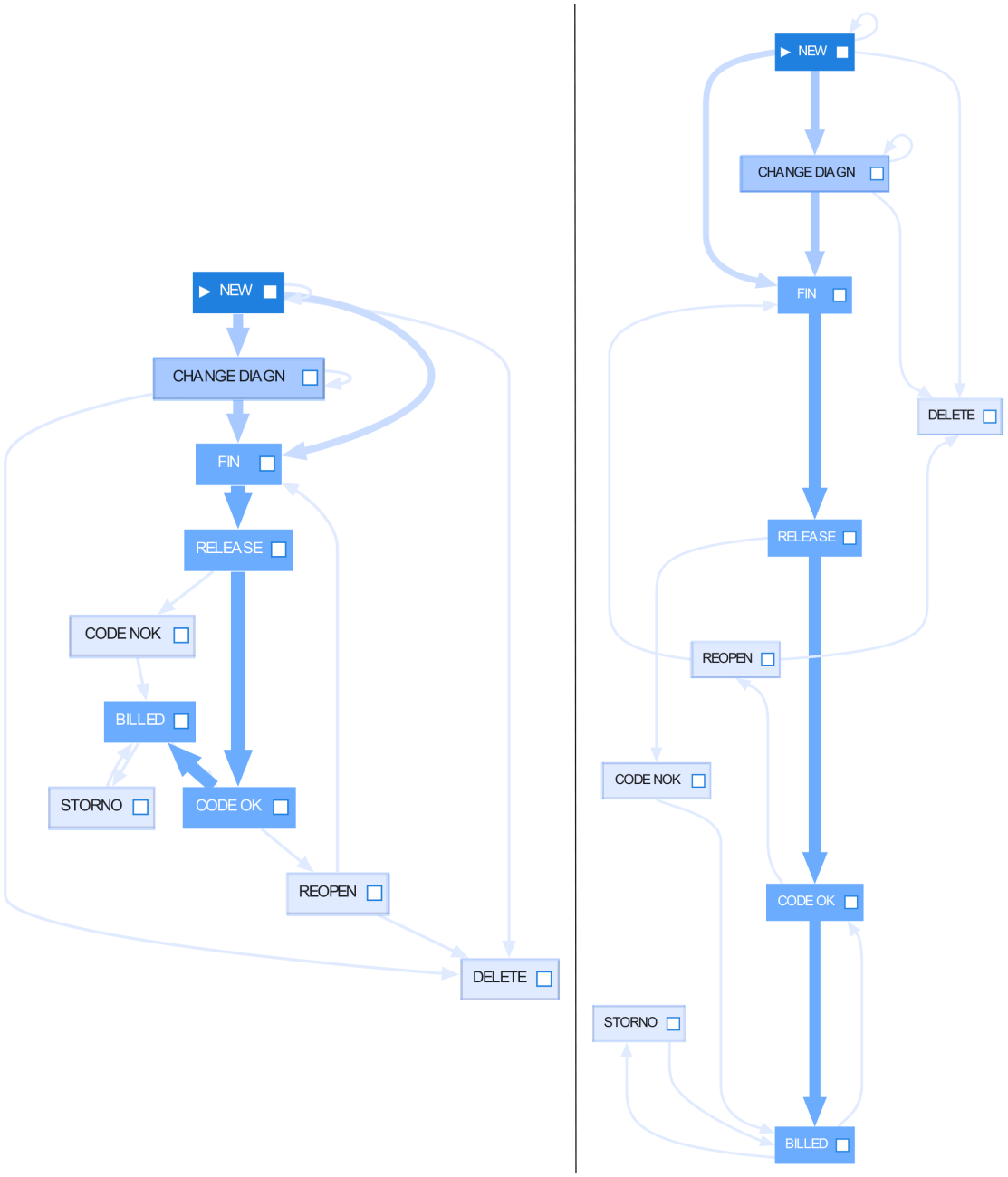

Figure D.8: Graph layouts for a subset of the Hospital dataset. In our computed layout, the 'main path' is nicely in the center and aligned (which is not the case in the layout computed by *dot*), making it easier to understand the process. The area of our layout, however, is quite large compared to the layout of *dot*.